Peter Herrmann, Heiko Krumm
Universität Dortmund, Fachbereich Informatik
D-44221 Dortmund
Internet: {herrmann|krumm}@ls4.cs.uni-dortmund.de
Tel.: 0231-7554674, Fax: 0231-7554730

In *Praxis der Informationsverarbeitung und Kommunikation*, 21(1998)2, 79-88

Protokollspezifikation und -verifikation mit dem Transferprotokoll-Framework

Kurzfassung

Das Transferprotokoll-Framework ist eine Sammlung von Bausteinen zur Unterstützung der formalen Spezifikation und Verifikation von Datentransfer-Kommunikationsprotokollen. Durch Kombination von Spezifikationsbausteinen können Dienste und Protokolle in komfortabler Weise modelliert werden. Der Nachweis, daß ein modelliertes Protokoll einen gewünschten Dienst mit den spezifizierten Eigenschaften erbringt, wird durch Kombination von Theoremen des Frameworks unter vergleichsweise geringem Aufwand erbracht. Der Beitrag stellt die Anwendung des Frameworks in den Mittelpunkt. Wir erläutern dazu die Spezifikation und Verifikation eines Sliding-Window-Protokolls.

Die im Framework benutzte Spezifikations- und Verifikationstechnik basiert auf L. Lamports Temporal Logic of Actions TLA bzw. auf einer speziell für Prozeßsysteme geeigneten Variante namens cTLA. Sie werden hier nur soweit vorgestellt, wie es für das Verständnis der Beispielanwendung notwendig ist.

1 EINLEITUNG

Aktuelle Entwicklungen im Bereich der Hochleistungs- und Multimediakommunikation umfassen häufig an zentraler Stelle auch den Entwurf neuer Transferprotokolle und Protokollvarianten. Obwohl inzwischen sogar standardisierte formale Beschreibungsmittel zur Verfügung stehen (z.B. ISO/OSI: Estelle [23] und Lotos [24], ITU: SDL [25]), werden die Protokolle üblicherweise nur nicht-formal spezifiziert. Darüberhinaus entfallen oft separate abstrakte Beschreibungen der durch die Protokolle erbrachten Kommunikationsdienste (nach ISO/OSI sogenannte Dienstspezifikationen), welche die für die Nutzer (bzw. Anwendungsentwickler) relevanten Diensteigenschaften dokumentieren (vgl. z.B. XTP [31]). So bleiben Entwurfsdokumentationen leider allzuhäufig lückenhaft und mehrdeutig. Mangels Ansatzpunkten unterbleiben systematische Betrachtungen zur Korrektheit der Protokolle.

Die Nachteile dieses Vorgehens sind wohl unbestritten. Spät erkannte Entwurfsfehler sowie durch fehlende präzise Dokumentationen verursachte Mißverständnisse der Implementierer und Anwendungsentwickler können zu hohem Mehraufwand und starken Verzögerungen eines Projekts führen. Andererseits verursacht aber auch der Einsatz formaler Techniken einen bedeutenden Aufwand. Obwohl es Werkzeugsysteme für die rechnergestützte Anwendung der Techniken gibt (z.B. spezielle Editoren, Interpreter, Compiler, Verifikationstools; vgl. [2, 4, 8, 10, 14, 22]), bleibt ein nennenswerter Aufwand, der durch den notwendigen kreativen Entwurf formaler Modelle und die Entwicklung ihrer formalen Beschreibung bedingt ist. Wie bei einer Programmentwicklung treten üblicherweise auch bei der Entwicklung formaler Spezifikationen die unterschiedlichsten Entwurfsfehler auf, die eine längere Phase des Debuggens von Spezifikationen erfordern können, bevor ein Protokoll systematisch auf seine Korrektheit geprüft werden kann und die Dokumentationen an Implementierer und Anwendungsentwickler weitergegeben werden können.

Das Transferprotokoll-Framework [12, 15, 16] unterstützt die zügige und zielgerichtete Entwicklung formaler Spezifikationen in einer Weise, die der Unterstützung von Implementierungsprojekten durch Bibliotheken wiederverwendbarer Programmbausteine vergleichbar ist: Es enthält Spezifikationsbausteine. Ein Entwickler bildet Spezifikationen aus den Bausteinen durch Instantiierung und Kombination. Eine Protokollspezifikation wird aus Bausteinen zusammengesetzt, die einzelne Protokollmechanismen modellieren (z.B. Sequenznumerierung von Protokolldateneinheiten, Wiederholungsanforderung, Zeitüberwachung). Der Entwickler muß die einzelnen Mechanismen nicht mehr im Detail modellieren. Er verwendet die vorgegebenen Bausteine und konzentriert sich auf ihre Parametrisierung und Kombination. Solcherart modelliert er sehr direkt die logische Struktur seines Protokolls. Der Entwickler ist weiterhin angehalten, neben der Protokollspezifikation auch eine Dienstspezifikation zu erstellen. Auch hierzu stellt das Framework Spezifikationsbausteine zur Verfügung. Sie modellieren einzelne Eigenschaften des vom Protokoll zu erbringenden Dienstes (z.B. Verfälschungsfreiheit der Nutzdatenübertragung, Lebendigkeit der Nutzdatenübertragung, Reihenfolgetreue, Verlustbeschränkung). Die Dienstspezifikation kombiniert die gewünschten Eigenschaften in einer vom Entwickler gewählten Auswahl und Parametrisierung.

Die Existenz einer separaten Dienstspezifikation ermöglicht die Protokollverifikation. Sie entspricht dem formalen Nachweis, daß das modellierte Protokoll einen Dienst mit den in der Dienstspezifikation genannten Eigenschaften erbringt, und ist ein sehr wirksames Mittel zur Entdeckung von Entwurfsfehlern. Die Protokollverifikation kann bei Protokollmodellen, die eine endliche und vergleichsweise geringe Zustandsanzahl haben, automatisch mithilfe von Erreichbarkeitsanalyse-basierten Werzeugen durchgeführt werden (z.B. State Space Exploration [21], Model Checker [5, 6, 9, 11, 20]). Bei den meisten praktisch interessanten

Protokollen ist die Zustandsanzahl aber zu hoch, so daß entweder verschiedene Vereinfachungen entwickelt werden müssen oder die Verifikation durch symbolisch logische Beweisführung geleistet werden muß. Beides kann zu sehr hohem Entwurfsaufwand führen.

Das Transferprotokoll-Framework kann, weil Dienst- und Protokollspezifikationen aus vorgegebenen Elementen des Frameworks gebildet sind, eine besondere Verifikationsunterstützung bieten: Theoreme, die zum Ausdruck bringen, daß bestimmte Kombinationen von Protokollmechanismen bestimmte Diensteigenschaften implizieren. Die Theoreme sind bereits bewiesen. Deshalb verbleibt dem Protokollentwickler zur Verifikation nur noch die Aufgabe, für jede Diensteigenschaft seiner Dienstspezifikation im Framework ein Theorem zu suchen, das unter geeigneter Parametrisierung die Diensteigenschaft mit einer Teilmenge der Protokollmechanismen verknüpft. Auf diese Weise können auch komplexere Transferprotokolle mit vergleichsweise geringem Aufwand verifiziert werden [18]. Darüberhinaus zeigt diese Verifikation sehr deutlich die logischen Zusammenhänge zwischen Protokollbestandteilen und erbrachtem Dienst auf und trägt so zum verbesserten Verständnis des entworfenen Protokolls bei.

Im einzelnen sind im Framework nicht nur Spezifikationsbausteine für Dienste und Protokolle enthalten. Es gibt noch eine dritte Klasse von Spezifikationsbausteinen, die abstrakte Protokollmechanismen genannt werden. Die Theoreme des Frameworks verknüpfen Protokollmechanismen und Diensteigenschaften nicht direkt. Stattdessen gibt es Theoreme, die zum Ausdruck bringen, daß einzelne abstrakte Protokollmechanismen durch bestimmte Kombinationen von detaillierteren Protokollmechanismen implementiert werden, sowie solche, die Kombinationen abstrakter Protokollmechanismen mit Diensteigenschaften verknüpfen. Die Zwischenschicht der abstrakten Protokollmechanismen wurde eingeführt, weil sich bei direkter Verknüpfung von Protokoll- und Dienstebene eine sehr hohe und unübersichtliche Anzahl von Theoremen ergeben hätte.

