Specification and Refinement of a Buffer of Length One*

Manfred Broy
Institut für Informatik
Technische Universität München
D–80290 München, Germany

April 26, 1995

Abstract

We illustrate the use of functional system specifications and their refinement in the development of system components by a simple case study. The development includes the modular specification, refinement, and verification of system components. We start the development with an informal description of the tasks of the case study and then step by step carry out the tasks formally. The informal requirement specification can be used as starting point for alternative formalization and development techniques. The emphasis in this study is laid on the modelling of a system at different levels of abstraction and the verification conditions obtained by the refinement relations between these versions. We show in particular, how we can refine a time independent component into a component that uses timeouts and thus depends on the timing of the input in an essential way.

*This work is supported by the Sonderforschungsbereich 342 "Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen"

Contents

1 Preliminary Remarks

For the development of reliable distributed reactive systems we need methods for their specification, refinement, and verification. In the literature we find numerous suggested formalisms for describing reactive systems. Prominent examples are

- state transition descriptions complemented by various versions of logical calculi such as assertion logics, temporal logics and several combinations thereof,
- Petri nets as a graphical formalisms for describing distributed state transition systems including explicit concurrency,
- process algebras in various instances, such as CCS or CSP, together with their operational semantics given by transition systems, axiomatic equations and semantic models.

All these approaches have advantages and drawbacks. The best way to obtain insights and criteria how useful these different approaches are in practice, are comparisons of development case studies.

In this paper we propose such a small case study consisting of a number of representative development steps. We then demonstrate how functional system development techniques can deal with the proposed problem.

The small example of the stepwise development of a buffer of length one, as described informally in section 2, is used to demonstrate functional formalisms for the specification, refinement, and verification of system components. The emphasis is put on the usefulness of functional formalisms in the development process and on the achieved modularity of the system descriptions and the development process.

We start with an informal description of the tasks. Then we give formal solutions to each of the tasks using functional system specifications, refinement and verification techniques.

We suggest this example also as a test case for other methods for the specification, development, and verification of system components.

2 Informal Description of the Components

In this section we give a brief informal description of the components that we will specify.

A *one element buffer* is a component that can store one data element and return it upon request.

A *fair loose one element buffer* is a component that can store one data element and return it upon request, however, it may fail in storing data elements or in serving requests. It indicates success or failure by a positive or negative

acknowledgement. It is fair in the sense that it will not fail forever on repeated attempts.

A *driver* is a component that when composed with a fair loose buffer forms a system that behaves like a one element buffer.

A *real time one element buffer* is a one element buffer that operates in a discrete time frame. It may take some time until it is ready to take the next input. It indicates by interaction, when it is ready to take the next input.

A *real time fair loose one element buffer* is a component that operates in a real time frame; it can store one data element and return it upon request, however, it may fail in storing a data element or in serving a request. It indicates success by a positive acknowledgement within a fixed amount of time. It is fair in the sense that it will not fail forever on repeated attempts.

A *real time driver* is a component that operates in a real time frame; when composed with a real time loose buffer it forms a system that behaves like a real time one element buffer.

3 Specification and Verification Tasks

The following development, refinement,[1] and verification tasks are carried out in the sequel:

(1) Specification of a one element buffer.

(2) Specification of a fair loose one element buffer.

(3) Specification of a driver.

(4) Verification that the composition of the driver and the fair loose one element buffer is an implementation (refinement) of the one element buffer.

(5) Specification of a real time one element buffer.

(6) Verification that (5) is a refinement of (1).

(7) Specification of a real time fair loose one element buffer.

(8) Verification that (7) is a refinement of (2).

(9) Specification of a real time driver.

(10) Verification that (9) is a refinement of (3).

(11) Verification that the composition of the real time driver and the real time loose one element buffer is an implementation (refinement) of the real time one element buffer.

[1] We do not explain the notion refinement here, since we want to leave freedom for formalizing it. Throughout the treatment of the case study we use a very particular concept of refinement, which will be introduced in the appendix.

(12) Verification that (11) can be concluded from (6), (8), and (10).

(13) Discussion how easily the development may be adapted to modified requirements such as a k–element buffer or modified refinements based on other versions of unreliability of the buffer.

In the following sections we carry out these tasks. We use functional techniques for system specification, refinement and verification. We introduce and explain these techniques through the development process. A short introduction to functional specification techniques is given in the appendix.

4 Specification of a One Element Buffer

In this section we give specifications of a one element buffer using and demonstrating three specification styles.

4.1 Informal Description and Syntactic Interface

A one element buffer is an interactive component with one input line and one output line. It may store at most one data element. It receives input messages which are either data elements or requests (represented by the signal $\odot$). As long as the buffer never gets a request signal when it is empty and as long as it never gets a data message when it is full then it behaves properly like a one element buffer.

The behavior of the buffer is formalized by stream processing functions. A short introduction of the concept of streams and stream processing functions is given in the appendix.

Let D be a set of data elements. We use the set M of input messages for the buffer which is defined as follows:

$$M = D \cup \{\odot\}$$

The element $\odot$ is used as a signal indicating a request for the element stored in the buffer. We represent the behavior of a one element buffer by stream processing functions:

$$f : M^\omega \to D^\omega$$

The domain and range of this function determine the syntactic interface of the buffer. The set of possible behaviors f of a one element buffer is specified by the proposition:

$$B.f$$

where B is a predicate:

$$B : (M^\omega \to D^\omega) \to \mathbb{B}$$

The predicate B characterizes the set of behaviors[2] of the one element buffer represented by stream processing functions.

4.2 Equational Specifications

The buffer is a component with a behavior described as follows. If the buffer is empty and it receives a data message, it stores the data message. If the buffer is full and it receives a request signal, it sends its data message on its output line. Mathematically the predicate B can be specified by the weakest predicate that fulfills the following equation:

$$B.f \equiv \forall d \in D : f.\langle d \rangle = \langle\rangle \wedge \forall x \in M^\omega : \exists \tilde{f} : B.\tilde{f} \wedge f(d^\frown \odot^\frown x) = d^\frown \tilde{f}.x$$

One may wonder why we do not write simply

$$f(d^\frown \odot^\frown x) = d^\frown f.x$$

instead of the more complicated formula

$$\exists \tilde{f} : B.\tilde{f} \ \wedge \ f(d^\frown \odot^\frown x) = d^\frown \tilde{f}.x$$

The reason is as follows. The simple equation for f would include the requirement

$$f(d^\frown \odot^\frown x) = d^\frown f.x$$

also for input streams x that do not fulfill the assumption that data are sent only if the buffer is not full and requests are sent only if the buffer is not empty. The equation includes the requirement that the unspecified behavior on such improper input coincides for the input $d^\frown \odot^\frown x$ and x after d has been produced as output. This overspecification is avoided by the formula above.

Note on the specification style: This way we have given a "recursive" or equational definition of the predicate B by an equation of the form

$$(*) \quad B.f = \ldots \ \exists \tilde{f} : B.\tilde{f} \wedge \ \ldots$$

Since the expression $B.\tilde{f}$ occurs in positive form on the righthand side of the defining equation, the logical expression on the righthand side of the equation represents an operation that is monotonic in B with respect to logical implication. Therefore there exists a weakest and a strongest solution of the defining equation (corresponding to the weakest and strongest fixpoints of the predicate transformer represented by the righthand side of the defining equation). We say that a predicate occurs in positive form in an expression, if it occurs syntactically only under an even number of negation signs. Since the equation $(*)$ does

[2] If this set of behaviours contains more than one element then we speak of *underspecification*.

not contain any negation signs the expression $B.\tilde{f}$ trivially occurs in positive form.

