Development and Implementation of a Communication Protocol — An Exercise in FOCUS

Claus Dendorfer, Rainer Weber
Institut für Informatik der Technischen Universität München
Postfach 20 24 20, D-8000 München 2, Germany

March 1992

Abstract

The use of the formal design method FOCUS is illustrated by an example development of the so-called Stenning-protocol. The development process starts from global, non-constructive service specifications and ends in executable programs of the protocol entities. The four abstraction levels of FOCUS — trace specification, functional specification, abstract program, and concrete program — are covered. Special emphasis is put on the design steps within and between the first two abstraction levels, which are also proven correct.

1. Introduction

The methodological aspects of formal protocol development (instead of just verification) currently receive increased attention [Probert, Saleh 91]. We use the general design method FOCUS to derive implementations of protocol entities (PEs) from initial service specifications by the technique of stepwise refinement. Each development step is carried out within a formal framework. The example given in this paper is the *Stenning-protocol* [Stenning 76]. We think that it is sufficient to show the principle of FOCUS. A comprehensive description of FOCUS is available in [Broy et al. 92].

The development starts from an abstract description of the service to be provided by the protocol entities (upper layer service) and of the service provided by the medium (lower layer service). Here the formalism of *trace logic* is used to impose requirements on the system runs. In several design steps, trace specifications of the PEs are derived. These are then transformed into predicates on *stream processing functions*, which model asynchronously communicating agents. Several further design decisions are incorporated into this specification until a *functional*

program is achieved. Program transformation rules are applied to derive the final *imperative program*.

The reason for combining several formal description techniques in FOCUS is that we want to cover a broad range of abstraction levels from non-constructive global descriptions to executable programs. For example, initially it is appropriate to have a trace-oriented global view of a system instead of thinking about it in terms of its processes. For the subsequent development process, the component-oriented functional setting provides a more modular description technique. Of course, the formalisms have to be well-integrated, and design principles are needed both for the refinement within a particular abstraction level and for the transition between successive levels.

The chosen example is inspired by [Li, Maibaum 88], in which also the stepwise development of protocol descriptions is also emphasized. However, Li and Maibaum did their development within a single abstraction level, which corresponds to our trace specification, and they did not end up in programs for the PEs.

Most published approaches to service-oriented protocol synthesis either are based on finite state machines or on LOTOS [Gotzhein, Bochmann 90], [Chu, Liu 88], [Saleh, Probert 90]; see [Probert, Saleh 91] for an overview. In contrast to these methods, we do not start with already executable specifications and use more than one formalism. Program development in our method is guided by the designer, who may make arbitrary design decisions, while in many existing methods the protocol specification is generated automatically. The reason why we favour interactive protocol design is that a service specification usually permits several different protocol specifications and implementations (in our example there are at least three of them). We do not think that the creative task of protocol development can be automated.

Related work on the stepwise refinement of distributed systems includes Unity [Chandy, Misra 88] and approaches in process algebras (e.g. [Olderog 90]). Unity is a state-oriented formalism, while our method is action-oriented. Process algebras use synchronous communication, whereas FOCUS has asynchronous communication and is well-suited for loosely-coupled systems. Furthermore, arbitrary liveness conditions may be expressed. According to [Bochmann 90], this is not the case for most protocol specification languages, including the standardized formal description techniques.

The paper is organized as follows: in section 2 we give an overview of our method. Section 3 to 6 comprise the design steps applied to the example. Section 7 presents conclusions.

2. Overview of FOCUS

FOCUS covers the process of system development from a non-constructive global specification of a system up to executable programs. See Fig. 1 for an overview.

A *trace specification* states requirements for the traces (runs, histories) of a distributed system. This specification style is comparable to (linear time) temporal logic [Pnueli 86]. However, in a trace specification sequences of *actions* are specified instead of sequences of *states*. This view is similar to that of time sequence charts.

Initially there is just a global specification, which only states implicit requirements for the PEs. The goal of the first design steps is therefore to derive explicit local specifications. This is

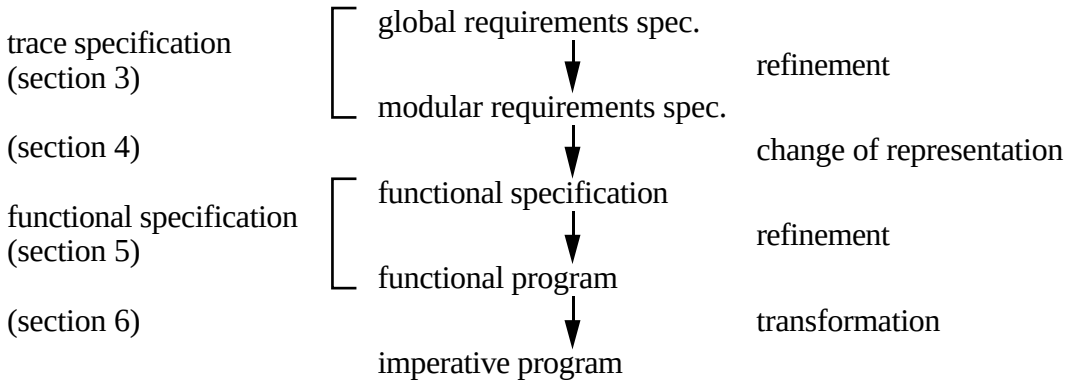

Fig. 1

called the *localisation of requirements*. Then we switch to *functional specifications* of the PEs, where stream-processing functions are used to model each PE. The initial functional specification is still non-constructive, i.e. it describes only the allowed system behaviours and not how these may be achieved. In a step-by-step process, further design decisions are made, and the components' reactions to an input are described in a more constructive way. An explicit state is introduced, which is updated when a new input arrives, and which determines a component's future behaviour. This specification is rewritten as a functional program and (for efficiency reasons) finally transformed into an imperative program.

3. Development on the trace level

The underlying data structures will be defined in section 3.1. Then, starting from the descriptions of the services of the upper and the lower layer in section 3.2, we develop trace specifications of the PEs in section 3.3 to 3.5. The specifications in section 3.5 are the basis for the development on the functional level.

3.1 The underlying data structures

Fig. 2 depicts a situation typical for protocol specification: two protocol entities communicate via a medium to provide a particular service. The service provided by the two protocol entities will be referred to as the *service of the upper layer*, the service guaranteed by the underlying medium as the *service of the lower layer*.

Before specifying the actions that my occur in our system, we will have a look at the data to be communicated. The parameters of the service primitives in our example are two kinds of protocol data units (PDUs). The PDUs relevant for the upper layer are from a set UDU ("upper layer data units"), and the PDUs relevant for the lower layer are from a set LDU ("lower layer data units"). It is not specified here how these sets exactly look like. However, it is assumed that on the way from the transmitter to the receiver only PDUs from a set $Info \subseteq LDU$ appear, and that on the way back only acknowledgements from a set $Ack \subseteq LDU$ appear.

In the trace formalism, atomic *actions* are used to model the service primitives. For every $d \in UDU$, we have the actions snd(d), which denotes the sending of a UDU d from the

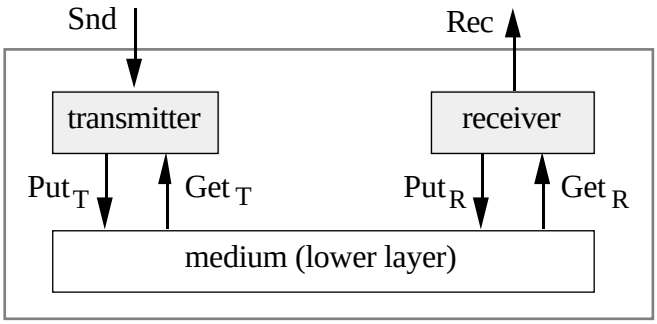

Fig. 2

service user to the transmitter, and rec(d), which denotes that d is delivered by the receiver to the service user. Note that snd(d) is just the name of an action, and not the application of some function snd. For $e \in$ LDU, $put_T(e)$ denotes the action of sending e from the transmitter to the medium, and $get_T(e)$ denotes the delivery of e from the medium to the transmitter. These two actions correspond to $put_R(e)$ and $get_R(e)$ at the receiver's side. We define the following sets of actions (see Fig. 2):

$$Snd \equiv \{snd(d) \mid d \in UDU\} \qquad Rec \equiv \{rec(d) \mid d \in UDU\}$$

$$Put_T \equiv \{put_T(x) \mid x \in Info\} \qquad Put_R \equiv \{put_R(y) \mid y \in Ack\}$$

$$Get_T \equiv \{get_T(y) \mid y \in Ack\} \qquad Get_R \equiv \{get_R(x) \mid x \in Info\}$$

Finally we have an action $\surd$ (called "tick"), which denotes the ticking of a clock. We shall come back to this timing aspect later. The set Act of all actions is defined by:

$$Act \equiv Snd \cup Rec \cup Put_T \cup Get_T \cup Put_R \cup Get_R \cup \{\surd\}$$

A *trace* is a finite or infinite sequence of actions. The set of all traces with actions from Act is denoted by Act^ω. The set of all finite traces is denoted by Act*. We use some standard operations and relations on traces. Let $t, u, v \in Act^\omega$, $a, b, c, a_0, ..., a_n \in$ Act, $k, n \in \mathbb{N}$:

- $t \circ u$ denotes the concatenation of t and u. If t is infinite, then $t \circ u = t$.
- $t \sqsubseteq u$ ("t is a *prefix* of u") and $u \sqsupseteq t$ ("u *extends* t") hold exactly if $\exists v: t \circ v = u$.
- #t denotes the length of t; it is ∞ if t is infinite.
- $\langle a_0,...,a_n \rangle$ denotes the trace consisting of the actions $a_0,...,a_n$.
- $\langle\rangle$ denotes the empty trace.
- t[k] denotes the k-th element of t. For $t = \langle a_0,...,a_k,... \rangle$ with $\#t > k$, t[k] is defined to be a_k. We have "strong equality", i.e. t[k] = a implies that t[k] is defined.
- a©t denotes the filtered trace of t that contains only actions a, e.g. $a©\langle a,b,a,c \rangle = \langle a,a \rangle$. As a generalization of this *filter* operation, the first operand may also be a *set* of actions, e.g. $\{a,b\}©\langle a,b,a,c \rangle = \langle a,b,a \rangle$.
- a *in* t holds exactly if action a occurs in trace t. This is equivalent to $\#(a©t) > 0$.