Bausteine, das Zusammensetzen von Bausteinen zu Systemmodellen und die besondere Form der Kompositionsfähigkeit der Bausteine spielen eine zentrale Rolle im Framework. Die verwendete Spezifikationstechnik gewährleistet, daß die relevanten Eigenschaften eines Bausteins auch in einem System vorhanden sind, das den Baustein enthält, und ist die Grundlage zur Strukturierung der Protokollverifikation in einzelne Theorem-Anwendungen. Das Framework nutzt dazu die auf TLA (Temporal Logic of Actions [26]) aufbauende Spezifikationstechnik cTLA [15, 28]. Spezifikationsbausteine des Frameworks sind cTLA-Module, die generische Prozeßtypen definieren. Durch Instantiierung entstehen Prozesse, die zu Prozeßsystemen zusammengesetzt werden können. Ähnlich zur Standardspezifikationssprache Lotos [24] interagieren die Prozesse eines cTLA-Systems über gemeinsame (synchrone) Aktionen und es können sowohl Prozesse mit Constraint-Charakter als auch Prozesse mit Implementierungskomponenten-Charakter beschrieben werden (vgl. dazu [30]).

Dieser Beitrag möchte bewußt nur von einem pragmatischen Anwenderstandpunkt aus auf die Spezifikationstechnik und ihre Semantik eingehen. Er soll vorrangig die Grundzüge der Anwendung des Frameworks beim Protokollentwurf erläutern und anhand eines Beispiels (Sliding-Window-Protokoll nach [29]) verdeutlichen, wie das Framework vorteilhaft in konkreten Protokollentwicklungsprojekten eingesetzt werden kann. Im folgenden wird deshalb nur eine sehr kurze Einführung in die Spezifikationstechnik cTLA gegeben. Ihr folgt ein ebenfalls knapp gehaltener Überblick über das Transferprotokoll-Framework, so daß mehr Raum für die drei Hauptabschnitte bleibt, welche die Dienstspezifikation, die Protokollspezifikation und die Protokollverifikation am Beispiel des Sliding-Window-Protokolls vorstellen.

```
PROCESS C (usd : Any)     Safety constraint: No corruptions
 IMPORT Symbols;                usd : data type „user data“
 BODY
  VARIABLES
   buf : SUBSET(key: usd);     all data units ever sent
  INIT ≡ buf=∅;
   ACTIONS
    Rq (krq : key; d : usd) ≡
              Transmission of user data d with sequence number 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 .
```

Abb. 1: Safety-Prozeß *C*

2 SPEZIFIKATIONSTECHNIK

Als Bausteine einer zusammengesetzten cTLA-Spezifikation dienen Prozesse. Ein Prozeß wird dabei durch ein Zustandsübergangssystem modelliert, das mit Programmiersprach-artigen Konstrukten definiert wird. Ein Beispiel liefert die Definition des Prozesses *C* in Abb. 1. Das Konstrukt `IMPORT` referenziert in anderen Modulen definierte Symbole (z.B. Datentypen, Funktionen, Konstanten). Im Beispiel seien im Modul *Symbols* die beiden Symbole `key` als Wertemenge für Laufnummern und `usd` als Wertemenge für Nutzdaten vereinbart. Das Schlüsselwort `VARIABLES` leitet die Deklaration der Zustandsvariablen des Prozesses ein. Der Zustandsraum des Prozesses C wird hier durch eine einzige Variable namens `buf` aufgespannt. `buf` kann eine Menge von Paaren aus Laufnummer und Nutzdatum enthalten. Das Schlüsselwort `INIT` bezeichnet eine Bedingung über den Variablen eines Prozesses, welche die zulässigen Startzustände definiert. Alle Variablenbelegungen, die `INIT` erfüllen, sind mögliche Startzustände des Prozesses. Im Beispiel besagt `INIT`, daß die Variable `buf` bei Prozeßstart mit der leeren Menge belegt ist. Das Schlüsselwort `ACTIONS` leitet die Vereinbarung der Aktionen des Prozesses ein. Jede Aktion definiert dabei eine Menge möglicher Zustandsübergänge des Prozesses durch eine Bedingung über Variablen (im Beispiel `buf`) und sogenannten gestrichenen Variablen (im Beispiel `buf'`). Die Variablen sprechen den Momentanzustand, die gestrichenen Variablen den Folgezustand einer Transition an. Alle Paare aus Momentan- und Folgezustand, welche die Bedingung einer Aktion erfüllen, sind der Aktion entsprechende Transitionen. Aktionsdefinitionen können mit Datenparametern versehen sein. Im Beispiel entsprechen alle Transitionen der Aktion `Rq`, für die eine Laufnummer `krq` und ein Nutzdatum `d` so gefunden werden können, daß die Variable `buf` im Folgezustand gerade die Paare aus dem Momentanzustand ergänzt um das Paar (`krq`,`d`) enthält. Auf diese Weise definieren die Variablen, `INIT` und die Aktionen eines Prozesses ein Zu-

standsübergangssystem. Eine Zustandsfolge stellt einen möglichen Ablauf des Prozesses dar, wenn sie mit einem Zustand gemäß INIT beginnt und wenn alle Zustandsänderungen einer Prozeßaktion entsprechen.

Der Beispielprozeß *C* ist im übrigen schon ein Element des Frameworks. Er beschreibt die Eigenschaft der Nutzdatenverfälschungsfreiheit eines Transferdienstes. Die Aktion Rq modelliert dazu Transferanforderungen, die Aktion In Transferanzeigen des Dienstes. Die Variable buf führt Buch über aufgetretene Anforderungen, so daß der Term (krq,d)∈buf in der Definition der Aktion In jeweils auf alle Laufnummern-Nutzdaten-Paare zutrifft, die bisher per Anforderung Rq an den Dienst übergeben wurden. Der Prozeß *C* beschreibt ausschließlich Sicherheitseigenschaften (im Sinne der Trennung von Safety und Liveness nach [1]). Somit sind Abläufe nicht ausgeschlossen, in denen irgendwann keinerlei Zustandsänderungen mehr auftreten, der Prozeß also stoppt.

```
PROCESS LIn          Service Constraint: Live delivery of user data
 IMPORT Symbols;
 BODY
  VARIABLES
    cRq : key ;      Sequence no. of next data unit to be sent
    maxIn : key ;    Sequence no. 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) ≡           Fair delivery of user data
     krq≠"notsent" ∧ krq=maxIn ∧ maxIn<cRq ∧
     maxIn'=maxIn+1 ∧ cRq'=cRq;
   nIn ( krq : key ) ≡         Other delivery of user data
     not(krq≠"notsent" ∧ krq=maxIn ∧
         maxIn<cRq) ∧
     maxIn' = IF (krq="notsent") THEN maxIn
              ELSE max(maxIn,krq+1) ∧
     cRq'=cRq;
 WF: fIn;                    Weak fairness assumption for fIn
 END .
```

Abb. 2: Liveness-Prozeß *LIn*

Der in Abb. 2 aufgeführte Prozeß LIn ist ein Beispiel für die Beschreibung einer Lebendigkeitseigenschaft im Framework. Im Vergleich zum vorherigen Prozeß *C* kommt bei LIn ein neues Konstrukt WF hinzu. Der Rest definiert wiederum ein Zustandstransitionssystem. Das WF-Konstrukt bringt nun zum Ausdruck, daß die Aktion fIn in Prozeßabläufen mindestens „weak-fair" schalten soll. Ähnlich können Aktionen auch mit einem SF-Konstrukt als „strong-fair" deklariert werden. Eine weak-faire Aktion darf in einem Ablauf nicht unendlich lange ununterbrochen schaltbar sein, ohne auch tatsächlich zur Ausführung zu kommen. Eine strong-faire Aktion muß auch zur Ausführung kommen, wenn die Schaltbarkeitsphasen durch Zustände unterbrochen werden, in denen die strong-faire Aktion nicht schaltbar ist. Weak-und strong-faire Aktionen wurden in [1] eingeführt und werden auch in TLA [26] und cTLA zur Spezifikation von Lebendigkeitseigenschaften benutzt, da sie im Gegensatz zur direkten Beschreibung von Lebendigkeitseigenschaften garantieren, daß durch die spezifizierte Lebendigkeit keine Widersprüche zu den Safety-Anforderungen auftreten können.

Im Transferprotokoll-Framework werden Lebendigkeitseigenschaften also indirekt durch Transitionssysteme mit Fairnessbehafteten Aktionen dargestellt. Zur Unterstützung der Modularität ist es dabei wichtig, daß die Fairness-Anforderungen möglichst schwach sind. Hierzu werden Aktionen in zwei Aktionen aufgeteilt, in eine Fairness-behaftete (z.B. fIn in Prozeß *LIn*) und eine nicht Fairness-behaftete (z.B. nIn in Prozeß *LIn*). In Prozeß *LIn* modellieren beide Aktionen fIn und nIn Dienstanzeigen zur Auslieferung übertragener Nutzdaten beim Empfänger (analog zur Aktion In in Prozeß *C*). Der Prozeß *LIn* soll die Eigenschaft zum Ausdruck bringen, daß das jeweils nächste in der Reihenfolge fällige Nutzdatum lebendig übertragen wird (Aktion fIn). Er soll möglichst nur diese Eigenschaft zum Ausdruck bringen. Er darf deshalb nicht verbieten, daß auch andere Nutzdaten ausgeliefert werden können, und toleriert dieses mit der Aktion nIn.