We choose the weakest solution of the equation $(*)$ for the predicate B, since this corresponds to the way we use this style of specifications. This way of specifying B gives immediately a proof principle. We abbreviate the equation $(*)$ for the specifying predicate by the equation

$$B \equiv T[B]$$

T denotes the predicate transformer defined by the righthand side of the equation. According to our definition the predicate B is required to be the weakest predicate that fulfills the equation $(*)$. According to this characterization we obtain the following logical principle for every predicate C we have:

$$(C \Rightarrow T[C]) \Rightarrow (C \Rightarrow B)$$

This formula expresses that every predicate that is a fixpoint of T is not weaker than the predicate B. It provides a proof principle for the specifying predicates defined by recursive equations.

To keep our formulas readable we sometimes use the following abbreviation in equations: the equation

$$t_1 = t_2[[\![B]\!]/\tilde{f}]$$

stands for (let $\tilde{f}$ be an identifier that does not occur in the term t_1)

$$\exists \tilde{f} : t_1 = t_2 \wedge B.\tilde{f}$$

This abbreviation allows to avoid the existential quantifier. It leads to the following shorter syntactic form of the specification of the predicate B:

$$B.f \equiv \forall d \in D : f.\langle d \rangle = \langle\rangle \wedge \forall x \in M^\omega : f(d^\frown \odot^\frown x) = d^\frown [\![B]\!].x$$

This notation can help to keep the formulas short and more readable.

End of note on the specification style.

There are many styles to write functional specifications. In the following we give a number of specifications of the predicate B written in specific other specification styles.

4.3 Assumption/Commitment Specification Format

Another possibility to write a specifying formula for the predicate B is given by the assumption/commitment format (cf. [Broy 94], [Stølen et al. 92]). For such specifications we need to find an assumption predicate

$$A : M^\omega \rightarrow \mathbb{B}$$

and a commitment predicate

$$C : M^\omega \times D^\omega \to \mathbb{B}$$

The assumption predicate formalizes the constraints about the input histories that have to be fulfilled in order to guarantee a proper behavior of the buffer. The commitment predicate formalizes the notion of a proper behavior.

In the case of the one element buffer these predicates are specified by the following equations (for $x \in M^\omega, d \in D, y \in D^\omega$):

$$\begin{aligned}
A.\langle\rangle &\equiv \text{true} \\
A(\odot^\frown x) &\equiv \text{false} \\
A(d^\frown x) &\equiv \big(x = \langle\rangle \vee (ft.x = \odot \wedge A.rt.x)\big) \\
C(\langle\rangle, y) &\equiv (y = \langle\rangle) \\
C(\langle d\rangle, y) &\equiv (y = \langle\rangle) \\
C(d^\frown \odot^\frown x, y) &\equiv \big(ft.y = d \wedge C(x, rt.y)\big)
\end{aligned}$$

Both predicates A and C are assumed to be the weakest predicates that fulfill the defining equations. The assumption expresses that proper input for the one element buffer consists of a stream that is empty or starts with a data element followed by a request signal (or by the empty stream) and continues this way by successively carrying a data element and a request signal.

The commitment $C(x, y)$ expresses for a stream x (that fulfills the assumption) the stream y contains as many elements as the stream x contains request signals and these elements are the data elements in the sufficiently large prefix of x. Another way to specify the commitment predicate C would be to use the following equation:

$$C(x, y) \equiv \#\{\odot\}©x = \#y \wedge y \sqsubseteq D©x$$

This yields a stronger predicate C, which, however, due to the restriction of the input stream x by the assumption, works equivalently for the assumption/commitment scheme.

Similarly we can specifiy the assumption predicate explicitly by the following equation

$$A(x) \equiv \forall \tilde{x} : \tilde{x} \sqsubseteq x \Rightarrow \#\{\odot\}©\tilde{x} \leq \#D©\tilde{x} \leq 1 + \#\{\odot\}©\tilde{x}$$

In this simple case of an assumption predicate we can also use a regular expression for specifying A:

$$A.x \equiv (\{\tilde{x} \in M^* : \tilde{x} \sqsubseteq x\} \subseteq \{D\odot\}^*\{D\})$$

Using the predicates A and C we define the specifying predicate B by the following formula:

$$B.f \equiv \forall x \in M^\omega : A.x \Rightarrow C(x, f.x)$$

When using this structure of a specification we speak of an *assumption/commitment specification format.*

4.4 State–Based Specifications

A third possibility for the formalization of the one element buffer is a specification that refers to local states of the component. We introduce a set of states:

$$State = D \cup \{\emptyset\}$$

For this set we define a stream processing function for every element of the set $State$ by the predicate

$$H : \big[State \to [M^\omega \to D^\omega]\big] \to I\!B$$

Formally the predicate H is specified as follows:

$$\begin{aligned} H.h \equiv \quad \forall \sigma \in State, d \in D : \exists \tilde{h} : \quad & H.\tilde{h} \wedge \\ & (h.\sigma).\langle\rangle = \langle\rangle \wedge \\ & (h.\emptyset)(d^\frown x) = (\tilde{h}.d).x \wedge \\ & (h.d)(\odot^\frown x) = d^\frown(\tilde{h}.\emptyset).x \end{aligned}$$

Using the predicate H we can define the specifying predicate B by the following formula:

$$B.f \equiv \exists h : H.h \wedge f = h.\emptyset$$

We can also give a slightly different state-oriented specification where the specification (the predicate) is parameterized by the state; mathematically expressed, we use a predicate

$$\hat{H} : State \to ([M^\omega \to D^\omega] \to I\!B)$$

By the parameterized predicate $\hat{H}$ we can associate a specification with every state by the following formula

$$(\hat{H}.\sigma).f \equiv \exists h : H.h \wedge f = h.\sigma$$

The formula also relates the parameterized predicate $\hat{H}$ and the predicate H.

4.5 Short Discussion of the Specifications

All three specifications of the predicate B are logically equivalent. We do not give proofs for the equivalence of these specifications. The proofs are rather straightforward by induction on the length of the input streams and therefore left to the reader.

The specifications leave unspecified what happens if the buffer gets a request when it is empty or a data message when it is full. The behavior of the buffer is underspecified. There is an infinite set of functions that fulfill the predicate B.

Given the specification B by the defining equation $(*)$ we can prove a number of simple properties about one element buffers, such as the following formula:

$$B.f \Rightarrow f.\langle\rangle = \langle\rangle \wedge f(d^\frown\odot) = \langle d\rangle.$$

The proof is straightforward by the monotonicity of f (we base the proof on the first specification of B):

Assume $B.f$. We use that f is continous. We obtain the following proof (let $d \in D$):

$\langle\rangle \sqsubseteq \langle d\rangle$	by the definition of prefix ordering,
$f.\langle\rangle \sqsubseteq f.\langle d\rangle$	by the monotonicity of f,
$f.\langle\rangle \sqsubseteq \langle\rangle$	by $B.f$ since $f.\langle d\rangle = \langle\rangle$
$f.\langle\rangle = \langle\rangle$,	by the definition of the prefix ordering ($\langle\rangle$ is least element).

The second part of the conclusion is proved as follows (let $d \in D$):

$f(d^\frown\odot^\frown\langle\rangle) =$	by $B.f$
$d^\frown\tilde{f}.\langle\rangle =$	by the lemma above (where $B.\tilde{f}$)
$\langle d\rangle$	

These two simple examples of proofs show two basic proof concepts for functional specifications based on the monotonicity assumption for the specified functions and equational reasoning.