We often need the auxiliary function data : $(Snd \cup Rec) \rightarrow UDU$ defined by data(snd(d)) = d, data(rec(d)) = d. This function is canonically extended to traces by stating:

$$data(\langle a_0,...,a_n,... \rangle) = \langle data(a_0),...,data(a_n),... \rangle$$

Convention: In the subsequent formulas we shall always use the following characteristic variables. This obviates the need to give the type (sort) of every variable in the formulas.

$t \in Act^{\omega}$ $\quad$ $d, d', d_0,...,d_n \in UDU$ $\quad$ $j, k, n \in \mathbb{N}$

$s \in Act^{*}$ $\quad$ $e \in LDU$

In a *trace specification* we specify the allowed traces using predicate logic formulas. A trace specification has the form $P_1(t) \wedge ... \wedge P_n(t)$, where each $P_i(t)$ is an (auxiliary) predicate on traces. Auxiliary predicates are introduced to structure a specification into several less complex requirements. We use the usual logical connectives. The symbol $\equiv$ reads "is defined by".

In our presentation, we distinguish between *safety* and *liveness* properties. Informally, a safety property is a property whose violation can always be detected after a finite amount of time, i.e. by considering a sufficiently large finite prefix of a trace. A liveness property is a property whose violation can only be detected after a complete, possibly infinite observation, i.e. a complete, possibly infinite trace (for a formal definition see [Alpern, Schneider 85]).

The initial requirements for the PEs are given by stating the service of the upper layer, which we want to supply, and the service of the lower layer, on which we rely.

The task of protocol development is as follows: given specifications of the service to be provided and of the service of the lower layer, develop specifications of the protocol entities such that the composition of these specifications with the lower layer specifications yields the service to be provided by the upper layer. Then, starting from the specifications of the protocol entities, develop executable programs for them.

3.2 Service specifications

a) Service to be provided by the upper layer:

$$US(t) \equiv \forall s \sqsubseteq t : data(Rec \copyright s) \sqsubseteq data(Snd \copyright s) \qquad (US)$$

$$UL(t) \equiv \#(Rec \copyright t) = \#(Snd \copyright t) \qquad (UL)$$

US expresses that nothing "wrong" is communicated; US is the safety requirement of the upper layer. UL says that finally the number of UDUs sent equals the number of the UDUs received; UL is the liveness requirement of the upper layer. Both conditions together specify reliable communication.

b) Service provided by the lower layer:

Let $z \in \{R,T\}$, where $\widetilde{R} = T$, $\widetilde{T} = R$:

$$LS(t) \equiv \forall s \sqsubseteq t : get_z(e) \; in \; s \Rightarrow put_{\widetilde{z}}(e) \; in \; s \qquad (LS)$$

$$LL(t) \equiv (\#(put_z(e) \copyright t) = \infty \Rightarrow \#(get_{\widetilde{z}}(e) \copyright t) = \infty) \wedge \#(\surd \copyright t) = \infty \qquad (LL)$$

LS requires that if some LDU arrives at one side, it must previously have been sent at the other side. The first part of LL states that if some message is sent infinitely often by one PE, then the other PE receives it infinitely often. So the medium is not broken forever. The second part of LL is a more intricate requirement: the action $\surd$ models the progress of time. It is considered an

action of the medium that is signalled to the transmitter. Thus the requirement just says that the transmitter receives some timing information. Actually, this requirement is only needed in section 5.

The service specification allows several designs for the protocol entities. For example, the well-known and extensively studied alternating bit protocol as well as the sliding window protocol would do. Our design will eventually lead to the *Stenning-protocol* [Stenning 76], which works as follows: at the transmitter, each UDU from the upper layer is associated with a unique sequence number and is sent repeatedly to the medium until an acknowledgement arrives that carries this sequence number. At the receiver the UDUs will be delivered to the upper layer service user in the order of their sequence numbers. Moreover, for each LDU received from the medium, the receiver will send back an acknowledgement with the corresponding sequence number.

The requirements to be provided by the upper layer cannot be separated in a simple way into local requirements for the transmitter, the receiver and the underlying medium. A requirement is called *local* to a component, if, for every input, it can be fulfilled by that component alone. We shall not discuss realizability in detail here (see for instance [Broy et al. 91]), but we just mention an important fact underlying our method: if we find agents (e.g. stream processing functions, see section 5) that generate traces fulfilling the local requirements, then the composition of the agents fulfils the logical conjunction of the requirements [Jonsson 90].

The localization of the upper layer requirements is the first goal in the development. The design steps are as follows:

1. Introduction of sequence numbers (section 3.3)
2. Introduction of acknowledgement messages (section 3.4)
3. Complete localization (section 3.5)

Formally, the design steps will be carried out by introducing some new requirements in every step. Old requirements may be dropped if they are consequences of the new ones. A design step is correct if the new trace predicate implies the previous one, i.e. if at most some further restriction is imposed on the traces of the system. A survey of our procedure can be found in appendix A.1.

3.3 Introduction of sequence numbers

We introduce *sequence numbers*. The service user's UDUs are tagged with natural numbers, so that the LDUs have the form $\langle k,d \rangle$ with $k \in \mathbb{N}$, $d \in$ UDU. We rely on the lower layer properties and additionally impose two safety and three liveness requirements that define the correct use of sequence numbers.

$$Step_1(t) \equiv LS(t) \wedge LL(t) \wedge S_1(t) \wedge S_2(t) \wedge L_1(t) \wedge L_2(t) \wedge L_3(t)$$

Safety:

$$S_1(t) \equiv \forall\, s \sqsubseteq t : data(Rec \copyright s) = \langle d_0,...,d_n \rangle \Rightarrow \forall\, k \leq n : get_R(\langle k,d_k \rangle)\ \textit{in}\ s \qquad (S_1)$$

$$S_2(t) \equiv \forall\, s \sqsubseteq t : put_T(\langle k,d \rangle)\ \textit{in}\ s \Rightarrow (Snd \copyright s)[k] = snd(d) \qquad (S_2)$$

The predicate S_1 expresses that a message is delivered to the upper layer only if all previous messages also have arrived at the receiver. S_2 specifies that the transmitter puts only those PDUs to the medium that contain an appropriate sequence number and a UDU which has already been sent. S_1 is local to the receiver, while S_2 is local to the transmitter.

Liveness:

$$L_1(t) \equiv \text{data}(\text{Snd} \copyright t) \sqsupseteq \langle d_0,...,d_n\rangle \Rightarrow \forall\, k \le n : \text{put}_T(\langle k,d_k\rangle)\ \textit{in}\ t \qquad (L_1)$$

$$L_2(t) \equiv \text{put}_T(\langle k,d\rangle)\ \textit{in}\ t \Rightarrow \text{get}_R(\langle k,d\rangle)\ \textit{in}\ t \qquad (L_2)$$

$$L_3(t) \equiv (\forall\, k \le n : \text{get}_R(\langle k,d_k\rangle)\ \textit{in}\ t) \Rightarrow \text{data}(\text{Rec} \copyright t) \sqsupseteq \langle d_0,...,d_n\rangle \qquad (L_3)$$

L_1 says that if some data is sent, then it is also eventually put onto the medium with the corresponding sequence number. L_2 describes that the receiver eventually obtains the informations sent by the transmitter. L_3 expresses that the messages delivered by the medium are eventually received by the upper layer. L_1 is local to the transmitter, L_3 is local to the receiver. L_2 is local to the medium, but not directly supported by it. So L_2 needs further refinement.
In the following we postpone most proofs to the appendix. We show here just one proof to give the flavour of the formal treatment:

Proposition 3.1: $\text{Step}_1(t) \Rightarrow \text{US}(t) \wedge \text{UL}(t)$, i.e. the first step is consistent with the service specification of the upper layer.
Proof: We have to prove US(t) and UL(t) provided $\text{Step}_1(t)$ holds.

$$\text{US}(t) \equiv \forall\, s \sqsubseteq t : \text{data}(\text{Rec} \copyright s) \sqsubseteq \text{data}(\text{Snd} \copyright s)$$

$\text{data}(\text{Rec} \copyright s) = \langle d_0,...,d_n\rangle$

$\Rightarrow$ $\forall\, k \le n : \text{get}_R(\langle k,d_k\rangle)\ \textit{in}\ s$ [by S_1]

$\Rightarrow$ $\forall\, k \le n : \text{put}_T(\langle k,d_k\rangle)\ \textit{in}\ s$ [by LS]

$\Rightarrow$ $\forall\, k \le n : (\text{Snd} \copyright s)[k] = \text{snd}(d_k)$ [by S_2]

$\Rightarrow$ $\text{data}(\text{Snd} \copyright s) \sqsupseteq \langle d_0 ...,d_n\rangle$ [definition of _[_]]

$$\text{UL}(t) \equiv \#(\text{Rec} \copyright t) = \#(\text{Snd} \copyright t)$$

Instead we prove a stronger version:

$$\text{UL}'(t) \equiv \text{data}(\text{Snd} \copyright t) \sqsupseteq \langle d_0,...,d_n\rangle \Rightarrow \text{data}(\text{Rec} \copyright t) \sqsupseteq \langle d_0,...,d_n\rangle$$

$\text{data}(\text{Snd} \copyright t) \sqsupseteq \langle d_0,...,d_n\rangle$

$\Rightarrow$ $\forall\, k \le n : \text{put}_T(\langle k,d_k\rangle)\ \textit{in}\ t$ [by L_1]

$\Rightarrow$ $\forall\, k \le n : \text{get}_R(\langle k,d_k\rangle)\ \textit{in}\ t$ [by L_2]

$\Rightarrow$ $\text{data}(\text{Rec} \copyright t) \sqsupseteq \langle d_0,...,d_n\rangle$ [by L_3] □

3.4 Introduction of acknowledgement messages

Two additional requirements, S_3 and L_4, are introduced to specify the correct use of acknowledgement messages. They replace requirement L_2.

$$Step_2(t) \equiv LS(t) \wedge LL(t) \wedge S_1(t) \wedge S_2(t) \wedge S_3(t) \wedge L_1(t) \wedge L_3(t) \wedge L_4(t)$$

Safety:

$$S_3(t) \equiv \forall\, s \sqsubseteq t : put_R(ack(k)) \textit{ in } s \Rightarrow \exists\, d : get_R(\langle k,d \rangle) \textit{ in } s \qquad (S_3)$$

S_3 says that before the acknowledgement for a certain message is sent to the medium, the corresponding message must have been received. It is a requirement local to the receiver.