In cTLA können ähnlich der Prozeßkomposition in Lotos [24] Prozesse zu Systemen zusammengesetzt werden. Wie in Lotos, werden die Prozesse über gemeinsame, synchron auszuführende Aktionen verbunden, deren Parameter die Übergabe von Datenparametern modellieren. Die Variablen eines Prozesses sind privat. Der Systemzustand ergibt sich als Vektor der Prozeßzustände. Ein Systemablauf ist eine Folge von Systemzuständen. Dabei überführen sogenannte Systemaktionen einen aktuellen Systemzustand in einen Folgezustand. Eine Systemaktion ist dadurch definiert, daß eine Teilmenge der Prozesse gleichzeitig eine gekoppelte Aktion ausführt und daß die anderen Prozesse ihren Zustand nicht verändern, d.h. einen sogenannten Stotterschritt ausführen (Pseudoaktionsname stutter).

```
PROCESS SlidWindService (usd : any )
                                  usd : set of user data transfered
 PROCESSES          The processes of the system
  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
  D : Duplicates (0) ;            No duplications
  P : Phantoms (usd, usd);        No phantoms
  Cap : Capacity (8);             Service capacity of eight data units
  LIn : LiveInNoAttr;             Data units are delivered lively
 ACTIONS                 The system 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) ≡
                                  Fair delivery of user data d with seq.no. krq
    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) ≡
                                  Other delivery of user data d with seq.no. krq
    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 .
```

Abb. 3: Dienstspezifikation *SlidWindService*

Die Spezifikation in Abb. 3 ist ein Beispiel für eine Komposition. Im PROCESSES-Teil werden die Prozesse des Systems vereinbart. Sie werden durch Instantiierung aus Prozeßtypen gebildet. So ist zum Beispiel *Id* eine Instanz des Typs *SDUId* und der schon von oben bekannte Prozeß *C* (Abb. 1) ist eine Instanz des Prozeßtyps *Corruptions* unter der aktuellen Parametrisierung (usd, {(k,k) | k∈usd}). Prozeßtypvereinbarungen entsprechen im übrigen den Prozeßdefinitionen. Sie können jedoch mit generischen Parametern versehen sein, so daß Datentypen, Operationen und Werte eines Prozesses auch erst bei der Instantiie-

rung festgelegt werden können. Im ACTIONS-Teil einer Komposition werden die Systemaktionen als logische UND-Verknüpfung von Prozeßaktionen und Prozeß-Stotterschritten vereinbart. So besteht die Systemaktion Rq daraus, daß die Prozesse *Id*, *C*, *Cap* und *LIn* gleichzeitig jeweils ihre Prozeßaktion Rq ausführen, während die Prozesse *G*, *R*, *D* und *P* an dieser Systemaktion nicht beteiligt sind und nur einen Stotterschritt ausführen.

Die Spezifikationstechnik cTLA besitzt eine besondere Form der Kompositionalität, die für die Konzeption des Frameworks und insbesondere für die Strukturierung der Verifikation in Subsystem-Implikationen wichtig ist. Die mit einem Prozeß beschriebenen Eigenschaften sollen in jedem System, das diesen Prozeß enthält, ebenfalls vorhanden sein. Das wird dadurch gewährleistet, daß die Systembildung einer logischen Konjunktion von Prozessen entspricht, deren Widerspruchsfreiheit sichergestellt ist. Sie besitzt damit übrigens Superpositionseigenschaften, die über die in [3] definierten hinausgehen und auch Lebendigkeitsaspekte erfassen.

Die Kompositionalität der Safety-Eigenschaften ist auf sehr einfache Weise gegeben. Safety-Eigenschaften schränken die Startzustände und die Zustandsübergänge eines Systems ein. Da Prozesse nur private Variablen besitzen und der Systemzustand direkt durch den Vektor der Variablen der Prozesse eines Systems beschrieben wird, gehen die Einschränkungen zur Entwicklung der Zustände eines Prozesses direkt auch auf das System über. Sie können im System nicht durch andere Prozesse gestört werden.

Hinsichtlich der durch Fairness-Anforderungen an Prozeßaktionen beschriebenen Liveness-Eigenschaften ist diese Kompositionalität allerdings nicht unmittelbar erfüllt. Eine Prozeßaktion kann im System nämlich an Aktionen anderer Prozesse gekoppelt sein. Damit können fremde Prozesse das Schalten einer Prozeßaktion blockieren. Die Umgebung eines Prozesses kann also verhindern, daß der Prozeß an seine Aktionen gestellte Fairness-Anforderungen erfüllt. Im Unterschied zu TLA, das direkt die in [1] vorgestellten Fairness-Anforderungen benutzt, verwendet cTLA deshalb nur sogenannte bedingte Fairness-Anforderungen. Sie fordern das faire Schalten einer Prozeßaktion nur für solche Zustände, in denen sowohl die Aktion feuern kann als auch die Umgebung des Prozesses das Schalten der Aktion zuläßt. Die Beschränkung auf derart bedingte Fairness-Anforderungen garantiert einerseits die gewünschte Kompositionalität von Liveness-Eigenschaften. Sie führt aber andererseits dazu, daß zur Formulierung absoluter Liveness-Eigenschaften eine zusätzliche Bedingung an die Einbettung eines Prozesses gestellt werden muß. So besagt der oben vorgestellte Prozeß *LIn*, daß mit Rq übergebene Daten nach endlicher Zeit mittels einer In-Aktion ausgeliefert werden, falls die Umgebung von *LIn* die In-Aktion nicht allzu oft blockiert. Wenn sich in einem System die Safety-Eigenschaft beweisen läßt, daß die faire Prozeßaktion fIn auch immer dann von der Umgebung her erlaubt ist, wenn sie aufgrund des Prozeßzustands von *LIn* schalten kann, dann entspricht die bedingte Fairness der Aktion *LIn* ihrer unbedingten Fairness und die eigentlich aus Anwendungssicht gewünschte absolute Liveness ist gegeben.

3 TRANSFERPROTOKOLL-FRAMEWORK

Das Transferprotokoll-Framework besteht aus Spezifikationsbausteinen und Theoremen (siehe Abb. 4). Als Spezifikationsbausteine werden cTLA-Prozeßtypdefinitionen verwendet. Sie beschreiben Protokollmechanismen, Eigenschaften benutzter Medien sowie Eigenschaften erbrachter Kommunikationsdienste und sind in drei Schichten gegliedert:

- Service-Constraints (SCs): Die SCs modellieren die einzelnen Eigenschaften des durch das interessierende Protokoll zu erbringenden Datentransferdienstes. Die Spezifikation dieses Zieldienstes wird durch ein System aus SC-Instanzen gebildet. Das in Abb. 3 vorgestellte System *SlidWindService* gibt dazu ein Beispiel. Die dort vorkommenden Prozeßtypen (z.B. *Corruptions*, *Gaps*, *Reorderings*) sind Beispiele für SCs.

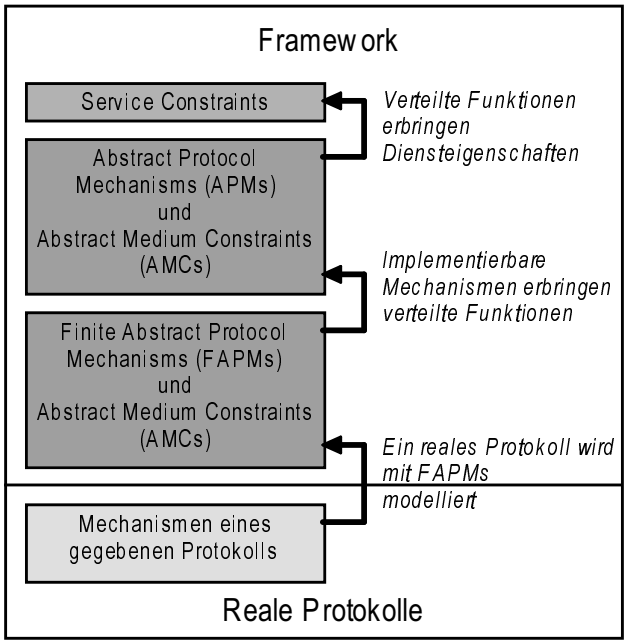

Abb. 4: Aufbau des Transferprotokoll-Frameworks

- Abstract Protocol Mechanisms (APMs) und Abstract Medium Constraints (AMCs): Ein System aus APMs und AMCs definiert ein sogenanntes abstraktes Protokoll. Wie bei einer üblichen Protokollbeschreibung liegt dazu ein Szenario aus Protokollinstanzen und benutztem Medium zugrunde. APM-Prozesse modellieren einzelne Mechanismen der Protokollinstanzen und AMC-Prozesse beschreiben in der Art von SCs einzelne Eigenschaften des von den Protokollinstanzen zur Kommunikation benutzten Mediums. Abstrakte Protokollmechanismen sind Abstraktionen der in praktischen Protokollen tatsächlich verwendeten Mechanismen. Sie konzentrieren sich auf die Modellierung der wesentlichen Idee der Mechanismen und nehmen insbesondere keine Rücksicht auf die effiziente direkte Implementierbarkeit. So haben die Datentypen der Zustandsvariablen und der Kontrollparameter von Protokolldateneinheiten in der Regel unendliche Wertemengen. Als typisches Beispiel sei das APM *SBufferKey* genannt, das eine senderseitige Sequenznumerierung einführt. Um die Eindeutigkeit der

übertragenen Nutzdaten zu gewährleisten, verwendet es dazu eine unendliche Menge an Laufnummern.