5 Specification of a Loose One Element Buffer

A one element loose buffer is a component with one input line and two output lines called data output channel and acknowledgement channel. It may store at most one data element. It receives input messages which are either data elements or requests (represented by the signal $\odot$). If the buffer never gets a request signal when it is empty and never gets a data message when it is full then it behaves properly like a one element buffer, but it may lose data messages and request signals. It may lose a data message that is sent to it when it is in an empty state, but then this loss is indicated by the signal $\ominus$ on its acknowledgement channel; if it stores its data message correctly this is indicated by the acknowledgement signal $\oplus$ on its acknowledgement channel. It may also refuse to answer properly to a request signal, but also this is indicated by the signal $\ominus$ on its acknowledgement channel; if it sends its data message correctly this is indicated by the acknowledgement signal $\oplus$ on its acknowledgement channel. We assume that the loose buffer is fair in the sense that it reacts by an acknowledgement signal eventually if the message transmission is tried sufficiently often.

We define the following message sets:

$$M = D \cup \{\odot\}$$

$$N = \{\oplus, \ominus\}$$

We represent the behaviors of a loose buffer by functions:

$$f : M^\omega \to (D^\omega \times N^\omega)$$

The domain and the range of this function fix the syntactic interface of the loose buffer. We use a predicate

$$P : \big(M^\omega \to (D^\omega \times N^\omega)\big) \to I\!B$$

to specify the behavior of a loose buffer. The specification of the predicate P reads as follows:

$$\begin{array}{lll} P.f \equiv \forall x \in M^\omega, d \in D : & \exists n, m \in I\!N : & \forall i \in I\!N : \\ & i \le n \Rightarrow & f(d^i) = [\langle\rangle, \ominus^i] \wedge \\ & & f(d^{n+1}) = [\langle\rangle, \ominus^n \frown \oplus] \wedge \\ & i \le m \Rightarrow & f(d^{n+1} \frown \odot^i) = [\langle\rangle, \ominus^n \frown \oplus \frown \ominus^i] \wedge \exists \tilde{f} : P.\tilde{f} \wedge \\ & & f(d^{n+1} \frown \odot^{m+1} \frown x) = [d, \ominus^n \frown \oplus \frown \ominus^m \frown \oplus] \frown \tilde{f}.x \end{array}$$

In the formula above we write m^i for the stream consisting of i copies of the message m.

The specification of the loose buffer includes both liveness properties (every input is answered and the buffer is fair in the sense that it eventually responds to proper input with a positive acknowledgement) and safety properties (the elements that are produced as output are those received as input). This is also the case for the specification of the one element buffer in the previous chapter, however, there the liveness conditions were less involved.

Another possibility to specify a loose buffer is obtained by using states and prophecy variables. In a functional specification we can use state concepts by introducing a set of states. With each state we associate a behavior represented by a stream processing function. A prophecy can be formally understood just as part of the state. However, it is used not to record the past of the input as far as it is relevant for the future behavior of a component, but it is used as an oracle for the nondeterministic decisions of the component and helps to express the fairness of the component. In our example we use natural numbers as prophecies.

Let the set of states of the fair loose buffer be specified as follows:

$$State = D \cup \{\emptyset\}$$

We define the auxiliary predicate (the set $I\!N$ of natural numbers is used to represent prophecies)

$$Q : \big[I\!N \times State \to (M^\omega \to D^\omega \times N^\omega)\big] \to I\!B$$

by the following formula (let Q again be the weakest predicate that fulfills the formula):

$$\begin{array}{lllll} Q.h & \equiv & \forall d \in D, x \in M^* : & \exists \tilde{h} : Q.\tilde{h} \wedge & \\ & & \exists n \in I\!N : & h(0,\emptyset).(d^\frown x) & = & [\langle\rangle, \oplus]^\frown \tilde{h}(n,d).x \wedge \\ & & \exists n \in I\!N : & h(0,d).(\odot^\frown x) & = & [d, \oplus]^\frown \tilde{h}(n,\emptyset).x \wedge \\ & & \forall n \in I\!N : & h(n+1,\emptyset).(d^\frown x) & = & [\langle\rangle, \ominus]^\frown \tilde{h}(n,\emptyset).x \wedge \\ & & & h(n+1,d).(\odot^\frown x) & = & [\langle\rangle, \ominus]^\frown \tilde{h}(n,d).x \end{array}$$

In this specification for the function application $h(n,s).x$ where $Q.h$ holds the number n is a prophecy variable which determines the number of failing attempts until an input is accepted and $s \in State$ represents the state of the buffer. We define the predicate $\tilde{P}$ specifying the loose buffer with the help of the predicate Q as follows:

$$\tilde{P}.f \;\equiv\; \exists h, n : Q.h \;\wedge\; f = h(n,\emptyset)$$

The specification of the predicate Q is written carefully such that the choice of the prophecy variable n may depend on the input stream x. So every time the first parameter of h is 0 another number can be chosen.

The specification covers the behavior of the buffer for infinite streams x by the continuity property. It is not difficult to prove that:

$$\tilde{P}.f \Rightarrow P.f$$

The other direction of this implication does not hold, however, since the predicate $\tilde{P}$ is (unnecessarily) stronger than the predicate P in a subtle way. From $P.f$ where $f(\langle d\rangle) = [\langle\rangle, \ominus]$ we cannot conclude anything about $f(d_1^\frown d_2)$ with $d_1 \neq d_2$. For the function $h(n,\emptyset)$ we know that there exists a number n such that $h(n,\emptyset).(d_0^\frown \ldots^\frown d_n) = [\langle\rangle, \ominus^{n\frown} \oplus]^\frown \tilde{h}(\tilde{n}, d_n)$ with $Q.\tilde{h}$. A more liberal specification for Q can be given, of course, by using a more elaborate state space in which we can record which data element is tried to be submitted. But such a more liberal specification is less suggestive, when writing state-oriented specifications.

The proof that the specification $\tilde{P}$ is a refinement of the specification P is straightforward by induction on the length of the input streams.

An assumption/commitment specification for the loose buffer is certainly more delicate, since we cannot give a characterization for input histories that fulfill the assumption without referring to the output history. Whether the buffer is empty or full can only be determined by looking both at the input and the output history. This problem typically arises for assumption/commitment specifications of nondeterministic components.

We demonstrate how to write an assumption/commitment specification even in these cases using prophecies (following [Stølen et al. 92]). To do this we introduce a filter function

$$filter : M^\omega \times N^\infty \to M^\omega$$

specified by (let $x \in M^\omega, y \in N^\infty$):

$$\begin{array}{lcl} filter(x, \oplus^\frown y) & = & ft.x^\frown filter(rt.x, y) \\ filter(x, \ominus^\frown y) & = & filter(rt.x, y) \end{array}$$

Based on this filter function we define the assumption and the commitment predicates for which we add prophecy parameters.

$$\begin{array}{ll} \tilde{A}: & M^\omega \times N^\infty \to I\!B \\ \tilde{C}: & M^\omega \times (D^\omega \times N^\omega) \times N^\infty \to I\!B \end{array}$$

The predicates are specified by (for $x \in M^\omega, y \in D^\omega, z \in N^\omega, r \in N^\infty$):

$$\begin{array}{lcl} \tilde{A}(x,r) & \equiv & A\big(filter(x,r)\big) \\ \tilde{C}\big(x,(y,z),r\big) & \equiv & \big(z \sqsubseteq r \Rightarrow C\big(filter(x,r),y\big) \wedge \\ & & (\#z = \infty \Rightarrow \#\{\oplus\}©z = \infty)\big) \end{array}$$

where the predicates A and C are defined as in the section on the assumption/commitment specification for the one element buffer B. Based on this specification we can specifiy the predicate P by a more general assumption/commitment format:

$$P.f \equiv \forall r \in N^\infty, x \in M^\omega : \tilde{A}(x,r) \Rightarrow \tilde{C}(x, f.x, r)$$

This assumption/commitment specification, however, is much more involved. It is not very helpful, since it is rather difficult to understand.