Liveness:

$$L_4(t) \equiv put_T(\langle k,d \rangle) \textit{ in } t \Rightarrow get_T(ack(k)) \textit{ in } t \qquad (L_4)$$

L_4 expresses that every information sent to the medium will get the corresponding acknowledgement. This requirement is local to the medium, but not directly supported by it; hence it needs further treatment. L_2 does not have to be mentioned explicitly any more because it is implied by the other predicates of $Step_2$.

Proposition 3.2: $Step_2(t) \Rightarrow Step_1(t)$, i.e. $Step_2(t) \Rightarrow L_2(t)$

Proof: See appendix A.2 □

3.5 Complete localization

We choose two additional liveness properties, L_5 and L_6, to replace the liveness requirement L_4. In fact, also L_1 is implied by $Step_3$ and therefore omitted.

$$Step_3(t) \equiv LS(t) \wedge LL(t) \wedge S_1(t) \wedge S_2(t) \wedge S_3(t) \wedge L_3(t) \wedge L_5(t) \wedge L_6(t)$$

Liveness:

$$L_5(t) \equiv \#(Snd \copyright t) > k \wedge \neg\, A(t,k) \wedge \#(\surd\copyright t) = \infty \Rightarrow$$
$$\exists\, j, d : \neg\, A(t,j) \wedge \#(put_T(\langle j,d \rangle) \copyright t) = \infty$$
$$\textit{where } A(t,k) \equiv \exists\, s : s \circ get_T(ack(k)) \sqsubseteq t \wedge \#(Snd \copyright s) > k \qquad (L_5)$$

$$L_6(t) \equiv \#(get_R(\langle k,d \rangle) \copyright t) = \infty \Rightarrow \#(put_R(ack(k)) \copyright t) = \infty \qquad (L_6)$$

Requirement L_5 is a little bit intricate. It might be the case that some acknowledgement ack(k) arrives at the transmitter before the k-th message has actually been sent. The formula A(t,k) says that there has been a "proper" acknowledgement for the k-th message, i.e. one that did not occur before at least k messages have been sent to the transmitter. So L_5 says that if at least k UDUs are sent and the k-th UDU has not got a "proper" acknowledgement yet, then the transmitter will send some not "properly" acknowledged UDUs (not necessarily the k-th one) infinitely often. In a previous version we just required the condition L_5':

$L_5'(t) \equiv \#(Snd \copyright t) > k \wedge \neg get_T(ack(k))\ \textit{in}\ t \Rightarrow$

$\exists j, d : \neg get_T(ack(j))\ \textit{in}\ t \wedge \#(put_T(\langle j,d \rangle) \copyright t) = \infty$

However, consider the following scenario: a system run starts with $\langle get_T(ack(0)), snd(d), snd(d'),... \rangle$. In fact the receiver's and the medium's specification are such that this situation will not occur, but according to L_5' the transmitter must be prepared for it. The first acknowledgement is not a "proper" acknowledgement, because nothing has been sent yet. L_5' would force the transmitter to output $put_T(\langle 1,d' \rangle)$ infinitely often, even if the first message d will never be successfully transmitted.

Property L_6 expresses that if the receiver gets an information infinitely often, it will also send the corresponding acknowledgement infinitely often. L_5 is local to the transmitter, L_6 is local to the receiver. L_4 (and also L_1) are implied by $Step_3$.

Proposition 3.3: $Step_3(t) \Rightarrow Step_2(t)$, i.e. $Step_3(t) \Rightarrow L_1(t) \wedge L_4(t)$

Proof: See appendix A.2 □

We may now assign the requirements to the protocol entities: the requirements local to the transmitter are S_2 and L_5, the requirements local to the receiver are S_1, S_3, L_3 and L_6. The other requirements are provided by the lower layer. Hence all requirements are localized.

Transmitter:

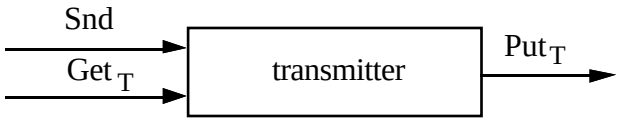

Fig. 3

$S_2(t) \equiv \forall s \sqsubseteq t : put_T(\langle k,d \rangle)\ \textit{in}\ s \Rightarrow (Snd \copyright s)[k] = snd(d)$

$L_5(t) \equiv \#(Snd \copyright t) > k \wedge \neg A(t,k) \wedge \#(\surd \copyright t) = \infty \Rightarrow$

$\exists j, d : \neg A(t,j) \wedge \#(put_T(\langle j,d \rangle) \copyright t) = \infty$

where $A(t,k) \equiv \exists s : s \circ get_T(ack(k)) \sqsubseteq t \wedge \#(Snd \copyright s) > k$

Receiver:

Fig. 4

$S_1(t) \equiv \forall s \sqsubseteq t : data(Rec \copyright s) = \langle d_0,...,d_n \rangle \Rightarrow \forall k \le n : get_R(\langle k,d_k \rangle)\ \textit{in}\ s$

$S_3(t) \equiv \forall s \sqsubseteq t : put_R(ack(k))\ \textit{in}\ s \Rightarrow \exists d : get_R(\langle k,d \rangle)\ \textit{in}\ s$

$L_3(t) \equiv (\forall k \le n : get_R(\langle k,d_k \rangle)\ \textit{in}\ t) \Rightarrow data(Rec \copyright t) \sqsupseteq \langle d_0,...,d_n \rangle$

$L_6(t) \equiv \#(get_R(\langle k,d \rangle) \copyright t) = \infty \Rightarrow \#(put_R(ack(k)) \copyright t) = \infty$

In the rest of this paper we will develop an implementation of the transmitter. An analogous development could be done for the receiver. This is omitted here.

4. From trace specifications to functional specifications

We now aim at giving a *functional specification* of the transmitter, i.e. we want to give a predicate on stream-processing functions such that the traces generated by the functions that fulfil the predicate satisfy the transmitter's trace specification. It turns out that, in the given case, this can be done by the schematic application of some transformations rules. A stream-processing function models an agent that receives streams of input messages on its input channels and transforms them into streams of output messages to be sent on its output channels.

For the formal treatment of the relationship between a component's trace specification and its functional specification, we need to define which traces are generated by a stream-processing function. The following definition is taken from [Broy et al. 92]: let $f \in I^\omega \to O^\omega$ be a stream processing function with $I \cap O = \emptyset$, such that inputs and outputs can be distinguished in the generated traces. For a trace $t \in (I \cup O)^\omega$, f *generates* t is given by:

$$f \textit{ generates } t \equiv f(I \copyright t) = O \copyright t \wedge \forall s \sqsubseteq t : O \copyright s \sqsubseteq f(I \copyright s)$$

Here t ranges over $(I \cup O)^\omega$, and s ranges over $(I \cup O)^*$. The second part of the conjunction says that there may be an arbitrary, but finite, delay for the output actions.

The intended relation between a trace specification T and the corresponding functional specification $Func_T$ is as follows:

$$Func_T(f) \equiv \forall t : f \textit{ generates } t \Rightarrow T(t)$$

Expanding f *generates* t, $Func_T(f)$ reads:

$$Func_T(f) \equiv \forall t : ((f(I \copyright t) = O \copyright t \wedge (\forall s \sqsubseteq t : O \copyright s \sqsubseteq f(I \copyright s))) \Rightarrow T(t))$$

The following proposition indicates that for particular safety predicates, $Func_T(f)$ can be given in a more "tractable" form.

Proposition 4.1: Let $T(t) \equiv \forall s \sqsubseteq t : T'(I \copyright s, O \copyright s)$, and $Func_T$ be defined as above. Then:

$$Func_T(f) \Leftrightarrow (\forall x \in I^*, y \in O^* : y \sqsubseteq f.x \Rightarrow T'(x,y))$$

Proof: See appendix A.3 □

On the functional level we often write f.x instead of f(x) in order to avoid brackets. Now we can give a functional specification of the transmitter. Recall the predicate S_2. Defining $I \equiv Snd \cup Get_T \cup \{\surd\}$ and $O \equiv Put_T$, $S_2(t)$ is equivalent to:

$$\forall s \sqsubseteq t : put_T(\langle k,d \rangle) \textit{ in } (O \copyright s) \Rightarrow (Snd \copyright (I \copyright s))[k] = snd(d)$$

By proposition 4.1 we can transform this trace predicate into the function predicate

$$Func_{S_2}(f) \equiv \forall x \in I^*, y \in O^* : y \sqsubseteq f.x \Rightarrow (put_T(\langle k,d \rangle) \textit{ in } y \Rightarrow (Snd \copyright x)[k] = snd(d))$$

Note that the auxiliary variable y in $Func_{S_2}$ is superfluous. It is easy to see that $Func_{S_2}$ is equivalent to $Func_{S_2}'$, which is defined by:

$$Func_{S_2}'(f) \equiv \forall x \in I^* : put_T(\langle k,d \rangle) \textit{ in } f.x \Rightarrow (Snd \copyright x)[k] = snd(d)$$

Proposition 4.2 is the counterpart of proposition 4.1 for liveness properties:

Proposition 4.2: Let $T(t) \equiv T'(I©t,O©t)$, and let $Func_T$ be defined as above. Then:

$$Func_T(f) \Leftrightarrow \forall\, x \in I^{\omega} : T'(x, f.x)$$

Proof: See appendix A.3 □

Recall the predicate L_5. Let I and O be defined as before. Then $L_5(t)$ is equivalent to

$$\#(Snd © (I©t)) > k \wedge \neg A(I©t,k) \wedge \#(\surd © (I©t)) = \infty \Rightarrow$$
$$\exists\, j, d : \neg A(I©t,j) \wedge \#(put_T(\langle j,d\rangle) © (O©t)) = \infty$$