- Finite Abstract Protocol Mechanisms (FAPMs) und Abstract Medium Constraints (AMCs): Ein System aus FAPMs und AMCs modelliert sehr direkt das zu untersuchende Transferprotokoll. Jeder Protokollmechanismus des interessierenden Protokolls bzw. jede relevante Eigenschaft des dem Protokoll unterliegenden Mediums kann durch eine FAPM-Instanz bzw. eine AMC-Instanz dargestellt werden, so daß der Verbund dieser Instanzen eine strukturierte formale Protokollspezifikation bildet. Das FAPM *SBufferKey* benutzt im Gegensatz zu dem oben erwähnten gleichnamigen APM eine endliche Menge wiederkehrender Laufnummern zur Sequenznumerierung. Dieser Mechanismus entspricht damit direkt der Numerierung eines Sliding-Window-Protokolls.

Die Theoreme des Frameworks haben die Form logischer Implikationen zwischen cTLA-Systemen. Der Strukturierung der Spezifikationsbausteine des Frameworks in drei Schichten entspricht eine Zweigliederung der Theoreme:

- SC-Theoreme bringen zum Ausdruck, daß Systeme aus APMs und AMCs verschiedene Diensteigenschaften, d.h. SCs, implizieren. Für die meisten SCs existieren mehrere Theoreme, die die unterschiedlichen Möglichkeiten zur Erbringung der SC-Eigenschaften über abstrakte Protokollmechanismen widerspiegeln.
- APM-Theoreme überbrücken die Abstraktionsebenen der FAPM/AMC-Systeme und APM/AMC-Systeme. Ein Theorem sichert zu, daß ein System aus bestimmten FAPM- und AMC-Komponenten ein bestimmtes APM implementiert.

Diese Gliederung des Frameworks wird weiterhin durch die Unterscheidung von Liveness- und Safety-Eigenschaften verfeinert. Jede der besprochenen drei Spezifikationsbaustein-Klassen gliedert sich in zwei entsprechende Gruppen von cTLA-Prozeßtypen, die kurz Safety- bzw. Liveness-Prozesse genannt werden. Auch die Theoreme werden in Safety- und Liveness-Theoreme eingeteilt. Die Art eines Theorems entspricht der Art des Prozeßtyps der rechten Seite der Implikation. Die Theoreme haben folgenden Aufbau:

- Safety-Theorem: $Pars \wedge Sys \Rightarrow Proc$,
- Liveness-Theorem: $Pars \wedge Sys \wedge \Box EnvCond \Rightarrow Proc$.

Zentraler Teil der linken Seite der Implikation eines Theorems ist die Definition eines Systems *Sys*, das aus einer Menge gekoppelter Prozeßinstanzen besteht. Das Theorem soll zum Ausdruck bringen, daß dieses System den auf der rechten Seite genannten Prozeß *Proc* implementiert. Die Prozeßinstanzen von *Sys* werden durch Parametrisierung und Instantiierung aus Prozeßtypen gebildet. Dabei ist nicht jede beliebige Parametrisierung möglich, sondern die prädikatenlogische Formel *Pars* definiert eine hinreichende Bedingung zur konsistenten Belegung der aktuellen Prozeßparameter. Bei Liveness-Theoremen kommt zur linken Seite eine Invariante hinzu, die Umgebungsbedingung *EnvCond*. Sie definiert Anforderungen an die Umgebung von *Sys* derart, daß fairnessbehaftete Aktionen des Systems *Sys* durch die Umgebung von Sys nicht blockiert werden dürfen.

In dieser Gliederung und Form enthält das Transferprotokoll-Framework zur Zeit 133 Spezifikationsbausteine (davon 28 SCs, 44 APMs, 14 AMCs, 47 FAPMs) und 165 Theoreme (davon 31 SC-Theoreme und 134 APM-Theoreme).

4 DIENSTSPEZIFIKATION

Das Beispielprotokoll soll einen zuverlässigen und lebendigen Simplex-Datentransferdienst mit einer festen Kapazität realisieren. Insbesondere sollen alle Daten fehlerfrei übertragen werden. Dazu muß der Dienst die folgenden fünf Diensteigenschaften erfüllen: übertragene Daten werden nicht verfälscht; der Ausgabedatenstrom enthält keine durch verlorengegangene Daten entstandenen Lücken; die Reihenfolge ausgegebener Daten ist nicht vertauscht; es werden keine Duplikate bereits ausgegebener Daten an den Dienstnehmer ausgeliefert; der Ausgabedatenstrom enthält keine Phantome, d.h. der Dienst erzeugt keine zusätzlichen Daten. Eine weitere Diensteigenschaft begrenzt die Kapazität des Dienstes. Gleichzeitig dürfen nur eine bestimmte Anzahl an Dateneinheiten – in unserem Beispiel acht – gleichzeitig gesendet aber noch nicht ausgeliefert sein. Unter Lebendigkeit verstehen wir, daß der Dienst die Übertragung nicht abbricht, wenn noch nicht alle gesendeten Daten ausgeliefert wurden. Dadurch ist sichergestellt, daß jede Dateneinheit irgendwann an den empfangenden Dienstnehmer ausgegeben wird. Diese Diensteigenschaft garantiert außerdem, daß der sendende Dienstnehmer immer wieder in der Lage ist, neue Daten zu übertragen, da bei jeder Auslieferung die Kapazitätsgrenze unterschritten wird und danach mindestens eine neue Dateneinheit gesendet werden darf.

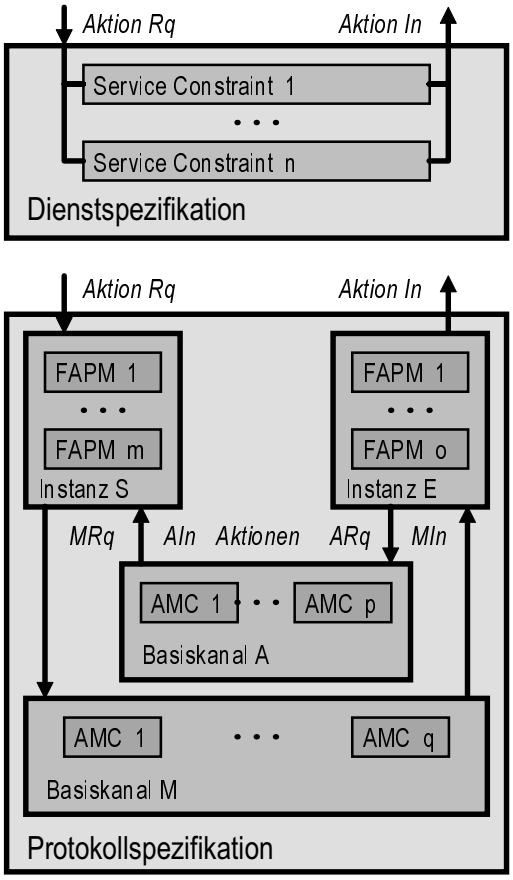

Abb. 5: Struktur von Dienst- und Protokollspezifikation

Um eine formale Spezifikation dieses Dienstes zu erstellen, erzeugen wir zunächst Constraints, die jeweils nur einer der oben

genannten Diensteigenschaften entsprechen (siehe oberen Teil in Abb. 5). Wir bilden die Constaints aus SCs des Framework, deren Parameter mit aktuellen Werten instantiiert werden. Anschließend komponieren wir die Constraints zu der Dienstspezifikation.

Die in Abb. 3 aufgeführte Spezifikation *SlidWindService* beschreibt den Dienst.*SlidWindService* enthält einen Parameter `usd`, der die Menge der Dateneinheiten angibt, die mit Hilfe des Dienstes übertragen werden können. Durch Ersetzen dieses Parameters durch die Menge *{0...255}* können wir zum Beispiel einen byteweisen Datentransferdienst spezifizieren. Die Dienstspezifikation *SlidWindService* setzt sich aus den acht Constraints *Id*, *C*, *G*, *R*, *D*, *P*, *Cap* und *LIn* zusammen, die wir zuvor aus den SCs des Frameworks *SDUId*, *Corruptions*, *Gaps*, *Reorderings*, *Duplicates*, *Phantoms*, *Capacity* bzw. *LiveInNoAttr* bilden.