6 Specification of a Driver

In this section we specifiy a component called a driver that can be used to construct a nonloose buffer from the loose buffer by composing it with the driver into a network.

A driver is a component that has two input lines and one output line. On one input line, called its message line, it gets a stream of messages which are to be transferred and on the other input line, called its control line, it gets positive and negative acknowledgements represented by the signals $\oplus$ and $\ominus$. It repeats the sending of a message on negative acknowledgements until it receives a positive acknowledgement.

A driver is a component with behaviors represented formally by functions of the functionality:

$$g : M^\omega \times N^\omega \to M^\omega$$

The behavior of the driver is specified by the predicate

$$V : (M^\omega \times N^\omega \to M^\omega) \to I\!B$$

as follows:

$$V.g \equiv \quad \forall a \in M, x \in M^\omega, y \in N^\omega : \\ g(a^\frown x, y) = a^\frown \textbf{if } ft.y = \oplus \textbf{ then } g(x, rt.y) \textbf{ else } g(a^\frown x, rt.y) \textbf{ fi}$$

Again this specification is written carefully taking into account the following informal requirement: a driver sends a first copy of a message on its message input channel independent of the question whether a positive or negative acknowledgement is available. Only after the first copy of the message has been sent the first signal in the acknowledgement stream is inspected. If this signal is positive, then the transmission of the message a is finished and the transmission of the next element in the message stream is started. If the signal is negative, then the transmission of a is repeated.

By the assumption of monotonicity for the function g we can prove the following formula

$$g(\langle\rangle, y) = \langle\rangle$$

under the (weak) additional assumption that the set M contains at least two elements. The proof is carried out by contradiction. Assume

$$g(\langle\rangle, y) \neq \langle\rangle$$

Then for some message $b \in M$

$$g(\langle\rangle, y) = b^\frown ...$$

However, for $a \neq b$ we have

$$g(a^\frown ..., y) = a^\frown ...$$

which yields the contradiction.

7 Specification of a System Composed of the Loose Buffer and the Driver

In this section we study a network that is obtained by composing the loose buffer with the driver. This networks behaves like a one-element buffers. We prove this at the end of this section.

We specify a system component C that is composed of the driver and of the one element loose buffer. A graphical representation of the component C in terms of a data flow network is given in Figure 1.

The behavior of the component represented by the network is characterized by the predicate

$$C : (M^\omega \to D^\omega) \to I\!B$$

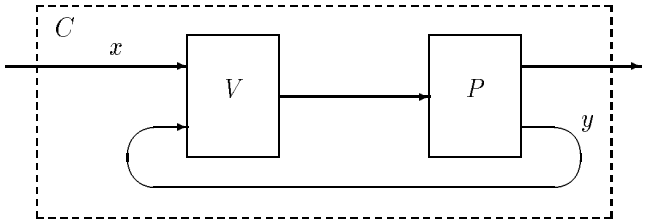

Figure 1: Graphical representation of the network for component C

which is defined as follows:

$$C.f \equiv \exists q, g : P.q \ \wedge \ V.g \ \wedge \ \forall x : \exists y : [f.x, y] = q.g(x, y)$$

The predicate is obtained by a straightforward translation of the network given in Figure 1 into a logical formula.

Now we can envisage a proof that the component specified by the predicate C according to the structure of the network behaves like a reliable one element buffer as specified by the predicate B. Formally the verification condition for the refinement relation is expressed by the following proposition:

$$\forall f : C.f \Rightarrow B.f$$

The proof of this proposition is performed by unfolding the specifying predicate C. Then a proof by induction on the length of the input stream x can be carried out. We outline this proof in the following.

According to the specification B to show that the proposition $C.f \Rightarrow B.f$ holds we have to prove $C \Rightarrow T[C]$ as explained in section 4.2, since B is the weakest predicate that fulfills its defining equation for B given in Section 4.2. This leads to the following proposition.

$$C.f \Rightarrow \quad \forall d \in D : f.\langle d \rangle = \langle\rangle \ \wedge \ \exists \tilde{f} : C.\tilde{f} \ \wedge \ f(d^\frown \odot^\frown x) = d^\frown \tilde{f}.x$$

By definition from the proposition $C.f$ we can conclude that there exist functions q and g for which $P.q$ and $V.g$ hold such that for all streams x there exists a stream y such that the following equation holds:

$$[f.x, y] = q.g(x, y)$$

For $x = \langle d \rangle$ we obtain by induction that there exists a number $n \in I\!N$, such that for all i, if $i \leq n$, there exists a stream $\hat{y}$ such that

$$[f.x, y] = q.g(x, y) \wedge y = \ominus^{i\frown} \hat{y}$$

For $i = n + 1$ we obtain

$$[f.x, y] = [\langle\rangle, \ominus^{n\frown} \oplus]^\frown q.g(\langle\rangle, \hat{y}) \wedge y = \ominus^{n\frown} \oplus^\frown \hat{y}$$

and since $g(\langle\rangle, y) = \langle\rangle$ and $q.\langle\rangle = [\langle\rangle, \langle\rangle]$ we obtain

$$f.\langle d\rangle = \langle\rangle$$

For $x = d^\frown\odot^\frown\hat{x}$ we obtain by the same type of reasoning that there exist numbers $n, m \in I\!N$ and a stream $\hat{y}$ such that

$$[f.x, y] = [\langle d\rangle, \ominus^n{}^\frown\oplus^\frown\ominus^m{}^\frown\oplus]^\frown q.g(\hat{x}, \hat{y}) \wedge y = \ominus^n{}^\frown\oplus^\frown\ominus^m{}^\frown\oplus^\frown\hat{y}$$

We obtain from these equations that there exist streams z and $\hat{y}$ such that the following formula holds:

$$f.x = d^\frown z \wedge [z, y] = q.g(\hat{x}, \hat{y})$$

By induction on the length of the stream x we obtain that there exists a function $\tilde{f}$ such that

$$f.x = d^\frown\tilde{f}.\hat{x} \text{ where } C.\tilde{f}$$

for finite streams x. By the continuity of the involved functions this result extends to infinite streams, too. This concludes the proof.

In the following sections we refine both the driver and the unreliable buffer such that they work within a framework of a discrete time.

8 Modelling Time

When developing interactive systems with behaviors that depend on the timing of input messages we need to have system models in which time is explicitly represented. We use a very simple discrete model of time which is sufficient for our purposes. We think about time as a sequence of time intervals of constant length. The timing of the messages of a stream is given by indicating in which time interval which of its messages appear.

8.1 Time Signals

In our functional framework we model the progress of time by special messages called *time signals* in the input and output histories. We use the symbol $\surd$, called a *time tick*, to indicate each end of a time interval in a stream.