By proposition 4.2 we can transform this trace predicate into the function predicate

$$Func_{L_5}(f) \equiv \forall\, x \in I^{\omega} : \#(Snd © x) > k \wedge \neg A(x,k) \wedge \#(\surd © x) = \infty \Rightarrow$$
$$\exists\, j, d : \neg A(x,j) \wedge \#(put_T(\langle j,d\rangle) © f.x) = \infty$$

Since we will only be talking about the transmitter from now on, it is no longer necessary to use actions built by applications of put_T and get_T, respectively. It suffices to use just the parameters of these actions. For the rest of this paper, let $I \equiv Snd \cup Ack \cup \{\surd\}$, $O \equiv Info$. Then the transmitter will be a function from I^{ω} to O^{ω}, which is specified by:

$$TS(f) \equiv \forall\, x \in I^{*} : \langle k,d\rangle \textit{ in } f.x \Rightarrow (Snd © x)[k] = snd(d)) \qquad (TS)$$

$$TL(f) \equiv \forall\, x \in I^{\omega} : \#(Snd © x) > k \wedge \neg A(x,k) \wedge \#(\surd © x) = \infty \Rightarrow$$
$$\exists\, j, d : \neg A(x,j) \wedge \#(\langle j,d\rangle © f.x) = \infty$$
$$\textit{where}\quad A(x,k) \equiv \exists\, s : s \circ ack(k) \sqsubseteq x \wedge \#(Snd © s) > k \qquad (TL)$$

The two predicates TS and TL are the starting point for the functional development of the transmitter in the next section. This is a functional specification of just the "data processing part" of the transmitter. We suppose that the inputs for the transmitter appear in interleaved form on a single input stream. See [Broy et al. 92] for a discussion on how the merging of streams can be handled.

5. Development on the functional level

In this section the functional specification of the transmitter, as given by the predicates TS and TL, is transformed step by step into a functional program. There are three major subsections:

1. Specification of the transmitter in a form that models stepwise computation, and refinement towards the Stenning-protocol (section 5.1)
2. Introduction of an explicit notion of state (section 5.2)
3. Functional implementation (section 5.3)

5.1 Developing a strategy for the transmitter

From now on a more operational specification style will be employed. We imagine the stepwise computation process performed by the transmitter as follows: the transmitter repeatedly receives one input element and reacts to every input by issuing a sequence of outputs based on all the inputs received so far. A *strategy* is a function that defines which additional outputs are produced as soon as an additional input element arrives. A schema for functional specification in this style is given below, where H is a predicate on strategies. The variables $z \in I^*$, $i \in I$, $h \in I^* \to O^*$ are used:

$$Strat_H(f) \equiv \exists\, h : \forall\, z, i : f.\langle\rangle = \langle\rangle \wedge f(z \circ i) = f.z \circ h(z \circ i) \wedge H(h)$$

This schema says that the output of the transmitter for some input $z \circ i$ consists of the output produced for the input sequence z concatenated with the additional output elements described by the strategy h. Note that $Strat_H$ imposes explicit conditions only on finite inputs. In the functional setting, we have the additional (implicit) constraint that only *continuous* functions are considered. These are functions whose behaviour for infinite inputs follows from their behaviour for finite inputs. See page 29 of [Broy et al. 92] for more information on this class of functions.

We now have to find a strategy specification H such that the transmitter specification is implied by every function that fulfils $Strat_H$. In the first step, we will only try to achieve TS. Recall the safety specification TS derived in the last section. Fortunately, this predicate, which is a condition on *whole* output streams, can equivalently be used as a condition on *steps* of the output (produced by the strategy). Let $z \in I^*$, $k \in \mathbb{N}$, and $d \in UDU$:

$$T_1(f) \equiv Strat_{H_1}(f)$$

$$H_1(h) \equiv \forall\, z, k, d : \langle k,d \rangle \textit{ in } h.z \Rightarrow (Snd©z)[k] = snd(d)$$

Proposition 5.1: $T_1(f) \Leftrightarrow TS(f)$

Proof:

"$TS(f) \Rightarrow T_1(f)$": Obvious.

"$TS(f) \Leftarrow T_1(f)$": Take any f such that $T_1(f)$ holds. We show $TS(f)$ by induction.

Base case: By the definition of T_1, we have $f.\langle\rangle = \langle\rangle$, which implies $TS(f)$.

Inductive case: Take any $z \in I^*$ and $i \in I$. The induction assumption is $\langle k,d \rangle$ *in* $f.z \Rightarrow (Snd©z)[k] = snd(d)$. From T_1 we see that $f(z \circ i) = f.z \circ h(z \circ i)$, and together with the definition of H_1 we obtain TS, namely: $\langle k,d \rangle$ *in* $f(z \circ i) \Rightarrow (Snd©(z \circ i))[k] = snd(d)$. □

The predicate T_1 does not imply the liveness requirement TL. Therefore we have to strengthen the specification. Actually, only the strategy specification H is strengthened, while the template $Strat_H$ remains unchanged.

As a first design decision we demand that at most one output element should be produced for every newly arriving input. We do not state the conditions under which some output is produced at all, but if there is some output, then it must be allowed by the old strategy H_1. This design

decision rules out any implementation which sends more than one information LDU in a single time interval.

$T_2(f) \equiv Strat_{H_2}(f)$

$H_2(h) \equiv \forall\, z : h.z = \langle\rangle \vee (\exists\, k, d : h.z = \langle k,d \rangle \wedge (Snd©z)[k] = snd(d))$

Proposition 5.2: $T_2(f) \Rightarrow T_1(f)$
Proof: Obvious since $H_2(h) \Rightarrow H_1(h)$. The reverse direction does not hold. □

The next step introduces two further constraints on when some output may be generated. In particular, we demand that an information LDU may only be output if its sequence number is the smallest one that has not been acknowledged so far. This design decision rules out all sliding window protocols with a window size larger than one. Some auxiliary predicates are used to structure the specification. The formula $Out(x,k,d)$ expresses that the information packet $\langle k,d \rangle$ may be output after the input sequence x has arrived. The predicate A is already known from the previous sections; $A(x,k)$ expresses that the k-th information LDU has properly been acknowledged. Let $x \in I^\omega$, $s \in I^*$ and the other variables as above:

$T_3(f) \equiv Strat_{H_3}(f)$

$H_3(h) \equiv \forall\, z : h.z = \langle\rangle \vee (\exists\, k, d : h.z = \langle k,d \rangle \wedge Out(z,k,d))$

$Out(x,k,d) \equiv (Snd©x)[k] = snd(d) \wedge \neg A(x,k) \wedge \forall\, j < k : A(x,j)$

where $A(x,k) \equiv \exists\, s : s \circ ack(k) \sqsubseteq x \wedge \#(Snd©s) > k$

Proposition 5.3: $T_3(f) \Rightarrow T_2(f)$
Proof: Obvious since $H_3(h) \Rightarrow H_2(h)$. The reverse direction does not hold. □

We will later need the lemma that the predicate Out, for any given $x \in I^\omega$, can be fulfilled by at most one pair of values $k \in \mathbb{N}$ and $d \in UDU$:

Lemma 5.4: $Out(x,k,d) \wedge Out(x,k',d') \Rightarrow k = k' \wedge d = d'$
Proof: Take arbitrary x, k, k', d, d' such that the left hand side of the implication holds. Without loss of generality assume $k' \leq k$. The additional assumption $k' < k$ leads to a contradiction: from $Out(x,k,d)$ we infer $\forall\, j < k : A(x,j)$ and in particular $A(x,k')$ while from $Out(x,k',d')$ we infer $\neg A(x,k')$. Therefore $k' = k$ must hold. From $snd(d) = (Snd©x)[k] = (Snd©x)[k'] = snd(d')$ it follows that $d = d'$. □

A strategy fulfilling H_3 need not output anything. In the last design decision of this section, we demand that some output must be produced if it is possible at all. In fact, the strategy is now uniquely defined, i.e., H_4 is fulfilled by exactly one (continuous) function.

$T_4(f) \equiv Strat_{H_4}(f)$

$H_4(h) \equiv \forall\, z : (\neg \exists\, k, d : h.z = \langle\rangle \wedge Out(z,k,d)) \vee (\exists\, k, d : h.z = \langle k,d \rangle \wedge Out(z,k,d))$

Proposition 5.5: $T_4(f) \Rightarrow T_3(f)$
Proof: Obvious since $H_4(h) \Rightarrow H_3(h)$. The reverse direction does not hold. □

Summing up the developments done so far, it is clear that $T_4(f) \Rightarrow TS(f)$. In can also be shown that every function which fulfils T_4 also represents a live behaviour of the transmitter. This is formally stated by the following proposition:

Proposition 5.6: $T_4(f) \Rightarrow TL(f)$

Proof: Recall the definition of TL, where $x \in I^\omega$, $s \in I^*$, $k, j \in \mathbb{N}$ and $d \in UDU$.

$$TL(f) \equiv \forall\, x : \#(Snd \copyright x) > k \wedge \neg A(x,k) \wedge \#(\surd \copyright x) = \infty \Rightarrow$$
$$\exists\, j, d : \neg A(x,j) \wedge \#(\langle j,d \rangle \copyright f.x) = \infty$$
$$\textit{where} \quad A(x,k) \equiv \exists\, s : s \circ ack(k) \sqsubseteq x \wedge \#(Snd \copyright s) > k$$

Take arbitrary x and k such that $\#(Snd \copyright x) > k \wedge \neg A(x,k) \wedge \#(\surd \copyright x) = \infty$ holds (*). We have to show that there exist $j \in \mathbb{N}$ and $d \in UDU$ such that $\neg A(x,j) \wedge \#(\langle j,d \rangle \copyright f.x) = \infty$. In order to do this, we distinguish two cases:

Case 1: There exist j, d such that $j < k$ and $Out(x,j,d)$.

According to lemma 5.4, j and d are then uniquely defined. The definition of Out implies that $\neg A(x,j)$, so it is only left to show $\#(\langle j,d \rangle \copyright f.x) = \infty$. Note that there must be a finite $s \sqsubseteq x$ such that $Out(s,j,d)$ holds. From the definition of T_4 it is clear that one message $\langle j,d \rangle$ is output for every additional input after s. Since we assumed in (*) that the input stream contains infinitely many elements, the output stream must also contain infinitely many elements $\langle j,d \rangle$.