Id ist ein spezielles Hilfsconstraint, das jeder gesendeten Dateneinheit im Modell eine eindeutige und geordnete Sequenznummer zuweist, so daß Fehler im Übertragungsverlauf lokalisiert werden können. Die anderen sieben Constraints modellieren die sieben oben genannten Diensteigenschaften. Zum Beispiel spezifiziert das in Abb. 1 aufgeführte Constraint *C*, das wir aus dem Prozeßtyp *Corruptions* bilden, die Eigenschaft, daß nur unverfälscht übertragene Daten ausgeliefert werden dürfen. *Corruptions* verwendet die Parameter `usd` und `tc`. Der Parameter `usd` entspricht dem oben genannten globalen Parameter in Prozeßtyp *SlidWindService*. Durch die Relation `tc` legen wir fest, in welchem Maß Verfälschungen tolerierbar sind. Wenn man `tc` zum Beispiel bei einem byteweisen Datentransfer durch die Menge $\{(k, 2\cdot(k \text{ div } 2)) \mid 0 \le k \le 255\} \cup \{(k, 2\cdot(k \text{ div } 2) + 1) \mid 0 \le k \le 255\}$ ersetzt, erlaubt man die Verfälschung des niederwertigsten Bits während der Übertragung. Da in unserem Beispiel keine Verfälschungen auftreten dürfen, ersetzen wir `tc` durch die Relation $\{(k,k) \mid k \in usd\}$. Das Constraint *G* beschreibt die Diensteigenschaft, daß keine Lücken im Ausgabedatenstrom auftreten können. Der Parameter `tg` des SCs *Gaps*, aus dem *G* erzeugt wird, zeigt die maximal tolerierbare Größe von Lücken im Datenstrom an. Da keine Lücken erlaubt sind, setzen wir `tg` auf den Wert *0*. Die Constraints *R*, *D* und *P* beschreiben die Diensteigenschaften, nach denen Vertauschungen, Duplikate und Phantome im Ausgabestrom ausgeschlossen sind. Die endliche Kapazität des Dienstes wird durch das Constraint *Cap* modelliert. Die Eigenschaft der Lebendigkeit des Dienstes wird durch das Constraint *LIn* spezifiziert (siehe Abb. 2), das aus dem Liveness-SC *LiveInNoAttr* erzeugt wird.

SlidWindService enthält die drei Aktionen `Rq`, `fIn` und `nIn`. Die Aktion `Rq` modelliert die Sendung einer Dateneinheit `d` vom Dienstnehmer. Der Dateneinheit wird die Sequenznummer `krq` zugeordnet. In dieser Aktion partizipieren die Aktionen `Rq` der Constraints *Id*, *C*, *Cap* und *LIn*. Die anderen Constraints *G*, *R*, *D* und *P* tragen nicht mit einer eigenen Aktion zu `Rq` bei. Wir verlangen, daß diese Constraints beim Schalten von `Rq` einen Stotterschritt ausführen.

Die Ausgabe von übertragenen Dateneinheiten `d` mit der Sequenznummer `krq` an den empfangenden Dienstnehmer wird durch die zwei Aktionen `fIn` und `nIn` beschrieben. Da der Dienst lebendig sein soll, müssen wir die Datenausgabe mit einer Fairness-Anforderung versehen. Wie in Kap. 2 dargestellt wurde, zerlegen wir die Aktion dazu in eine schwach Fairness-behaftete Aktion `fIn` und in eine komplementäre nicht Fairness-behaftete Aktion `nIn`. In beiden Aktionen partizipieren die Constraints *Id*, *C*, *G*, *R*, *D*, *P* und *Cap* jeweils mit ihren Aktionen `In`. Aus dem Constraint *LIn* ist die Constraint-Aktion `fIn` an `fIn` und die Constraint-Aktion `nIn` an `nIn` gekoppelt. `fIn` beschreibt Auslieferungen von Daten, die für den Fortschritt der Kommunikation des Dienstes zwingend erforderlich sind. Dazu gehört zum Beispiel die Ausgabe einer Dateneinheit, die zwar gesendet aber bisher noch nicht ausgeliefert wurde. Durch `nIn` werden dagegen Auslieferungen beschrieben, die für den Fortschritt nicht zwingend sind, zum Beispiel Auslieferungen bereits zuvor ausgelieferter Dateneinheiten. Da in unserem Beispiel keine überflüssigen Dateneinheiten ausgeliefert werden, ist `nIn` niemals schaltbar. In einer vereinfachten Beschreibung kann diese Aktion somit weggelassen werden.

Die Aktion `fIn` ist schwach fair, da auch die in ihr enthaltene Aktion `fIn` von Constraint *LIn* schwach fair ist (siehe Abb. 2). Wenn sie ständig schaltbar ist, wird sie deshalb irgendwann aus der Menge der gerade schaltbaren Aktionen zum Schalten ausgewählt.

5 PROTOKOLLSPEZIFIKATIONEN

Das Sliding-Window-Protokoll [29] soll den oben vorgestellten Datentransferdienst implementieren. Die Struktur der Protokollspezifikation wird im unteren Teil von Abb. 5 dargestellt. Es besteht aus je einer Protokollinstanz S auf der Seite des Senders und R auf der Seite des Empfängers, die miteinander unter Verwendung eines Voll-Duplex-Basisdienstes kommunizieren. Die Zuverlässigkeit des Basisdienstes ist geringer als diejenige des in Kap. 4 vorgestellten Zieldienstes. Er wird durch zwei Simplex-Datenkanäle *M* und *A* beschrieben. Wir gehen davon aus, daß bei der Übertragung über den Basisdienst Datenverluste und Duplikate auftreten können, jedoch alle anderen Fehlerarten ausgeschlossen sind. Der Basisdienst besitzt somit die folgenden Diensteigenschaften: PDUs werden nicht verfälscht; PDUs werden in korrekter Reihenfolge ausgegeben; der Ausgabedatenstrom enthält nur zuvor gesendete Nachrichten (Phantomfreiheit). Eine derartige Übertragungsgüte liegt zum Beispiel vor, wenn die unterliegende Punkt-zu-Punkt-Datenübertragung zwar gegen Übertragungsfehler gesichert ist, jedoch Datenverluste aufgrund von Speicherüberläufen und Duplikate aufgrund von Kopierfehlern in den Vermittlungsknoten auftreten können. Ferner verlangen wir vom Basisdienst die folgende Lebendigkeitseigenschaft: Wenn der Basisdienstnehmer immer wieder PDUs mit einem bestimmten Attribut sendet, muß irgendwann eine dieser PDUs an seinen Kommunikationspartner ausgeliefert werden. Ohne diese Lebendigkeitsanforderung könnte eine Dateneinheit immer wieder verlorengehen, was aufgrund der vom Protokoll garantierten Reihenfolgetreue zu einem Stillstand der Kommunikation führte.

Die Aufgabe der Protokollinstanzen des Sliding-Window-Protokolls besteht darin, bei der Übertragung über die Basiskanäle auftretende Datenverluste und Duplikate zu erkennen und zu beheben. Außerdem müssen sie die Anforderungen an die Kapazität und die Lebendigkeit des Zieldienstes garantieren. Das Protokoll verwendet dazu einen Satz an Protokollfunktionen, wie sie bei den meisten modernen Datentransferprotokollen üblich sind (vgl. [7, 27]). Datenverluste und Duplikate werden mit Hilfe von Sequenznummern erkannt, die in der im folgenden als S

bezeichneten Protokollinstanz auf der Seite des Senders jeder vom Zieldienstnehmer übergebenen Dateneinheit zugewiesen werden. Duplikate bereits empfangener Daten werden von der Protokollinstanz R auf der Empfängerseite ignoriert. Datenverluste werden durch selektive Übertragungswiederholung behoben, bei der alle verlorengegangenen Dateneinheiten erneut übertragen werden. Um der Sendeinstanz S des Sliding-Window-Protokolls zu ermöglichen, alle Dateneinheiten zu erkennen, die noch einmal über den Basisdienst übertragen werden müssen, sendet die Empfangsinstanz R die Sequenznummer der zuletzt dem Dienstnehmer ausgelieferten Dateneinheit in einer Bestätigungsmitteilung an S. Die Instanz S wiederholt bei diesem „Positive Acknowledgement with Retransmission (PAR)“ genannten Verfahren nur Dateneinheiten, deren Auslieferung von R noch nicht bestätigt wurde. S hält die Kapazitätsgrenze des Zieldienstes ein, indem sie nur Sendungen vom Zieldienstnehmer akzeptiert, wenn weniger als acht gesendete Dateneinheiten unbestätigt sind. Die Lebendigkeit des Zieldienstes wird dadurch garantiert, daß S zum einen sicherstellt, daß neue oder zu wiederholende Dateneinheiten tatsächlich über den Basisdienst übertragen werden. Zum anderen garantiert R die Auslieferung aller korrekt empfangenen Dateneinheiten an den Dienstnehmer und die Sendung von Bestätigungen an S über den Basisdienst.