To model the behavior of a component in discrete time we use stream processing functions where all input and output streams carry time ticks. For a tuple of streams

$$x \in ((M_1 \cup \{\surd\})^\omega \times ... \times (M_m \cup \{\surd\})^\omega)$$

with time ticks we write

$$\#\{\surd\}\textcircled{c}x \quad \text{for} \quad min(\#\{\surd\}\textcircled{c}x_1, ..., \#\{\surd\}\textcircled{c}x_n)$$

By $\#\{\surd\}©x$ we denote thus the number of time intervals with complete communication information on all n streams. We write

$$\overline{x} \text{ for } (M_1©x_1, ..., M_n©x_n).$$

The stream $\overline{x}$ denotes the tuple of streams where all information about the timing is eliminated and only the proper messages are kept.

8.2 Timed Stream Processing Functions

A timed stream processing function takes streams containing time ticks as input and produces streams containing time ticks as output. The specific properties of time are taken care of by the so-called time progress property that is specified below.

Given a timed stream-processing function:

$$f : (M_1 \cup \{\surd\})^\omega \times ... \times (M_n \cup \{\surd\})^\omega \rightarrow (N_1 \cup \{\surd\})^\omega \times ... \times (N_m \cup \{\surd\})^\omega$$

we specify the predicate $TIME$ that expresses the basic *time progress property* of stream processing functions f as follows:

$$TIME.f \equiv \forall x \in (M_1 \cup \{\surd\})^\omega \times ... \times (M_n \cup \{\surd\})^\omega : \#\{\surd\}©f.x = \#\{\surd\}©x$$

The predicate $TIME.f$ expresses the fact that for the function f its output history for k time units is determined by every input history that fixes the input on all input lines for k time units. This is a general assumption that implies that a component cannot predict its input and time cannot go backwards.

8.3 Specifying Time Distances and Time Delays

When specifiying the behavior of time dependent components we often want to express that two successive messages do not arrive within a very short time distance. To be able to express this in specifications we introduce a function

$$dist : \left(M \cup \{\surd\}\right)^\omega \rightarrow I\!N \cup \{\infty\}$$

which yields the minimal time distance (least number of time ticks) between two successive messages that are not time ticks in a stream x. Mathematically, the function $dist$ is specified by the following equation:

$$dist(x) = min\{\#t : t \in \{\surd\}^* \wedge \exists a, b \in M, z \in \left(M \cup \{\surd\}\right)^* : z^\frown a^\frown t^\frown b \sqsubseteq x \vee z^\frown a^\frown t = x\}$$

With the help of this function it is not difficult to specifiy for a given number $c \in I\!N$ functions that introduce time ticks into a stream such that the time distance is at least c.

We specify for every number $c \in \mathbb{N}$ the predicate

$$H_c : \left(M^\omega \to (M \cup \{\surd\})^\omega\right) \to \mathbb{B}$$

which characterizes delay functions that add an arbitrary number of time ticks to a stream such that the time distance is at least c. This is formally expressed by the following equation:

$$H_c(f) \equiv \forall x \in M^\omega : dist(f.x) \geq c \ \wedge \ x = \overline{f.x}$$

The introduced functions and predicates are used in the following. They allow a very compact notation of formulas specifying time dependent components.

9 Specification of a Timed One Element Buffer

A timed one element buffer is a component with one input line and one output line. It may store at most one data element. It receives input messages which are either data elements or requests (represented by the signal $\odot$). If the buffer never gets a request signal when it is empty and never gets a data message when it is full then it behaves properly like a one element buffer provided the time distance between input signals in its input stream is large enough.

We define the sets of messages that appear in the input and output streams of the component as follows:

$$\hat{M} = D \cup \{\odot, \surd\}$$

$$\hat{D} = D \cup \{\surd\}$$

We represent the behaviors of a real time buffer by timed stream processing functions:

$$f : \hat{M}^\omega \to \hat{D}^\omega$$

The set of possible behaviors of a real time buffer is specified by the following proposition:

$$\hat{B}.f$$

where $\hat{B}$ is a predicate:

$$\hat{B} : (\hat{M}^\omega \to \hat{D}^\omega) \to \mathbb{B}$$

We assume that there exists some given number $e \in \mathbb{N}$ which is the required time distance between the messages sent to the buffer. The predicate $\hat{B}$ is specified as follows:

$$\begin{aligned}
\hat{B}.f \quad \equiv \quad & TIME.f \wedge \\
& \forall x \in \hat{M}^\omega : \exists \tilde{f} : \hat{B}.\tilde{f} \wedge \overline{f(\surd^\frown x)} = \overline{\tilde{f}.x} \\
& \forall i, j \in \mathbb{N} : \\
& \overline{f(d^\frown \surd^i)} = \langle\rangle \wedge \\
& i \geq e \ \wedge j \geq e \Rightarrow \overline{f(d^\frown \surd^{i\frown} \odot^\frown \surd^{j\frown} x)} = d^\frown \overline{\tilde{f}.x}
\end{aligned}$$

In this specification we carefully kept in mind the monotonicity requirement when expressing the fact that for an input stream

$$d^\frown\surd^{i\frown}\odot^\frown\surd^{j\frown}x$$

we are guaranteed an output stream that contains d as its first data element, provided i and j are sufficiently large. This data element is not produced as output before the request signal $\odot$ has been received. The monotonicity reflects the causality between input and output.

In this specification we did not give more properties about the timing of the output than needed. From this specification we can prove a number of further simple properties about timed one element buffers. We restrict our attention to the relationship between the specifications B and $\hat{B}$.

The real time one element buffer is a refinement in the sense of [Broy 92] of the one element buffer. This is expressed by the following theorem:

$$\hat{B}.\hat{f} \ \wedge \ \left(\forall x \in \hat{M}^\omega : dist(x) \geq e \Rightarrow \ f.\overline{x} = \overline{\hat{f}.x}\right) \Rightarrow B.f$$

The proof is straightforward by induction on the length of x. The proof is in particular easy, since the formula defining the specification $\hat{B}$ is very similar to the one used for describing the predicate B.

Using ";" for sequential composition and a predicate A for the time abstraction specification which is defined as follows

$$A.f \equiv \forall x : f.x = \overline{x}$$

we obtain the following theorem

$$H_e; \hat{B}; A \Rightarrow B$$

This is the classical refinement condition as formulated in [Broy 93]. Here the predicate H_e serves as the representation specification which characterizes the set of functions that insert at least e time ticks between two successive messages in its input streams. The predicate A serves as the abstraction function that maps timed streams onto untimed streams.

10 Specification of a Timed Loose One Element Buffer

A real time loose one element buffer is a component with one input line and two output lines. It can store at most one data element. It receives input messages which are either data elements or requests (represented by the signal $\odot$). If the buffer never gets a request signal when it is empty and never gets a data message when it is full then it behaves properly like a one element buffer,

but it may lose data messages and request signals. It may lose a data message that is sent to it in an empty state, but then this loss is indicated by the fact that a positive acknowledgement is not sent within c units of time; if the loose buffer stores its data message correctly this is indicated by the signal $\oplus$ on its acknowledgement channel. It may also refuse to respond to a request signal, but also this is indicated by the fact that no answer is produced within c units of time; if it sends its data message correctly this is indicated by the signal $\oplus$ that is sent after at most c units of time.