Case 2: There are no j, d such that $j < k$ and $Out(x,j,d)$.

We show that in this case, for some suitable d, the three conjuncts of $Out(x,k,d)$ hold:

- $(Snd \copyright x)[k] = snd(d)$ [can be fulfilled by a suitable d, since we assumed $\#(Snd \copyright x) > k$]
- $\neg A(x,k)$ [was assumed above in (*)]
- $\forall\, j < k : A(x,j)$ [otherwise $Out(x,j,d')$ would hold for some $j < k$ and $d' \in UDU$]

Therefore $Out(x,k,d)$ holds for some d. By the same argument as in case 1, the output stream must contain infinitely many elements $\langle k,d \rangle$. □

5.2 From specification to implementation

The specification given so far described the transmitter's behaviour for every input sequence of the form $x \circ i$. In functional programming, however, the reverse pattern $i \circ x$ is used. Therefore the specification schema has to be changed accordingly. This is a generic equivalence transformation. As an additional parameter in the new recursive specification schema we need the sequence of inputs received so far. This sequence is called the "implicit state" because it plays the rôle of a state parameter and represents the full history of the computation so far. The new variable $g \in I^* \rightarrow (I^\omega \rightarrow O^\omega)$ is used in T_5:

$$T_5(f) \equiv \exists\, g, h : f = g_{\langle\rangle} \wedge \forall\, s, i, x : g_s.\langle\rangle = \langle\rangle \wedge g_s.(i \circ x) = h.(s \circ i) \circ g_{(s \circ i)}.x \wedge H_5(h)$$
$$H_5(h) \equiv H_4(h)$$

Proposition 5.7: $T_5(f) \Leftrightarrow T_4(f)$

Proof: Take any strategy h that fulfils H_4 (and, equivalently, H_5). Take any $f \in I^\omega \to O^\omega$ such that $f.\langle\rangle = \langle\rangle \wedge f(x \circ i) = f.x \circ h(x \circ i)$ holds, and take an arbitrary $g \in I^* \to (I^\omega \to O^\omega)$ such that $g_s.\langle\rangle = \langle\rangle \wedge g_s.(i \circ x) = h.(s \circ i) \circ g_{(s \circ i)}.x$ holds. Because of continuity, it suffices to show $f.z = g_{\langle\rangle}.z$ for every finite $z \in I^*$:

$$
\begin{aligned}
& f.\langle z_1, ..., z_n\rangle \;= \\
& \quad ... \;=\; ... \;=\; f.\langle\rangle \circ h.\langle z_1\rangle \circ h.\langle z_1,z_2\rangle \circ ... \circ h.\langle z_1, ..., z_{n-1}\rangle \circ h.\langle z_1, ..., z_n\rangle \\
& \qquad\qquad = h.\langle z_1\rangle \circ h.\langle z_1,z_2\rangle \circ ... \circ h.\langle z_1, ..., z_{n-1}\rangle \circ h.\langle z_1, ..., z_n\rangle \\
& \qquad\qquad = h.\langle z_1\rangle \circ h.\langle z_1,z_2\rangle \circ ... \circ h.\langle z_1, ..., z_{n-1}\rangle \circ h.\langle z_1, ..., z_n\rangle \circ g_{\langle z_1,...,z_n\rangle}.\langle\rangle \\
& \quad ... \;=\; ... \;=\; g_{\langle\rangle}.\langle z_1, ..., z_n\rangle
\end{aligned}
$$

□

A transmitter implementation that actually uses the implicit state would be very inefficient. We would rather like to save just the *essential information*, which will be required to determine the future outputs. This essential information is called the *constructed state*. It is a design step to decide exactly what has to be stored such that the constructed state contains sufficient information and is easy to access and update. In our formal framework, we have to define the set of possible constructed states (a *state space*) and a function that describes the connection between implicit and constructed states.

Given a state space $State$ and an appropriate connection function $\varphi : I^* \to State$, the following specification schema is constructed from T_5 by replacing every implicit state by the corresponding constructed state:

$$
\begin{aligned}
T_6(f) \;\equiv\; & \exists\, g, h : f = g_{(\varphi.\langle\rangle)} \;\wedge \\
& \qquad \forall\, s, i, x : g_{(\varphi.s)}.\langle\rangle = \langle\rangle \;\wedge\; g_{(\varphi.s)}.(i \circ x) = h.(\varphi.(s \circ i)) \circ g_{(\varphi.(s \circ i))}.x \;\wedge\; H_6(h)
\end{aligned}
$$

It remains to give a strategy specification H_6 such that the specifications T_5 and T_6 are equivalent (note that a strategy now is a mapping from constructed states $State$ to streams of outputs). From the definition of H_5 we see that the (implicit) state is only used in the predicate Out. Therefore it is sufficient to find a predicate Out' that is equivalent to Out for each constructed state, i.e., which meets the following requirement:

$$\forall\, z, k, d : Out(z,k,d) \;\Leftrightarrow\; Out'(\varphi.z,k,d) \qquad (0)$$

Given such a predicate Out', an appropriate strategy specification H_6 can be derived from H_5 by replacing the predicate Out with the equivalent Out'. Let $h \in State \to O^*$, $\sigma \in State$, and the other variables as above:

$$H_6(h) \;\equiv\; \forall\, \sigma : (\neg\, \exists\, k, d : h.\sigma = \langle\rangle \;\wedge\; Out'(\sigma,k,d)) \;\vee\; (\exists\, k, d : h.\sigma = \langle k,d\rangle \;\wedge\; Out'(\sigma,k,d))$$

Proposition 5.8: $T_6(f) \Leftrightarrow T_5(f)$

Proof: Given the (finite) sequence s of previous inputs and an additional input i, the sequences of additional outputs specified by T_5 and T_6 are $h.(s \circ i)$ and $h'.(\varphi.(s \circ i))$, respectively, where $H_5(h)$ and $H_6(h')$ hold. By the definitions of H_5 and H_6 and the equivalence (0) of Out and Out' it follows that $h.(s \circ i) = h'.(\varphi.(s \circ i))$. □

Now the constructed state for the transmitter will be defined: it is obvious that the sequence of pending information LDUs must be a part of the state space (a pending LDU is one that has been sent to the transmitter, but for which no proper acknowledgement has been received so far). Since the transmitter must also assign sequence numbers to newly arriving UDUs, another necessary piece of information is the next valid sequence number. Therefore, our state space will be a tuple consisting of an integer and a sequence of information LDUs:

$$\text{State} \equiv \mathbb{N} \times \text{Info}^*$$

We already stated the informal meaning of the state components. This is formally described by a function φ that takes an implicit state and returns the corresponding constructed state. We now specify φ by giving the requirements (1), (2), and (3). Formula (1) expresses that the next valid sequence number is just the number of UDUs received as input so far. Requirements (2) and (3) state that the sequence of waiting PDUs contains exactly those informations that have been sent but not acknowledged. Note that (2) only appears to make this specification of φ easier to read; it is actually implied by (3). Let $z \in I^*$, $i, j, k \in \mathbb{N}$, $d \in$ UDU:

$$\varphi : I^* \to \text{State}$$

$$\varphi.z = (\varphi_1.z, \varphi_2.z)$$

$$\varphi_1.z = \#(\text{Snd}©z) \tag{1}$$

$$\#(\varphi_2.z) = |\{ j < \#(\text{Snd}©z) : \neg A(z,j) \}| \tag{2}$$

$$(\varphi_2.z)[i] = ‹k,d› \Leftrightarrow (\text{Snd}©z)[k] = \text{snd}(d) \wedge \neg A(z,k) \wedge |\{ j < k : \neg A(z,j) \}| = i \tag{3}$$

Having specified requirements for the mapping from the implicit state to the constructed one, we next want to refine T_6 using a more explicit formula for Out'. Suppose that that φ meets the specifications (1) to (3). Then the predicate Out' can actually be calculated from Out such that requirement (0) is fulfilled:

$$\text{Out}'((\varphi_1.z,\varphi_2.z),k,d)$$

[definition of φ]	$\Leftrightarrow$	$\text{Out}'(\varphi.z,k,d)$
[by (0)]	$\Leftrightarrow$	$\text{Out}(z,k,d)$
[definition of Out]	$\Leftrightarrow$	$(\text{Snd}©z)[k] = \text{snd}(d) \wedge \neg A(z,k) \wedge \forall\, j < k : A(z,j)$
[set comprehension]	$\Leftrightarrow$	$(\text{Snd}©z)[k] = \text{snd}(d) \wedge \neg A(z,k) \wedge \|\{ j < k : \neg A(z,j) \}\| = 0$
[by (3)]	$\Leftrightarrow$	$(\varphi_2.z)[0] = ‹k,d›$

So we obtain the following definition of Out' for $n \in \mathbb{N}$ and $q \in$ Info:

$$\text{Out}'((n,q),k,d) \equiv q[0] = ‹k,d› \tag{4}$$

It is obvious that, given some n and q, the predicate Out'((n,q),k,d) can be fulfilled exactly if $q \neq \langle\rangle$. This observation can be used to further simplify the auxiliary predicate H_7:

$$T_7(f) \equiv \exists\, g, h : f = g_{(\varphi.\langle\rangle)} \wedge \forall\, s, i, x : g_{(\varphi.s)}.\langle\rangle = \langle\rangle \wedge g_{(\varphi.s)}.(i \circ x) = h.(\varphi.(s \circ i)) \circ g_{(\varphi.(s \circ i))}.x \wedge H_7(h)$$

$$H_7(h) \equiv \forall\, n, q : (q = \langle\rangle \Rightarrow h.(n,q) = \langle\rangle) \wedge (q \neq \langle\rangle \Rightarrow h.(n,q) = q[0])$$

Proposition 5.9: $T_7(f) \Leftrightarrow T_6(f)$

Proof: Since the construction of (4) guarantees that the specification (0) is met. □

So we finally achieved a very operational description of the transmitter. Note that T_7 is equivalent to T_4 because of propositions 5.7, 5.8, and 5.9. Hence T_7 implies TS (by the remark before proposition 5.6) and TL (by proposition 5.6).