Die Protokollspezifikation *SlidWindProtocol* (siehe Abb. 6) bilden wir, indem wir Prozesse aus dem Framework instantiieren und miteinander komponieren. Zugunsten der Übersichtlichkeit verzichten wir bei der Angabe der Prozesse im Teil PROCESSES weitgehend auf die Angabe der Prozeßparameter. Im Teil ACTIONS werden die Systemaktionen meist ohne Aktionsparameter und lokale Prozeßaktionen aufgelistet.

Wie die Dienstbeschreibung enthält *SlidWindProtocol* den Parameter usd, mit dem man die Menge der übertragbaren Nutzdaten angibt. Zur besseren Übersicht führen wir Bezeichner für die Parameterersetzungen ein, die wir in der weiter unten erläuterten Datei *SWParameters* definieren. Im durch PROCESSES gekennzeichneten Teil der Spezifikation (siehe Abb. 6) werden die Framework-Prozesse aufgelistet, mit denen wir die Protokollinstanzen und die Basiskanäle modellieren. Die Protokollinstanz S auf der Seite des Senders spezifizieren wir durch die FAPMs *SBK*, *SBU*, *SAck*, *SCap* und *SLMRq*. Diese FAPMs modellieren die Mechanismen zur Verwaltung der Sequenznummern, der Nutzdaten, der Bestätigungen korrekt übertragener Daten und der Datenspeichergröße sowie den Protokollmechanismus, der die lebendige Übertragung noch unbestätigter Dateneinheiten sicherstellt. Wir beschreiben die Protokollinstanz R auf der Seite des Empfängers durch die FAPMs *RBK*, *RBU*, *RG*, *RR*, *RD*, *RP*, *RAck*, *RLARq* und *RLIn*. Dadurch werden die Protokollmechanismen zur Verwaltung von Sequenznummern und Nutzdaten, zur reihenfolgetreuen Auslieferung der Nutzdaten und der Bestätigung empfangener Nutzdaten modelliert. Die Lebendigkeit der Instanz stellen wir durch die FAPMs *RLARq* und *RLIn* sicher, die die Bestätigung bzw. Auslieferung aller empfangener Dateneinheiten garantieren.

Den Datenkanal M, der die Übertragung von Zieldienstdaten und Protokollsteuerinformation (Protocol Control Information, PCI) im Rahmen von PDUs von S nach R beschreibt, spezifizieren wir mit Hilfe der Prozesse *MId*, *MC*, *MR*, *MP* und *MLI*. *MId* ist wie das SC *Id* ein spezielles Hilfsconstraint, das übertragenen PDUs im Modell Identifikationsnummern zuordnet. Durch *MC*, *MR* und *MP* modellieren wir, daß übertragene Daten nicht verfälscht oder vertauscht werden, und daß keine Phantome auftreten. *MLI* garantiert, daß eine immer wieder durch Instanz S in einer PDU gesendete Dienstdateneinheit irgendwann an die Instanz R ausgeliefert wird. Der Datenkanal A, der die Übertragung von PCI in der Richtung von R nach S modelliert, wird durch die Prozesse *AId*, *AC*, *AR*, *AP* und *ALI* spezifiziert. Mit der Ausnahme, daß *ALI* die Auslieferung aller Bestätigungsnachrichten an S garantiert, wenn sie von der R immer wieder gesendet werden, entsprechen die Prozesse denen von Kanal M.

```
PROCESS SlidWindProtocol (usd : any)
                        Specification of the Sliding-Window-Protocol
 IMPORT SWParameters(usd);
 PROCESSES        Specification SlidWindProtocol: System components
              ------- FAPMs  modelling transmitter entity S
   SBK : SBufferKey                Sequence number handler
          (swpdu, swpci, usd, swpdu, swspci,
           swskey, 1, swskk, swskn, swskm,
           swusdsize, 1, 16, 8);
   SBU : SBufferUsd(...);            User data buffer handler
   SAck : SAcknowledge(...);    Data acknowledge mechanism
   SCap : SCapacity(...);         Preventing data unit overflow
   SLMRq : SLiveMRq(...);              Transmission liveness
              ------- FAPMs modelling receiver entity R
   RBK : RBufferKey(...);           Sequence number handler
   RBU : RBufferUsd(...);            User data buffer handler
   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
   RAck : RAcknowledge(...);    Data acknowledge mechanism
   RLARq : RLiveARq(...);              Live acknowledgement
   RLIn : RLiveIn(...);                          Live delivery
              ------- AMCs : Constraints of the basic service channel M
   MS : MSDUId ;       Ordered assignment of sequence numbers
   MC : MCorruptions(...);       No corruptions during transfer
   MR : MReorderings(...);       No reorderings during transfer
   MP : MPhantoms(...);         No phantoms generated by  M
   MLI : MLiveIn(...);        Eventual transfer of repeated PDUs
              ------- AMCs : Constraints of the basic service channel A
   AS : ASDUId;         Ordered assignment of sequence numbers
   AC : ACorruptions(...);       No corruptions during transfer
   AR : AReorderings(...);       No reorderings during transfer
   AP : APhantoms(...);          No phantoms generated by A
   ALI : ALiveIn(...);        Eventual delivery of repeated PDUs
 ACTIONS             Specification SlidWindProtocol: System actions
  Rq(krq:fkey;d:usd)≡     Submission of user data d, seq.no. krq
    SBK.Rq(krq,d) ∧ SBU.Rq(krq,d) ∧
    SAck.Rq(krq) ∧ SCap.Rq(krq) ∧
    SLMRq.Rq (krq, d) ∧ ...;
  fIn(krq:fkey;d:usd)≡ ...; Delivery: user data d, seq.no. krq
  nIn (krq:fkey;d:usd) ≡ ...;          fair delivery / other
  fMRq ( ... ) ≡ ...;                  Submission of a pdu to M
  nMRq ( ... ) ≡ ...;                      fair submission / other
  fMIn ( ... ) ≡ ...;        Delivery of a data pdu from M to receiver
  nMIn ( ... ) ≡ ...;                                fair / other
  fARq ( ... ) ≡ ...;               Submission of an ack-pdu to A
  nARq ( ... ) ≡ ...;                                fair / other
  fAIn ( ... ) ≡ ...;        Delivery of an ack-pdu from A to sender
  nAIn ( ... ) ≡ ...;                                fair / other
  MTick ≡ MLI.MTick ∧ ...;          Internal actions resulting in
  ATick ≡ ALI.ATick ∧ ...;          in loss of a pdu
  fMNoTick                  Internal actions for the fair delivery of
   (p:[info:usd;seq:key∪{⊥};ack:key]) ≡ ...;
  fANoTick                   repeated pdus
    (p:[info:usd;seq:key∪{⊥};ack:key]) ≡ ...;
 END .
```

Abb. 6: Protokoll-Spezifikation *SlidWindProtocol*

Der in Abb. 6 aufgeführte Teil ACTIONS der Spezifikation *SlidWindProtocol* modelliert die Kopplung der einzelnen Prozeßaktionen zu Aktionen des Gesamtsystems. Wir beschreiben die Aktionen In, MRq, MIn, ARq und AIn jeweils durch eine faire Aktion und eine nicht-faire Aktion. So spezifiziert zum Bei-

spiel die Aktion fMRq das Senden von Dateneinheiten, die für den Fortschritt des Ablaufs zwingend notwendig sind, während nMRq Sendungen modelliert, die im aktuellen Zustand erlaubt und aus Effizienzgründen auch wünschenswert sind, die Lebendigkeit des Protokolls jedoch (noch) nicht tangieren. Die räumliche Verteilung der Protokollinstanzen stellen wir dadurch sicher, daß in jeder Systemaktion entweder alle Prozesse von Instanz S oder von Instanz R nur mit Stotterschritten partizipieren.

Die Prozesse des Frameworks passen wir durch die Parameterbelegungen an die spezifischen Eigenschaften des Sliding-Window-Protokolls an. Abb. 7 skizziert dazu die Spezifikation *SWParameters*, die Symbole zur Belegung der Parameter definiert.