The real time loose buffer is very similar to the loose buffer that we described above, however, it does not send negative acknowledgements. The lack of negative acknowledgement can be recognized, however, by the property that the real time loose buffer sends a positive acknowledgement, if at all, at least after c time units. We define the following message sets:

$$\hat{M} = D \cup \{\odot, \checkmark\}$$

$$\hat{N} = \{\oplus, \checkmark\}$$

We represent behaviors of a real time loose one element buffer by functions:

$$f : \hat{M}^\omega \to (\hat{D}^\omega \times \hat{N}^\omega)$$

The set of correct behaviors of a real time loose buffer is specified by the predicate:

$$\hat{P} : (\hat{M}^\omega \to (\hat{D}^\omega \times \hat{N}^\omega)) \to \mathbb{B}$$

The predicate $\hat{P}$ is specified as follows. We assume the time constant $c \in \mathbb{N}$. The component behaves as follows provided its input messages are at least in time distance c. If the component is in the empty state and it receives a data message, it either stores the data message and acknowledges this by sending the signal $\oplus$ within c units of time or it loses the data message. If the component is full and it receives a request signal, it either sends its data message on its output line and acknowledges this by sending the signal $\oplus$ within c units of time or it refuses to carry out the request.

In both cases the component reacts to an input message either by a positive acknowledgement within c units of time or it does not accept the message and does not react to it. So, if a reaction to a message is not received within c units of time, it is clear that the message has not been accepted. However, the loose buffer sends a positive acknowledgement after at most b attempts of transmission where $b \in \mathbb{N}$ is a given constant.

These properties in the behavior of the loose buffer are expressed by the

following equation for the predicate $\hat{P}$:

$$\begin{aligned}\hat{P}.f \quad \equiv \quad & TIME.f \ \wedge \ \forall x, y \in \hat{M}^{\omega}, d \in D : dist(x) \geq c \Rightarrow \exists \tilde{f} : \hat{P}.\tilde{f} \wedge \\ & \exists i \in [1:b] : \overline{x} = d^{i-1} \Rightarrow \overline{f(x)} = [\langle\rangle, \langle\rangle] \wedge \\ & \overline{x} = d^{i} \Rightarrow \overline{f(x)} = [\langle\rangle, \oplus] \wedge \\ & \exists j \in [1:b] : \overline{x} = d^{i} \frown \odot^{j} \Rightarrow \overline{f(x \frown y)} = [d, \oplus \frown \oplus] \frown \overline{\tilde{f}(y)}\end{aligned}$$

We consider the specification $\hat{P}$ as a refinement of the specification P, but the notion of refinement we use here is more involved since now we have represented the negative acknowlegements that we use for the loose buffer specified by P on the level of real time systems by a lack of a reply in a certain amount of time. Nevertheless $\hat{P}$ can be understood as a refinement of P.

Let $n \in I\!N$ be a given constant. To express the refinement relation explicitly we specify the set of functions that turn the lack of positive acknowledgements into negative acknowledgements by the predicate:

$$G_e : \left(\hat{N}^{\omega} \times \hat{M}^{\omega} \rightarrow N^{\omega}\right) \rightarrow I\!B$$

It is specified by the following formula:

$$\begin{aligned}G_n.f \equiv \quad & \forall \ i \in I\!N, a \in M : \forall \ x \in \hat{M}^{\omega}, y \in \hat{N}^{\omega} : \\ & f(\surd \frown y, \surd \frown x) = f(y, x) \wedge \\ & i > n \Rightarrow f(\surd^{i} \frown y, a \frown \surd^{i} \frown x) = \ominus \frown f(y, x) \wedge \\ & i \leq n \Rightarrow f(\surd^{i} \frown \oplus \frown y, a \frown \surd^{i} \frown x) = \oplus \frown f(y, x)\end{aligned}$$

With the help of the specification G_n we can interpret the specification $\hat{P}$ as a refinement of the specification P. In particular we have

$$H_c; \Upsilon; (\hat{P} \parallel I); (A \parallel G_c) \Rightarrow P$$

where the component I denotes the identity function and the component Υ denotes the function that generates two copies of its input. In mathematical terms:

$$I.x = x$$

$$\Upsilon.x = (x, x)$$

This is again a refinement along the lines of [Broy 92]. Figure 2 gives a graphical representation of the refinement of P by $\hat{P}$ together with the representation and the abstraction specification. Formally, Figure 2 describes a refinement of the component

$$P \parallel I$$

In a more readable version along the lines of Figure 2 the formula above reads as follows. Let $h, \hat{h}$ and $\hat{f}$ be functions such that $G_c.\hat{h}, \hat{P}.\hat{f}$ and $H_c.h$, then from

$$\forall \ x : f.x = (\bar{z}, \hat{h}(y, h.x)) \text{ where } (z, y) = \hat{f}.h.x$$

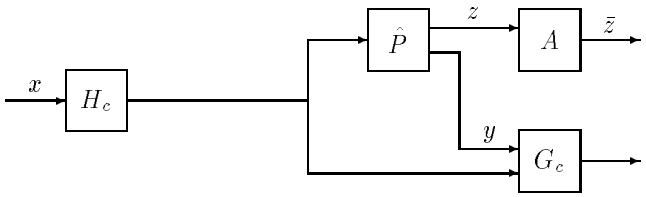

Figure 2: Graphical representation of the refinement of P

we can conclude $P.f$.

In terms of [Broy 92] the predicate H_c defines the representation specification and $A \parallel G_c$ the abstraction specification.

11 Specification of a Real Time Driver

A real time driver has two input lines and one output line. It receives on one line data messages and on the other line acknowledgements. Each message that it receives is sent after some time on its output line. Then it observes its second input line for a certain amount of time. If it does not receive an acknowledgement in this amount of time it repeats the data message, otherwise it deals with the next data message.

A real time driver is a component with behaviors represented by stream processing functions:

$$g : \hat{M}^\omega \times \hat{N}^\omega \to \hat{M}^\omega$$

The domain and range of these functions determine the syntactic interface of the driver. The behavior of a real time driver is specified by the predicate

$$\hat{V} : (\hat{M}^\omega \times \hat{N}^\omega \to \hat{M}^\omega) \to I\!B$$

as follows:

$$\begin{aligned} \hat{V}.g \equiv \quad & \forall a \in M, y \in \hat{N}^*, x \in \hat{M}^* : \exists k \in I\!N : \forall i \in I\!N : TIME.g \wedge \\ & g(\surd^i \frown a \frown x, \surd^i \frown y) = \surd^i \frown a \frown \surd \frown catch(a, x, y, k + c) \end{aligned}$$

where the function

$$catch : D \times \hat{D}^\omega \times \hat{N}^\omega \times I\!N \to \hat{D}^\omega$$

is used to express that the transmission of the message a is retried if a positiove acknowlegement is not received within $k + c$ time intervals. The function g is

specified as follows:

$$\begin{array}{l}\forall a \in M, y \in \hat{N}^\omega, x \in \hat{M}^\omega, n \in I\!N : \\ \overline{catch(a, \surd^\frown x, \surd^\frown y, n+1)} = \overline{catch(a, x, y, n)} \\ \overline{catch(a, x, y, 0)} = \overline{g(a^\frown x, y)} \\ \overline{catch(a, x, \oplus^\frown \surd^\frown y, n+1)} = \overline{g(x, y)}\end{array}$$

The driver sends repeatedly a message received on its data input line in time distance c until it gets a positive acknowledgement, provided a new message on its data input line arrives only after the positive acknowledgement has been received on the acknowledgement input channel.

Again the real time driver can be seen as a refinement of the driver that is not sensitive to time as specified by the predicate V above.

The predicate

$$\gamma : \hat{M}^\omega \times \hat{N}^\omega \to I\!B$$

specifies the properties of pairs of input streams that we require for a driver for its proper functioning. Let $i < c$ hold.

$$\begin{array}{lcl}\gamma(\surd^\frown x, \surd^\frown y) & = & \gamma(x, y) \\ \gamma(a^\frown \surd^{i\frown} x, \surd^{i\frown} \oplus^\frown y) & = & \gamma(x, y) \\ \gamma(a^\frown \surd^{c\frown} x, \surd^{c\frown} y) & = & \gamma(x, y) \\ \gamma(\surd^\frown x, \oplus^\frown y) & = & \text{false}\end{array}$$

Let γ be the weakest predicate that fulfills these equations.