5.3 Implementing the transmitter

In the previous section we essentially obtained a recursive definition of the transmitter function. The state mapping function φ, however, was still specified in a non-executable way. Using the same techniques of stepwise refinement as in the development done so far, the following recursive definition of φ can be found:

$$\begin{aligned}
\varphi_1.\langle\rangle &= 0 & \qquad \varphi_2.\langle\rangle &= \langle\rangle \\
\varphi_1.(z \circ \surd) &= \varphi_1.z & \qquad \varphi_2.(z \circ \surd) &= \varphi_2.z \\
\varphi_1.(z \circ \mathrm{snd}(d)) &= \varphi_1.z + 1 & \qquad \varphi_2.(z \circ \mathrm{snd}(d)) &= \varphi_2.z \circ \langle \varphi_1.z, d \rangle \\
\varphi_1.(z \circ \mathrm{ack}(k)) &= \varphi_1.z & \qquad \varphi_2.(z \circ \mathrm{ack}(k)) &= \mathrm{remove}.(k, \varphi_2.z)
\end{aligned}$$

$$\begin{aligned}
&\mathrm{remove}.(a, \langle\rangle) = \langle\rangle \\
&\mathrm{remove}.(a, \langle k,d \rangle \circ x) = \textit{if } a = k \textit{ then } x \textit{ else } \langle k,d \rangle \circ \mathrm{remove}.(a,x)
\end{aligned}$$

We have the proof obligation to show that the specifications (1), (2) and (3) are met by the recursive definitions given above. This is done by induction on the implicit states. The actual proof is quite long and tedious. It can be found in appendix A.4. We need a theorem specifying remove to complete the proof. It is a general observation that the functions used in a program development (in this case φ_1, φ_2, remove) must both be specified by a formula of predicate logic and by a recursive definition because the specification is needed as induction hypothesis.

Now the transition to a functional programming language style is obvious. We expand the occurrences of φ and use a standard form of pattern matching:

$$\mathrm{transmitter} : I^\omega \rightarrow O^\omega$$
$$\mathrm{transmitter} = \mathrm{trans}.(0, \langle\rangle)$$

$$\mathrm{trans}.(n, q).\langle\rangle = \langle\rangle$$

$$\begin{aligned}
&\mathrm{trans}.(n, \langle\rangle).(\surd \circ x) = \mathrm{trans}.(n, \langle\rangle).x \\
&\mathrm{trans}.(n, \langle\rangle).(\mathrm{snd}(d) \circ x) = \mathrm{trans}.(n+1, \langle n,d \rangle).x \\
&\mathrm{trans}.(n, \langle\rangle).(\mathrm{ack}(k) \circ x) = \mathrm{trans}.(n, \langle\rangle).x
\end{aligned}$$

$$\begin{aligned}
&\mathrm{trans}.(n, \langle k,d \rangle \circ q).(\surd \circ x) = \langle k,d \rangle \circ \mathrm{trans}.(n, \langle k,d \rangle \circ q).x \\
&\mathrm{trans}.(n, \langle k,d \rangle \circ q).(\mathrm{snd}(e) \circ x) = \langle k,d \rangle \circ \mathrm{trans}.(n+1, \langle k,d \rangle \circ q \circ \langle n,e \rangle).x \\
&\mathrm{trans}.(n, \langle k,d \rangle \circ q).(\mathrm{ack}(a) \circ x) = \langle k,d \rangle \circ \mathrm{trans}.(n, \mathrm{remove}.(a, \langle k,d \rangle \circ q)).x
\end{aligned}$$

$$\begin{aligned}
&\mathrm{remove}.(a, \langle\rangle) = \langle\rangle \\
&\mathrm{remove}.(a, \langle k,d \rangle \circ x) = \textit{if } a = k \textit{ then } x \textit{ else } \langle k,d \rangle \circ \mathrm{remove}.(a,x)
\end{aligned}$$

6. The transmitter in the programming languages AL and PL

Two programming languages, AL and PL, have been proposed in [Broy et al. 92]. In the concrete syntax of the functional language AL, the transmitter program developed above in section 5.3 has the following form. We use the two auxiliary functions ackn : Ack → ℕ defined by ackn(ack(n)) = n and seqn : Info → ℕ defined by seqn(‹k,d›) = k.

program *transmitter* ≡ **chan I** *x* → **chan O** *o* :
  o ≡ *trans*(0,⟨⟩,*x*)
endprogram

agent *trans* ≡ **Nat** *n*, **seq Info** *q*, **chan I** *x* → **chan O** *o* :
  o ≡ **if** *ft.x* = √ ∧ *q* = ⟨⟩ **then** *trans*(*n*,⟨⟩,*rt.x*)
  elif *ft.x* ∈ Snd ∧ *q* = ⟨⟩ **then** *trans*(*n*+1,‹*n*,*data*(*ft.x*)›,*rt.x*)
  elif *ft.x* ∈ Ack ∧ *q* = ⟨⟩ **then** *trans*(*n*,⟨⟩,*rt.x*)
  elif *ft.x* = √ ∧ *q* ≠ ⟨⟩ **then** *ft.q* ◦ *trans*(*n*,*q*,*rt.x*)
  elif *ft.x* ∈ Snd ∧ *q* ≠ ⟨⟩ **then** *ft.q* ◦ *trans*(*n*+1,*q*◦‹*n*,*data*(*ft.x*)›,*rt.x*)
  elif *ft.x* ∈ Ack ∧ *q* ≠ ⟨⟩ **then** *ft.q* ◦ *trans*(*n*,*remove*(*ackn*(*ft.x*),*q*),*rt.x*) **fi**
endagent

func *remove* ≡ **Nat** *a*, **seq Info** *x* → **seq Info** :
  if *x* = ⟨⟩ **then** ⟨⟩ **else if** *a* = *seqn*(*ft.x*) **then** *rt.x* **else** *ft.x* ◦ *remove*(*a*,*rt.x*) **fi fi**

The AL program given above is simply the definition of a state-automaton, which is a close link to the formal description technique Estelle. Note that we did not use any functional programming features which would be hard to implement in an imperative language. In particular, *trans* is tail-recursive modulo concatenation. Hence the transition to a program in the imperative language PL is a straightforward application of schematic transformation rules. We do not go into details here. The rule *recursion-to-iteration II* (given on page 57 of [Broy et al. 92]) is essentially what we need to transform the agent *trans* into the following imperative program:

agent *trans* ≡ **Nat** *n*, **seq Info** *q*, **chan I** *x* → **chan O** *o* :
  var Nat *count* := *n*; **var seq Info** *queue* := *q*; **var I** *input*;
  loop *x*?*input*;
    if *queue* ≠ ⟨⟩ **then** *o*!*ft.queue* **fi**;
    if *input* = √ **then** *skip*
    elif *input* ∈ Snd **then** *queue* := *queue* ◦ ‹*count*,*data*(*input*)›; *count* := *count* +1;
    elif *input* ∈ Ack **then** *queue* := *remove*(*ackn*(*input*),*queue*) **fi**
  endloop
endagent

7. Conclusion

We have shown the design of protocol entities from non-constructive service specifications towards executable imperative programs. All design steps were formally proved or justified. The proofs were of a mathematical nature; we plan to investigate how the deductions can be carried out in a formal calculus using an interactive verification system.

The paper shows that distributed system development from formal requirements to executable programs is possible in FOCUS. However, even for this small example the whole proof procedure becomes very lengthy and tedious. In our opinion this is not a characteristic of just our method, but shows that the design and verification of (distributed) systems is a complicated matter in general (see [Broy 87], [Chandy, Misra 88], [Lamport 90] for similar observations).

For the development of distributed systems in general, we think that for the near future only a "rigorous" approach [Jones 86] is feasible, in which not all design steps are formally proved. For example, the following steps can be done:

- Formalize the requirements for a system.
- Apply design steps on an intuitive basis.
- Formally prove only the "critical" steps of the development process.

Of course, what is considered "critical" is subject to personal opinion and thus error-prone. We believe, however, that even just the first step of requirement formalization catches many errors.

The use of machine supported proof tools is a long-term perspective. While it is not reasonable to expect completely automated theorem proving, for special cases of distributed systems (like protocol entities) it may be possible to provide mechanic and even automatic proof assistance.

Acknowledgement

We would like to thank Manfred Broy, Tobias Nipkow and Ketil Stølen for reading preliminary versions of this paper. This work was supported by the Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen".

Appendix

A.1 Development steps on the trace level

	US	UL	LS	LL	S_1	S_2	S_3	L_1	L_2	L_3	L_4	L_5	L_6
Start	×	×	×	×									
Step 1			×	×	×	×		×	×	×			
Step 2			×	×	×	×	×	×		×	×		
Step 3			×	×	×	×	×			×		×	×

A.2 Proofs of Chapter 3

Proof of Proposition 3.2: We want to show: if $Step_2(t)$ holds, then $L_2(t)$ is true.