Der Bezeichner swpdu modelliert das PDU-Format des Sliding-Window-Protokolls. Es besteht nur aus den drei Felder info, seq und ack. Die mit der PDU übertragenen Nutzdaten werden im Feld info gespeichert. Sie können alle Werte der Menge aller Dateneinheiten usd annehmen. Das Feld seq enthält entweder die Sequenznummer der in der PDU übertragenen Nutzdateneinheit vom Datentyp key (eine natürliche Zahl oder die Kennung "notsent" für Phantome) oder das Sonderzeichen ⊥. Mit ⊥ kennzeichnen wir PDUs, die nur Bestätigungsinformation für den Kommunikationspartner aber keine eigenen Dateneinheiten enthalten. Die Sequenznummer der zuletzt ausgelieferten Dateneinheit wird im Feld ack gespeichert.

Die Sequenznummern der gesendeten und der bestätigten Daten in der PDU bilden ihre Protokollsteuerinformation (PCI), so daß der Bezeichner swpci einen Verbundtyp mit den Feldern seq und ack enthält. swspci beschreibt die Referenz auf die PCI einer PDU. Durch die Funktion wird eine PDU auf eine PCI abgebildet, deren Felder seq und ack dieselben Werte wie die gleichnamigen Felder in der PDU besitzen.

```
CONSTANT MODULE SWParameters (usd : any)
 Parameters of the Specification of the Sliding-Window-Protocol
 CONSTANTS
  ------ Used in FAPMs and AMCs
  swpdu ≡ [info:usd;seq:fkey∪{⊥};ack:fkey];
            Abstract pdu format
  swpci ≡ [seq:fkey∪{⊥};ack:fkey];
            Protocol control information
  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 ≡       Relation: identity of pdus
      {(k,k)|k∈[info:usd;seq:key∪{⊥};ack:key]};
  …;
 END .
```

Abb. 7: Parameterbelegungen *SWParameters*

Wie in Kap. 3 beschrieben wurde, spezifizieren wir das Sliding-Window-Protokoll in zwei Schritten. Im ersten Schritt erzeugen wir die eigentliche Protokollspezifikation. Im zweiten Schritt bilden wir eine abstraktere Protokollspezifikation, die zwar auch die räumliche Verteilung der Instanzen modelliert, jedoch Variablen mit unendlichen Wertebereichen enthält. In dieser Spezifikation abstrahieren wir somit von Protokollfehlern, die durch die im realen Betrieb notwendige Mehrfachnutzung von Sequenznummern und Verbindungskennungen entstehen können. Durch diese Unterteilung können wir die Verifikation des Sliding-Window-Protokolls in zwei einfachere Schritte zerlegen (siehe Kap. 6).

Die Instanzen der abstrakten Protokollspezifikation modellieren wir wieder durch die Framework-Prozesse *SBK*, *RBK*, *SBU*, *RBU*, *RG*, *RR*, *RD*, *SAck*, *RAck*, *SCap*, *SLMRq*, *RLARq* und *RLIn*. Allerdings verwenden wir dazu Abstract Protocol Mechanisms (APMs) des Frameworks, die im Gegensatz zu den FAPMs Variablen mit unendlichem Wertebereich enthalten. Zur Modellierung der Basiskanäle M und A verwenden wir in der abstrakten Spezifikation die gleichen AMCs wie in der Spezikation *SlidWindProtocol*.

6 VERIFIKATION

Durch die Verifikation stellen wir sicher, daß der in Kap. 4 spezifizierte Kommunikationsdienst durch das in Kap. 5 modellierte Sliding-Window-Protokoll erbracht wird. Aufgrund der Kompositionalität von cTLA können wir sie in mehrere einfachere Beweisschritte zerlegen. In jedem dieser Beweisschritte weisen wir nach, daß eine Diensteigenschaft durch ein aus einigen Protokollmechanismen bestehendes Subsystem des Sliding-Window-Protokolls realisiert wird. Wir verwenden für jeden dieser Beweisschritte ein Theorem des Frameworks und müssen dabei lediglich überprüfen, ob das Protokollsystem aus einer bestimmten Kombination von Protokollmechanismen besteht, deren aktuelle Parameter zueinander und zu denen der realisierten Diensteigenschaft konsistent sind. Die Verifikation ist somit auf die Auswahl und Konsistenzüberprüfung von Theoremen des Frameworks zurückgeführt.

```
LET
 Pars ≡          ------ Parameter condition
  {(p,q)|p,q∈[info:usd;seq:key∪{⊥};ack:key] ∧
          p.seq=q.seq} =
  {(p,q)|p.seq=q.seq ∧
          p,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 ≡          ------ Subsystem definition
       SLiveMRq
        ([info:usd;seq:key∪{⊥};ack:key], …) ∧
       RLiveARq (…) ∧ RLiveIn (…) ∧
       RAcknowledge (…) ∧ MSDUId ∧
       MCorruptions (…) ∧ MPhantoms (…) ∧
       MLiveIn (…) ∧ ASDUId ∧
       ACorruptions (…) ∧ APhantoms (…) ∧
       ALiveIn (…) ∧ CCLiveInNoAttr;
 EnvCond ≡    ------ Environment condition
  ∀ krq,p,kd:
    Enabled(SLiveMRq.fMRq(krq,p,kd)) ⇒
    (krq,p,kd)∈Sys.efMRq ∧
  ∀ p,kd:
    Enabled(RLiveARq.fARq(p,kd)) ⇒
    (p,kd)∈Sys.efARq ∧
    …
  ∀ d:
    Enabled(ALiveIn.fANoTick(d)) ⇒
    d∈Sys.efANoTick ;
IN          ------ The theorem
    Pars ∧ Sys ∧ □EnvCond ⇒ LiveInNoAttr
```

Abb. 8: Theorem *LiveInNoAttr*

Der Protokollbeweis wird in zwei Schritte zerlegt. Im ersten Schritt verifizieren wir, daß der Kommunikationsdienst durch das mit APMs und AMCs spezifizierte abstrakte Protokoll erbracht wird. Im zweiten Schritt weisen wir nach, daß die abstrakte Pro-

tokollspezifikation durch die eigentliche Sliding-Window-Protokollspezifikation erfüllt wird, die aus FAPMs und AMCs gebildet ist. Für den ersten Schritt arrangieren wir acht Theoreme des Frameworks, in denen jeweils ein SC der Dienstspezifikation verifiziert wird. Als Beispiel zeigen wir die Verifikation des Liveness-SCs *LIn*. In Abb. 8 geben wir das dazu verwendete Theorem des Frameworks an. (Um den Umfang der Spezifikation zu begrenzen, haben wir die aktuellen Parameter der Prozesse des Protokollsubsystems weggelassen.)

Die Aussage dieses Theorems ist, daß eine Instanz des SCs *LiveInNoAttr*, in unserem Fall *LIn*, von einem Protokollsystem realisiert wird, das das System *Sys* als Subsystem hat, wenn die Bedingungen *Pars* und ❑*EnvCond* gelten. *Sys* besteht aus Instanzen der APMs *SLiveMRq*, *RLiveARq*, *RLiveIn* und *RAcknowledge* sowie aus Instanzen der AMCs *MSDUId*, *MCorruptions*, *MPhantoms*, *MLiveIn*, *ASDUId*, *ACorruptions*, *APhantoms* und *ALiveIn*. Dieses Subsystem garantiert, daß Daten (*SLiveMRq*) und Bestätigungen (*RLiveARq*) genügend häufig zwischen den Protokollinstanzen übermittelt werden, korrekt empfangene Daten an den Dienstnehmer ausgeliefert werden (*RLiveIn*) und nicht korrekt empfangene Daten unbestätigt bleiben (*RAcknowledge*). Außerdem sind die Basiskanäle lebendig (*MLiveIn*, *ALiveIn*) und liefern keine verfälschten oder Phantom-Sequenznummern aus (*MSDUId*, *MCorruptions*, *MPhantoms*, *ASDUId*, *ACorruptions*, *APhantoms*).

Durch Ersetzung der formalen Parameter aller in *Sys* enthaltenen Komponenten auf dieselbe Weise wie bei der Spezifikationserstellung (siehe Kap. 5) passen wir das Theorem an unser Beispiel an. Da die abstrakte Protokollspezifikation Instanzen aller in *Sys* angegebenen APMs und AMCs besitzt, deren Aktionen miteinander gemäß der hier nicht näher aufgeführten Formel *CCLiveInNoAttr* gekoppelt sind, ist *Sys* somit ein Subsystem des abstrakten Sliding-Window-Protokolls. Die Parameterersetzung ist konsistent, wenn die Formel *Pars* wahr ist. Das erste Konjunkt von *Pars* ist eine Tautologie und deshalb trivialerweise wahr. Die beiden weiteren Konjunkte sagen aus, daß zwei identische Protokolldateneinheiten *p* und *q* die Eigenschaft besitzen, daß ihre Komponenten *p.seq* und *q.seq* bzw. *p.ack* und *q.ack* gleich sind. Diese Eigenschaften gelten offensichtlich, da sich zwei identische PDUs gerade dadurch auszeichnen, daß alle Komponenten (insbesondere *seq* und *ack*) identische Werte besitzen.