Now we define abstraction functions by the specification

$$A_V : (\hat{M}^\omega \times \hat{N}^\omega \to M^\omega \times N^\omega) \to I\!B$$

where A_V is described by the following formula

$$\begin{array}{ll}A_V.\alpha \equiv & \forall x \in \hat{M}^\omega, y \in \hat{N}^\omega, i \in I\!N, a \in M : i < c \Rightarrow \\ & \alpha(\surd^\frown x, \surd^\frown y) = \alpha(x, y) \\ & \alpha(a^\frown \surd^{c\frown} x, \surd^{c\frown} y) = [\langle\rangle, \ominus]^\frown \alpha(x, y) \\ & \alpha(a^\frown \surd^{i\frown} x, \surd^{i\frown} \oplus^\frown y) = [a, \oplus]^\frown \alpha(x, y)\end{array}$$

We define a representation specification

$$R_V : (M^\omega \times N^\omega \to \hat{M}^\omega \times \hat{N}^\omega) \to I\!B$$

by

$$R_V.\rho = \forall \alpha : A_V(\alpha) \Rightarrow \rho; \alpha = I$$

With this definition we obtain the theorem (where A denotes the abstraction specification defined in section 9 that specifies the function that eliminates all time ticks in a stream):

$$R_V; \hat{V}; A \Rightarrow V$$

In other words the specification $\hat{V}$ is a refinement of the specification V according to the representation specification R_V and the abstraction specification A.

In particular we have for every function h with $G_c.h$ and every function $\hat{f}$ with $\hat{V}.\hat{f}$:

$$\big(\forall\ x, y : \gamma(x, y) \Rightarrow f(\bar{x}, h(y, x)) = \bar{z}\big) \Rightarrow V.f$$

where $z = \hat{f}(x, y)$. This formula indicates that the specification $\hat{V}$ can be understood as a refinement of the driver specification V.

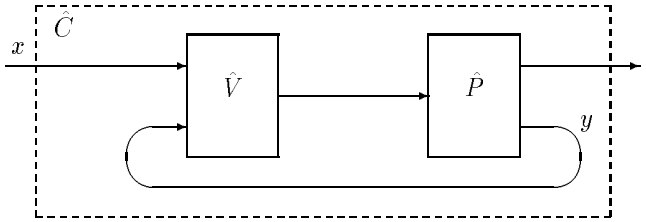

Figure 3: Graphical representation of $\hat{C}$

12 Specification of a System Composed From the Timed Loose Buffer and the Driver

We specify a system by a data flow network composed of the real time driver and of the timed loose one element buffer and associate with it a predicate $\hat{C}$. A graphical representation of the network corresponding to the component $\hat{C}$ is given in Figure 3.

The logical specification of the predicate $\hat{C}$ is described by following formula:

$$\hat{C}.f \equiv \exists q, g : \hat{P}.q\ \wedge\ \hat{V}.g\ \wedge\ \forall x : \exists y : [f.x, y] = q.g(x, y)$$

The formula is just a translation of the data flow network given in Figure 3 into equations.

Based on the logical specification we can prove that the component specified by $\hat{C}$ behaves like a reliable timed one element buffer provided $b * c < e$. Recall that e denotes the time distance between messages required for the one-element buffer, c was the time distance of the messages between the driver and the loose buffer. The loose buffer was supposed to accept a message after at most b attempts. From now on we assume $b * c < e$.

Formally the verification condition for the refinement relation is expressed by the following proposition.

$$\forall f : \hat{C}.f \Rightarrow \hat{B}.f$$

Again the proof of this proposition can be done by unfolding the predicate $\hat{C}$ and doing a proof by induction on the length of the stream x.

However, we might also be interested just to prove that the component specification $\hat{C}$ is a refinement of the component specification B. Mathematically expressed:

$$H_e; \hat{C}; A \Rightarrow B$$

This can be proved by unfolding $\hat{C}$. We obtain the following verification condition. Let $\hat{f}$ be a function where $\hat{C}.\hat{f}$ holds. Let $\hat{q}, \hat{g}$ be functions such that $\hat{P}.\hat{q}$ and $\hat{V}.\hat{g}$. Assume furthermore for all streams x that there exists a stream y such that

$$[\hat{f}.x, y] = \hat{q}.\hat{g}(x, y)$$

We define functions g and q by the logical formula

$$\forall x : g(\bar{x}, h(z, y)) = \bar{z} \text{ where } z = \hat{g}(x, y)$$

and by the formula

$$\forall x : q.x = (\bar{z}, \hat{h}(x, y)) \text{ where } (z, y) = \hat{q}.h.z \text{ and } G_c(\hat{h})$$

We obtain the equation:

$$[\hat{f}.x, y] = \hat{q}.\hat{g}(x, y)$$

with $z = \hat{g}(x, y)$ and the equation:

$$[\overline{\hat{f}.x}, h(z, y)] = q.g(\bar{x}, h(z, y))$$

So the function $\bar{\hat{f}}$ fulfills the specification of component C; mathematically expressed we have

$$H_e; \hat{C}; A \Rightarrow C$$

and since we already have shown in Section 7 that C is a refinement of B, mathematically expressed

$$C \Rightarrow B$$

we obtain, that $\hat{C}$ is a refinement of B.
The relationship between C and $\hat{C}$ is shown in Figure 4. Since we also have proved above the following formula:

$$H_n; \hat{B}; A \Rightarrow B$$

by the fact

$$\hat{C} \Rightarrow \hat{B}$$

and by the monotonicity of sequential composition with respect to the predicate $\hat{C}$ we can conclude the validity of the following formula

$$H_n; \hat{C}; A \Rightarrow H_n; \hat{B}; A$$

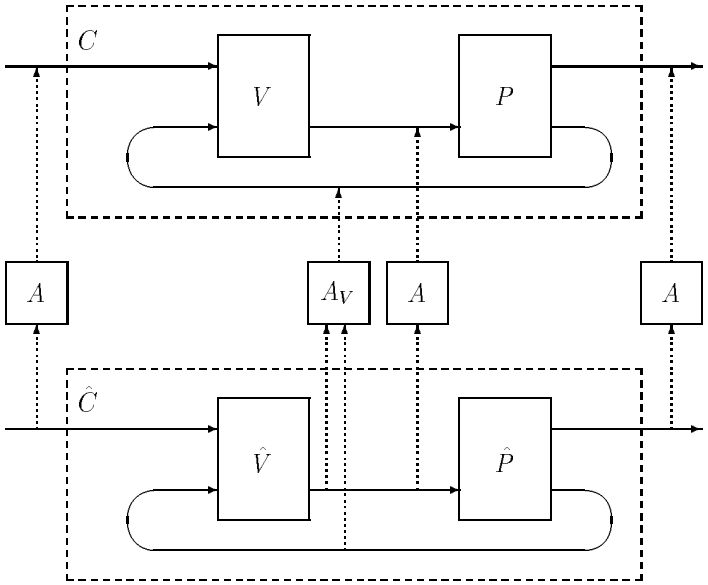

Figure 4: Refinement relation between C and $\hat{C}$ illustrated by the abstraction specifications[4]

By transitivity of implication we obtain from the formula above the following refinement property:

$$H_n; \hat{C}; A \Rightarrow B$$

This concludes the proof of the required properties and our development.