$put_T(\langle k,d\rangle)$ *in* t
$\Rightarrow$ $get_T(ack(k))$ *in* t [by L_4]
$\Rightarrow$ $put_R(ack(k))$ *in* t [by LS]
$\Rightarrow$ $\exists\, d' : get_R(\langle k,d'\rangle)$ *in* t [by S_3]
$\Rightarrow$ $get_R(\langle k,d\rangle)$ *in* t [by Lemma 3.4 below]

Lemma 3.4: Assume LS(t) and $S_2(t)$. Then $put_T(\langle k,d\rangle)$ *in* t $\wedge$ $get_R(\langle k,d'\rangle)$ *in* t $\Rightarrow$ d = d'.
Proof of Lemma 3.4:

$put_T(\langle k,d\rangle)$ *in* t
$\Rightarrow$ (Snd © t)[k] = snd(d) (*) [by S_2]

$get_R(\langle k,d'\rangle)$
$\Rightarrow$ $put_T(\langle k,d'\rangle)$ *in* t [by LS]
$\Rightarrow$ (Snd © t)[k] = snd(d') (**) [by S_2]

Thus d = d' with (*) and (**) □

Proof of Proposition 3.3: We want to show: if $Step_3(t)$ holds, then $L_1(t)$ and $L_4(t)$ are true.
Let data(Snd © t) $\sqsupseteq \langle d_0, ...,d_n\rangle$ (***)
First we prove that $get_T(ack(k))$ *in* t $\Rightarrow$ $put_T(\langle k,d_k\rangle)$ *in* t holds:

$get_T(ack(k))$ *in* t

$\Rightarrow$ $put_R(ack(k))$ *in* t [by LS]

$\Rightarrow$ $\exists$ d : $get_R(\langle k,d\rangle)$ *in* t [by S_3]

$\Rightarrow$ $\exists$ d : $put_T(\langle k,d\rangle)$ *in* t [by LS]

$\Rightarrow$ $\exists$ d : $put_T(\langle k,d\rangle)$ *in* t $\wedge$ (Snd © t)[k] = snd(d) [by S_2]

$\Rightarrow$ $put_T(\langle k,d_k\rangle)$ *in* t [by (***)]

Now we show that the case $\forall$ k $\leq$ n : $get_T(ack(k))$ *in* t must hold:

$\neg$ $get_T(ack(k))$ *in* t $\wedge$ k $\leq$ n

$\Rightarrow$ #(Snd © t) > k $\wedge$ $\neg$A(t,k) $\wedge$ #($\surd$©t) = ∞ [by (***), Lemma 3.5, LL]

$\Rightarrow$ $\exists$ j, d : $\neg$ A(t,j) $\wedge$ #($put_T(\langle j,d\rangle)$ © t) = ∞ [by L_5]

$\Rightarrow$ $\exists$ j, d : $\neg$ $get_T(ack(j))$ *in* t $\wedge$ #($put_T(\langle j,d\rangle)$ © t) = ∞ [by Lemma 3.5]

$\Rightarrow$ $\exists$ j,d : $\neg$ $get_T(ack(j))$ *in* t $\wedge$ #($get_R(\langle j,d\rangle)$ © t) = ∞ [by LL]

$\Rightarrow$ $\exists$ j, d : $\neg$ $get_T(ack(j))$ *in* t $\wedge$ #($put_R(ack(j))$ © t) = ∞ [by L_6]

$\Rightarrow$ $\exists$ j, d : $\neg$ $get_T(ack(j))$ *in* t $\wedge$ #($get_T(ack(j))$ © t) = ∞ [by LL]

which is a contradiction.

To be proven: $L_4(t)$ $\equiv$ $put_T(\langle k,d\rangle)$ *in* t $\Rightarrow$ $get_T(ack(k))$ *in* t holds:

$put_T(\langle k,d\rangle)$ *in* t

$\Rightarrow$ (Snd © t)[k] = snd(d) [by S_2]

$\Rightarrow$ $get_T(ack(k))$ *in* t

The last step is due to the fact that the assumption $\neg$ $get_T(ack(k))$ *in* t leads to a contradiction (see above).

Lemma 3.5: If $Step_3(t)$ holds, then A(t,k) $\Leftrightarrow$ $get_T(ack(k))$ *in* t.

Proof of Lemma 3.5: We show that if we assume $Step_3(t)$, then

s $\circ$ $get_T(ack(k))$ $\sqsubseteq$ t $\Rightarrow$ #(Snd © s) > k

holds for all finite partial traces s, which obviously proves the lemma.

s $\circ$ $get_T(ack(k))$ $\sqsubseteq$ t

$\Rightarrow$ $put_R(ack(k))$ *in* s [by LS, def. of *in*]

$\Rightarrow$ $\exists$ d : $get_R(\langle k,d\rangle)$ *in* s [by S_3]

$\Rightarrow$ $\exists$ d : $put_T(\langle k,d\rangle)$ *in* s [by LS]

$\Rightarrow$ (Snd © s)[k] = snd(d) [by S_2]

$\Rightarrow$ #(Snd © s) > k [def. of _[_]] □

A.3 Proofs of Chapter 4

Proof of Proposition 4.1:

"⇒": Let $Func_T(f)$ be true, $x \in I^*$ and $y \in O^*$ such that $y \sqsubseteq f.x$. $t = x \circ f.x$ obviously fulfils $f(I©t) = O©t \wedge (\forall s \sqsubseteq t : O©s \sqsubseteq f(I©s))$. With $Func_T(f)$ we have $\forall s \sqsubseteq t : T'(I©s,O©s)$, hence $T'(I©(x \circ y),O©(x \circ y))$ holds, therefore $T'(x,y)$.

"⇐": Let $t \in Act^\omega$ and assume

(1) $f(I©t) = O©t \wedge \forall s \sqsubseteq t : O©s \sqsubseteq f(I©s)$

(2) $\forall x \in I^*, y \in O^* : y \sqsubseteq f.x \Rightarrow T'(x,y)$

Let $s \in Act^*$ and $s \sqsubseteq t$. Then with (1) we have $O©s \sqsubseteq f(I©s)$, with (2): $T'(I©s,O©s)$. □

Proof of Proposition 4.2:

$$
\begin{aligned}
Func_T(f) &\Leftrightarrow \forall t : ((f(I©t) = O©t \wedge (\forall s \sqsubseteq t : O©s \sqsubseteq f(I©s))) \Rightarrow T'(I©t,O©t)) \\
&\Leftrightarrow \forall t : ((f(I©t) = O©t \wedge (\forall s \sqsubseteq t : O©s \sqsubseteq f(I©s))) \Rightarrow T'(I©t,f(I©t)) \\
&\Leftrightarrow \forall t : T'(I©t,f(I©t)) \qquad (*) \\
&\Leftrightarrow \forall x \in I^\omega : T'(x,f.x) \qquad (**)
\end{aligned}
$$

The transition marked (*) is valid due to the following argument:

"⇒": Let $t \in Act^\omega$. There is a $t' \in Act^\omega$ with $I©t' = I©t$ and
$f(I©t') = O©t' \wedge \forall s \sqsubseteq t' : O©s \sqsubseteq f(I©s)$.

"⇐": Obvious.

The transition marked (**) is valid because for each t there is a x such that $x = I©t$ and vice versa. □

A.4 Proofs of Chapter 5

Proposition 5.10: Take arbitrary $a, i \in \mathbb{N}$, $z \in Info^*$, and let $n \in \mathbb{N}$. Then the following property (*) holds:

$$(\forall n \le i, e : z[n] \neq \langle a,e \rangle) \Rightarrow remove(a,z)[i] = z[i] \wedge$$
$$(\exists n \le i, e : z[n] = \langle a,e \rangle) \Rightarrow remove(a,z)[i] = z[i+1]$$

Proof of Proposition 5.10: By induction on the second argument of remove:

Base case: $z = \langle\rangle$: Here, (*) holds trivially, since neither $remove(a,\langle\rangle)[i]$ nor $\langle\rangle[i]$ are defined (remember that we use "strong equality", which yields true if both sides are undefined).

Inductive case: Let (*) hold for some z. We have to show (*) for $\langle k,d \rangle \circ z$. We will show that the inductive assumption implies the two conjuncts of (*):

a) First conjunct:

$\forall$ i : ($\forall$ n $\leq$ i, e : z[n] $\neq$ ‹a,e›) $\Rightarrow$ remove(a,z)[i] = z[i]

$\Rightarrow$ [strengthening the antecedens and equivalence transformation of the consequence]

$\forall$ i : ($\forall$ n $\leq$ i+1, e : (‹k,d›$\circ$z)[n] $\neq$ ‹a,e›) $\Rightarrow$ (‹k,d›$\circ$remove(a,z))[i+1] = (‹k,d›$\circ$z)[i+1]

$\Leftrightarrow$ [expanding and shifting the range of i, since (‹k,d›$\circ$…)[0] = (‹k,d›$\circ$z)[0]]

$\forall$ i : ($\forall$ n $\leq$ i, e : (‹k,d›$\circ$z)[n] $\neq$ ‹a,e›) $\Rightarrow$ (‹k,d›$\circ$remove(a,z))[i] = (‹k,d›$\circ$z)[i]

$\Leftrightarrow$ [definition of remove (since k $\neq$ a)]

$\forall$ i : ($\forall$ n $\leq$ i, e : (‹k,d›$\circ$z)[n] $\neq$ ‹a,e›) $\Rightarrow$ remove(a,‹k,d›$\circ$z)[i] = (‹k,d›$\circ$z)[i]

b) Second conjunct:

$\forall$ i : ($\exists$ n $\leq$ i, e : z[n] = ‹a,e›) $\Rightarrow$ remove(a,z)[i] = z[i+1]

$\Rightarrow$ [strengthening the antecedens and equivalent transformation of the consequence]

$\forall$ i : k $\neq$ a $\wedge$ ($\exists$ n $\leq$ i, e : z[n] = ‹a,e›) $\Rightarrow$ (‹k,d›$\circ$remove(a,z))[i+1] = (‹k,d›$\circ$z)[i+2]

$\Leftrightarrow$ [adding an always true conjunct and shifting the range of i]

$\forall$ i : k = a $\Rightarrow$ z[i] = (‹k,d›$\circ$z)[i+1] $\wedge$

$\forall$ i : k $\neq$ a $\wedge$ ($\exists$ n $\leq$ i-1, e : z[n] = ‹a,e›) $\Rightarrow$ (‹k,d›$\circ$remove(a,z))[i] = (‹k,d›$\circ$z)[i+1]

$\Leftrightarrow$ [definition of remove (twice)]

$\forall$ i : k = a $\Rightarrow$ remove(a,‹k,d›$\circ$z)[i] = (‹k,d›$\circ$z)[i+1] $\wedge$

$\forall$ i : k $\neq$ a $\wedge$ ($\exists$ n $\leq$ i-1, e : z[n] = ‹a,e›) $\Rightarrow$ remove(a,‹k,d›$\circ$z)[i] = (‹k,d›$\circ$z)[i+1]

$\Leftrightarrow$ [joining the cases]

$\forall$ i : (k = a $\vee$ (k $\neq$ a $\wedge$ ($\exists$ n $\leq$ i-1, e : z[n] = ‹a,e›))) $\Rightarrow$ remove(a,‹k,d›$\circ$z)[i] = (‹k,d›$\circ$z)[i+1]

$\Leftrightarrow$ [predicate calculus (distributive law)]

$\forall$ i : ($\exists$ n $\leq$ i, e : (‹k,d›$\circ$z)[n] = ‹a,e›) $\Rightarrow$ remove(a,‹k,d›$\circ$z)[i] = (‹k,d›$\circ$z)[i+1] □

Proof that the implementation of φ meets specification (1):

Base case: z = $\langle\rangle$: $\varphi_1.\langle\rangle$ = 0 = #(Snd©$\langle\rangle$)

Inductive case: Let (1) hold for some z $\in$ I^*. We have to show (1) for z$\circ$c, where c $\in$ I:

a) c = $\surd$: $\varphi_1.(z \circ \surd)$ = $\varphi_1.z$ = #(Snd©z) = #(Snd©(z $\circ$ $\surd$))

b) c = snd(e): $\varphi_1.(z \circ snd(e))$ = $\varphi_1.z$ + 1 = #(Snd©z) + 1 = #(Snd©(z $\circ$ snd(e)))

c) c = ack(a): $\varphi_1.(z \circ ack(a))$ = $\varphi_1.z$ = #(Snd©z) = #(Snd©(z $\circ$ ack(a)))

Proof that the implementation of φ meets specifications (2) and (3):

Note that (3) implies (2) for any z $\in$ I^*. It is therefore sufficient to show (3) by induction. We may, however, use both (2) and (3) as our induction assumptions.