Die temporale Bedingung ❑*EnvCond* wird nur bei der Verifikation von Liveness-SCs verwendet. Sie stellt sicher, daß *LiveInNoAttr* nicht nur durch *Sys*, sondern auch durch das gesamte abstrakte Sliding-Window-Protokoll erbracht wird. Sie muß übrigens nicht ausdrücklich bewiesen werden, da bei der Entwicklung jedes Liveness-Theorems untersucht wurde, welche Prozeßtypen zur Verletzung der Bedingung beitragen können. So wurde bei der Erstellung dieses Theorems nachgewiesen, daß mit Ausnahme des APMs *DataChanOpenR* kein APM oder AMC des Frameworks die Bedingung ❑*EnvCond* verletzt. Da *DataChanOpenR* nicht in der abstrakten Protokollspezifikation enthalten ist, gilt somit ❑*EnvCond*. Da *Pars* und ❑*EnvCond* gelten und *Sys* ein Subsystem der abstrakten Protokollspezifikation ist, haben wir verifiziert, daß das abstrakte Sliding-Window-Protokoll die Diensteigenschaft *LIn* realisiert. Auf die gleiche Weise beweisen wir die sieben anderen SCs der Diensteigenschaft.

Den Beweis, daß die konkrete Sliding-Window-Protokollspezifikation die abstrakte erfüllt, führen wir analog. Mit Hilfe von 13 Theoremen des Frameworks verifizieren wir die Gültigkeit der 13 APMs. Die Korrektheit der AMCs muß nicht explizit bewiesen werden, da das konkrete und das abstrakte Protokoll denselben Basisdienst verwenden, der durch identische AMCs gebildet wird.

7 SCHLUSSBEMERKUNGEN

Die Grundzüge des Transferprotokoll-Frameworks und seiner Anwendung zur Modellierung, formalen Spezifikation und Verifikation von Kommunikationsprotokollen wurden anhand der exemplarischen Behandlung eines Sliding-Window-Protokolls erläutert. In ähnlicher Weise konnten auch schon umfangreichere Protokolle (wie z.B. XTP; siehe [18]) mit bemerkenswert geringem Arbeitsaufwand untersucht werden und wir möchten interessierten Lesern die Nutzung des Frameworks nahelegen. Es ist über WWW zugänglich (http://ls4-www.informatik.uni-dortmund.de/RVS/P-TPM). Außerdem wird es in [12] ausführlich beschrieben.

Zur Zeit arbeiten wir an Erweiterungen der Spezifikationstechnik cTLA, die zusätzlich zur Modellierung ereignisdiskreter, nichtzeitbewerteter Dynamik auch die Erfassung von Realzeit-Eigenschaften und zeitlich-kontinuierlichen Vorgängen leisten [13, 17, 19]. Wir möchten auf dieser Basis den Framework-Ansatz auf verteilte Realzeit-Systeme übertragen und untersuchen dazu momentan das Anwendungsfeld der Kontrolle chemietechnischer Anlagen. Darüberhinaus sind diese cTLA-Erweiterungen natürlich auch wieder für Kommunikationsprotokolle interessant, wenn die Übertragung von Multimedia-Daten modelliert werden soll.

Literatur

[1] B. Alpern u. F. Schneider. Defining Liveness. Inf. Proc. Let., 21:181–185, 1985.

[2] M. Broy et al. The Design of Distributed Systems – An Introduction to FOCUS. Interner Bericht TUM-I9202, Technische Universität München, Inst. für Informatik,1992.

[3] R. Back u. R. Kurkio-Suonio. Decentralization of process nets with a centralized control. Distributed Computing, (3):73–87, 1989.

[4] S. Budkowski. Estelle Development Toolset. Computer Networks and ISDN Systems, 25(1):63–82, 1992.

[5] E. Clarke, E. Emerson u. A. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. ACM Trans. on Prog. Lang. a. Syst., 8(2):244–263, 1986.

[6] E. Clarke, O. Grumberg u. D. Long. Model Checking and Abstraction. ACM Trans. on Prog. Lang. a. Syst., 16(5):1512–1542, 1994.

[7] W. Doeringer et al. A Survey of Light-Weight Transport Protocols for High-Speed Networks. IEEE Trans. on Comm., 38(11):2025–2039, 1990.

[8] U. Engberg, P. Gronning u. L. Lamport. Mechanical Verification of Concurrent Systems with TLA. In v. Bochmann, G. et al, Hrsg., Verification, LNCS 663, S. 44–55, Springer-Verlag, 1992.

[9] J. Fernandez u. L. Mounier. On the Fly Verification of Behavioural Equivalences and Preorders. LNCS 575, S. 181–191, Berlin Heidelberg, Springer-Verlag, 1991.

[10] S. Garland, J. Guttag u. J. Horning. Debugging Larch Shared Language Specifications. IEEE Trans. on Soft. Eng., 16(9):1044–1057, 1990.

[11] R. Gerth et al. Simple On-the-fly Automatic Verification of Linear Temporal Logic. In P. Dembinski u. M. Sredniawa, Hrsg., Protocol Specification, Testing, and Verification XV, S. 3–18, IFIP, Chapman & Hall, 1995.

[12] P. Herrmann. Problemnaher korrektheitssichernder Entwurf von Hochleistungsprotokollen. Dissertation, Universität Dortmund, 1997. Erscheint im Deutschen Universitätsverlag, 1998.

[13] P. Herrmann, G. Graw u. H. Krumm. Compositional Specification and Structured Verification of Hybrid Systems in cTLA. Erscheint in Proc. of 1st IEEE Int. Symp. on Object-oriented Real-time distributed Computing (ISORC98), Kyoto, Japan, 1998.

[14] Z. Har'El u. R. Kurshan. Software for analytical development of communications protocols. AT&T Technical Journal, 69(1):45–59, 1990.

[15] P. Herrmann u. H. Krumm. Compositional Specification and Verification of High-Speed Transfer Protocols. In S. Vuong u. S. Chanson, Hrsg., Protocol Specification, Testing, and Verification XIV, S. 339–346, IFIP, Chapman & Hall, 1994.

[16] P. Herrmann u. H. Krumm. Re-Usable Verification Elements for High-Speed Transfer Protocol Configurations. In P. Dembinski u. M. Sredniawa, Hrsg., Protocol Specification, Testing, and Verification XV, S. 171–186, IFIP, Chapman & Hall, 1995.

[17] P. Herrmann u. H. Krumm. Kompositionale Constraints hybrider Systeme. In E. Schnieder u. D. Abel, Hrsg., Entwurf komplexer Automatisierungssysteme, S. 243–264, Braunschweig, 1997.

[18] P. Herrmann u. H. Krumm. Modular Specification and Verification of XTP. In Proc. of the 5th Int. Conf. on Telecommunication Systems – Modelling and Analysis, S. 477–486, IFIP, 1997.

[19] P. Herrmann u. H. Krumm. Specification of Hybrid Systems in cTLA+. In Proc. of the 5th Int. Workshop on Parallel & Distributed Real-Time Systems – (WPDRTS'97), S. 212–216, IEEE, 1997.

[20] P. Herrmann et al. Automated Verification of Refinements of Concurrent and Distributed Systems. Research Report 541, Universität Dortmund, Informatik IV, 1994.

[21] G. Holzmann. Algorithms for Automated Protocol Verification. AT&T Technical Journal, S. 32–44, 1990.

[22] G. Holzmann. Design and validation of protocols: a tutorial. Computer Networks and ISDN Systems, 25:981–1017, 1993.

[23] ESTELLE: A formal description technique based on an extended state transition model, Int. Standard ISO/IS 9074, 1989.

[24] LOTOS: Language for the temporal ordering specification of observational behaviour, Int. Standard ISO/IS 8807, 1989.

[25] Rec. Z.100: CCITT/ITUSpecification and Description Language SDL, 1993.

[26] L. Lamport. The Temporal Logic of Actions. ACM Trans. on Prog. Lang. a. Syst., 16(3):872–923, 1994.

[27] T. LaPorta u. M. Schwartz. Architectures, Features, and Implementation of High-Speed Transport Protocols. IEEE Network Magazine, S. 14–22, May 1991.

[28] A. Mester u. H. Krumm. Composition and Refinement Mapping based Construction of Distributed Applications. In: U. Engberg et al., Hrsg., Proc. of the TAPSOFT'95 satellite workshop on Tools and Algorithms for the Construction and Analysis of Systems, Aarhus, Denmark, May 1995. BRICS (Basic Research in Computer Science) Notes Series (ISSN 0909-3206) NS-95-2, S. 290-303.

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

[30] C. Vissers, G. Scollo u. M. v. Sinderen. Architecture and specification style in formal descriptions of distributed systems. In S. Agarwal u. K. Sabnani, Hrsg., Protocol Specification, Testing and Verification VIII, S. 189–204, IFIP, Elsevier, 1988.

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