Base case: z = $\langle\rangle$: ($\varphi_2.\langle\rangle$)[i] = ‹k,d› $\Leftrightarrow$ false $\Leftrightarrow$ (Snd©$\langle\rangle$)[k] = snd(d)

Inductive case: Let (3) hold for some $z \in I^*$. We have to show (3) for $z \circ c$, where $c \in I$:

a) $c = \surd$:

$\varphi_2(z \circ \surd)[i] = \langle k,d \rangle$

$\Leftrightarrow$ [definition of φ_2]

$(\varphi_2.z)[i] = \langle k,d \rangle$

$\Leftrightarrow$ [induction assumption]

$(Snd©z)[k] = snd(d) \wedge \neg A(z,k) \wedge |\{ j < k : \neg A(z,j) \}| = i$

$\Leftrightarrow$ [predicate calculus]

$(Snd©(z \circ \surd))[k] = snd(d) \wedge \neg A(z \circ \surd, k) \wedge |\{ j < k : \neg A(z \circ \surd, j) \}| = i$

b) $c = snd(e)$:

$\varphi_2(z \circ snd(e))[i] = \langle k,d \rangle$

$\Leftrightarrow$ [definition of φ_2]

$(\varphi_2.z \circ \langle \varphi_1.z, e \rangle)[i] = \langle k,d \rangle$

$\Leftrightarrow$ [case splitting]

$((\varphi_2.z)[i] = \langle k,d \rangle) \vee (\#(\varphi_2.z) = i \wedge \varphi_1.z = k \wedge e = d)$

$\Leftrightarrow$ [(1), (2) and (3)]

$((Snd©z)[k] = snd(d) \wedge \neg A(z,k) \wedge |\{ j < k : \neg A(z,j) \}| = i)$

$\vee (|\{ j < \#(Snd©z) : \neg A(z,j) \}| = i \wedge \#(Snd©z) = k \wedge e = d)$

$\Leftrightarrow$ [Setting $k = \#(Snd©z)$ in second line and definition of A]

$((Snd©z)[k] = snd(d) \wedge \neg A(z,k) \wedge |\{ j < k : \neg A(z,j) \}| = i)$

$\vee (|\{ j < k : \neg A(z,j) \}| = i \wedge \#(Snd©z) = k \wedge e = d \wedge \neg A(z,k))$

$\Leftrightarrow$ [distributive law]

$\neg A(z,k) \wedge |\{ j < k : \neg A(z,j) \}| = i \wedge (((Snd©z)[k] = snd(d)) \vee (\#(Snd©z) = k \wedge e = d))$

$\Leftrightarrow$ [predicate calculus (since $A(z,k)$ and $A(z \circ snd(e),k)$ are equivalent)]

$(Snd©(z \circ snd(e)))[k] = snd(d) \wedge \neg A(z \circ snd(e), k) \wedge |\{ j < k : \neg A(z \circ snd(e), j) \}| = i$

c) $c = ack(a)$:

$\varphi_2(z \circ ack(a))[i] = \langle k,d \rangle$

$\Leftrightarrow$ [definition of φ_2]

$remove(a, \varphi_2.z)[i] = \langle k,d \rangle$

$\Leftrightarrow$ [prop. 5.10 and predicate calculus, since $(A \Rightarrow B) \wedge (\neg A \Rightarrow C) = (A \wedge B) \vee (\neg A \wedge C)$]

$((\varphi_2.z)[i] = \langle k,d \rangle \wedge \forall\, n \le i, e : (\varphi_2.z)[n] \neq \langle a,e \rangle)$

$\vee ((\varphi_2.z)[i+1] = \langle k,d \rangle \wedge \exists\, n \le i, e : (\varphi_2.z)[n] = \langle a,e \rangle)$

$\Leftrightarrow$ [… see next page]

$\Leftrightarrow$ [by four applications of the induction assumption (3)]

$((Snd©z)[k] = snd(d) \wedge \neg A(z,k) \wedge |\{ j < k : \neg A(z,j) \}| = i$

$\quad \wedge \forall n \le i, e : (Snd©z)[a] \neq snd(e) \vee A(z,a) \vee |\{ j < a : \neg A(z,j) \}| \neq n)$

$\vee ((Snd©z)[k] = snd(d) \wedge \neg A(z,k) \wedge |\{ j < k : \neg A(z,j) \}| = i+1$

$\quad \wedge \exists n \le i, e : (Snd©z)[a] = snd(e) \wedge \neg A(z,a) \wedge |\{ j < a : \neg A(z,j) \}| = n)$

$\Leftrightarrow$ [predicate calculus]

$((Snd©z)[k] = snd(d) \wedge \neg A(z,k) \wedge |\{ j < k : \neg A(z,j) \}| = i$

$\quad \wedge \#(Snd©z) \le a \vee A(z,a) \vee |\{ j < a : \neg A(z,j) \}| > i)$

$\vee ((Snd©z)[k] = snd(d) \wedge \neg A(z,k) \wedge |\{ j < k : \neg A(z,j) \}| = i+1$

$\quad \wedge \#(Snd©z) > a \wedge \neg A(z,a) \wedge |\{ j < a : \neg A(z,j) \}| \le i)$

$\Leftrightarrow$ [to understand this step, please take a piece of paper and carefully trace every subterm]

$((Snd©z)[k] = snd(d) \wedge \neg A(z,k) \wedge a \neq k \wedge (A(z,a) \vee a \ge k) \wedge |\{ j < k : \neg A(z,j) \}| = i)$

$\vee ((Snd©z)[k] = snd(d) \wedge \neg A(z,k) \wedge \neg A(z,a) \wedge a < k \wedge |\{ j < k : \neg A(z,j) \}| = i+1)$

$\Leftrightarrow$ [case combination and definition of A (again, this is quite a complex step)]

$(Snd©(z \circ ack(a)))[k] = snd(d) \wedge \neg A(z \circ ack(a), k) \wedge |\{ j < k : \neg A(z \circ ack(a), j) \}| = i$ □

References

[Alpern, Schneider 85] B. Alpern, F.B. Schneider: Defining liveness. Information Processing Letters, Vol. 21, 1985, pp. 181-185.

[Bochmann 90] G. v. Bochmann: Protocol specification for OSI. Computer Networks and ISDN Systems 18, 1989/90, pp. 167-184.

[Broy 87] M. Broy: Some algebraic and functional hocus pocus with Abracadabra. Bericht MIP-8717, Universität Passau 1987.

[Broy 91] M. Broy: Towards a Formal Foundation of the Specification and Description Language SDL. Formal Aspects of Computing (1991) 3: 21–57

[Broy et al. 91] M. Broy, F. Dederichs, C. Dendorfer, R. Weber: Characterizing the behaviour of reactive systems by trace sets. SFB-Bericht Nr. 342/2/91 A, Technische Universität München 1991.

[Broy et al. 92] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. Gritzner, R. Weber: The design of distributed systems – an introduction to FOCUS. SFB-Bericht Nr. 342/2/92 A, Technische Universität München 1992.

[Chandy, Misra 88] K.M. Chandy, J. Misra: Parallel Program Design. Addison-Wesley 1988.

[Chu, Liu 88] P.M. Chu, M.T. Liu: Synthesizing protocol specifications from service specifications in the FSM model. In: Proc. Computer Networking Symposium, IEEE 1988, pp. 173-182.

[Dederichs 92] F. Dederichs: Transformation verteilter Systeme: Von applikativen zu prozeduralen Darstellungen. Doctoral Dissertation, Technische Universität München 1992.

[Gotzhein, Bochmann 90] R. Gotzhein, G. v. Bochmann: Deriving protocol specifications from service specifications including parameters. ACM TOPLAS, Vol. 8, 1991, pp. 255-283.

[Jonsson 90] B. Jonsson: A hierarchy of compositional models of I/O-automata. In: B. Rovan (editor): Mathematical Foundations of Computer Science. LNCS 452, Springer-Verlag 1990, pp. 347-354.

[Li, Maibaum 88] D.-H. Li, T.S.E. Maibaum: A top-down step-wise refinement methodology for protocol specification. In: F.H. Vogt (editor): Concurrency 88. LNCS 335, Springer-Verlag 1988, pp. 197-221.

[Olderog 90] E.-R. Olderog: From trace specifications to process terms. In: J.W. de Bakker et al. (editors): Stepwise Refinement of Distributed Systems. LNCS 430, Springer-Verlag 1990, pp. 592-621.

[Pnueli 86] A. Pnueli: Applications of temporal logic to the specification and verification of reactive systems: A Survey of Current Trends. In: J.W. de Bakker et al. (editors): Current Trends in Concurrency. LNCS 224, Springer-Verlag 1986, pp. 510-584.

[Probert, Saleh 91] R.L. Probert, K. Saleh: Synthesis of communication protocols: survey and assessment. IEEE Transactions on Computers, Vol. 40, 1991, pp. 468-475.

[Saleh, Probert 90] K. Saleh, R.L. Probert: A service-based method for the synthesis of communications protocols. Int. J. Mini and Microcomputers. Special Issue on Distributed Systems, Vol. 12, 1990, pp. 97-103.

[Stenning 76] V. Stenning: A data transfer protocol. Computer Networks 1, 1976, pp. 98-110.