13 Adaption to Modified Requirements

In practice the requirements often are modified and adapted to the changing needs of an application. In these cases it is important how easy it is to adapt the specifications, refinements and proofs to the modified requirements.

In our case a typical example would be to treat an n–element buffer instead of the one element buffer. We claim that our specifications and refinements can be adapted to this case with acceptable overhead.

Another example of a different branch of development is obtained, if we consider a real time buffer that does not work with a constant time distance, but indicates eventually by a signal that it is prepared to take the next input. Then more sophisticated notions of refinement are needed. The representation

[4]Here we work only with abstraction specifications.

specification that translates the non timed input into timed input has to depend also on the output produced by the refined component, since time ticks have to be inserted into the input until the component is prepared for further input. This needs more sophisticated refinement concepts such as refining contexts as described in the appendix.

14 Conclusion

It is the purpose of this paper to demonstrate the flexibility and usefulness of functional system specification, verification and refinement techniques by a small but nevertheless intricate example. We would appreciate very much similar demonstrations by other researchers using other formal techniques for the specification, refinement, and verification of reactive systems for the same example.

Of course from a more practical point of view it might not be necessary to go in detail through so many levels of abstraction as worked out above. Nevertheless we computing scientists should be able in principle to describe such refinements precisely. A formal method for the development of distributed systems should support all steps in the development in a flexible way such that we are able to express all aspects of intermediate design steps.

Acknowledgements:

I gratefully acknowledge helpful comments by Ursula Hinkel, Katharina Spies and Max Fuchs as well as a number of stimulating discussions with Ketil Stølen.

A Appendix: Concepts of Specification

In this section we give a brief summary of the basic mathematical concepts of functional system models. We consider system components with a finite number of input and output channels. Over the channels messages are exchanged. A channel history is mathematically modelled by a stream of messages. The behavior of a (deterministic) component corresponds to a function mapping the streams on its input channels onto streams for its output channels.

A *stream* of messages over a given message set M is a finite or infinite sequence of messages. We define

$$M^{\omega} =_{def} M^{*} \cup M^{\infty}$$

by $x^\frown y$ we denote the result of concatenating two streams x and y. We assume that $x^\frown y = x$, if x is infinite. By $\langle\rangle$ we denote the empty stream. For simplicity we write for $a \in M, x \in M^\omega$

$$a^\frown x \text{ instead of } \langle a\rangle^\frown x$$

$$x^\frown a \text{ instead of } x^\frown\langle a\rangle$$

If a stream x is a *prefix* of a stream y, we write $x \sqsubseteq y$. The relation $\sqsubseteq$ is called *prefix order*. It is formally specified by

$$x \sqsubseteq y =_{def} \exists z \in M^\omega : x^\frown z = y$$

The behavior of deterministic interactive systems with n input channels and m output channels is modelled by functions

$$f : (M^\omega)^n \to (M^\omega)^m$$

called *(m, n)-ary stream processing functions*. We denote function application $f(x)$ often by $f.x$ to avoid brackets. A stream processing function is called *prefix monotonic*, if for all tuples of streams $x, y \in (M^\omega)^n$ we have

$$x \sqsubseteq y \Rightarrow f.x \sqsubseteq f.y$$

A stream processing function f is called *continuous*, if f is monotonic and for every directed set $S \subseteq M^\omega$ we have:

$$f.\sqcup S = \sqcup\{f.x : x \in S\}$$

By $\sqcup S$ we denote a least upper bound of a set S, if it exists. A set S is called *directed*, if for any pair of elements x and y in S there exists an upper bound in S. The set of streams is complete in the sense that for every directed set of streams there exists a least upper bound.

The set of all prefix continuous stream processing functions of functionality $(M^\omega)^n \to (M^\omega)^m$ is denoted by

$$SPF^n_m$$

For simplicity we do not consider type information here and assume just M to be a set of messages.

By $SPEC^n_m$ we denote the set of all predicates Q where

$$Q : SPF^n_m \to \mathbb{B}$$

The set $SPEC^n_m$ denotes the set of all component specifications for a component with n input channels and m output channels.

The following functions on streams are used in specifications:

$rt : M^\omega \to M^\omega$	rest of a stream
$ft : M^\omega \to M \cup \{\bot\}$	first element of a stream
$\# : M^\omega \to I\!N \cup \{\infty\}$	length of a stream
$\copyright : \wp(M) \times M^\omega \to M^\omega$	filter of a stream

These functions are easily specified by the following equations (let $x \in M^\omega, m \in M, S \in \wp(M)$):

$$
\begin{array}{ll}
rt.\langle\rangle = \langle\rangle, & rt(m^\frown x) = x, \\
ft.\langle\rangle = \bot, & ft(m^\frown x) = m, \\
\#\langle\rangle = 0, & \#(m^\frown x) = 1 + \#x, \\
S\copyright\langle\rangle = \langle\rangle, & \\
S\copyright(m^\frown x) = m^\frown(S\copyright x), & \text{if } m \in S \\
S\copyright(m^\frown x) = S\copyright x, & \text{if } m \notin S
\end{array}
$$

These axioms specify the functions completely. They are useful in proofs, too.

We use two forms of composition: parallel composition and sequential composition.

Given functions

$$f \in SPF^n_k, g \in SPF^k_m$$

we write

$$f; g$$

for the *sequential composition* of the functions f and g which yields a function in SPF^n_m where

$$(f; g).x = g(f(x))$$

Given functions

$$f \in SPF^{n1}_{m1}, g \in SPF^{n2}_{m2}$$

we write

$$f \| g$$

for the *parallel composition* of the functions f and g which yields a function in SPF^{n1+n2}_{m1+m2} where (let $x \in (M^\omega)^{n1}, y \in (M^\omega)^{n2}$):

$$(f \| g).(x, y) = (f.x, g.y)$$

We assume that ";" has higher precedence than "||".

We want to compose specifications of components to networks. Each form of composition introduced for functions can be extended to component specifications in a straightforward way. Given component specifications

$$Q \in SPEC^n_k, R \in SPEC^k_m$$

we write

$$Q;R$$

for the predicate in $SPEC_m^n$ where

$$(Q;R).f \Leftrightarrow \exists q,r : Q.q \wedge R.r \wedge f = q;r$$

Trivially we have for all specifications $Q \in SPEC_m^n$ the following equations:

$$Q;I = Q$$

$$I;Q = Q$$

Given specifications

$$Q \in SPEC_{m1}^{n1}, R \in SPEC_{m2}^{n2}$$

we write

$$Q \| R$$

for the predicate in $SPEC_{m1+m2}^{n1+n2}$ where

$$(Q\|R).f \Leftrightarrow \exists q,r : Q.q \wedge R.r \wedge f = q\|r$$

A specification $\tilde{Q} \in SPEC_m^n$ is called a property refinement of a specification $Q \in SPEC_m^n$ if for all functions f we have $\tilde{Q}.f \Rightarrow Q.f$. We write then

$$\tilde{Q} \Rightarrow Q$$

More sophisticated notions of refinement are obtained by abstraction and representation specifications as described in [Broy 92].

A pair of specifications A and R are called abstraction and representation, if

$$R;A = I$$

where I denotes the identity function. Let A_1 be an abstraction specification and R_2 be a representation specification. The specification $\hat{C}$ is called a refinement of component C if we have

$$\hat{C} \Rightarrow A_1;C;R_2$$

Given the corresponding abstraction specification A_2 and a representation specification R_1 we obtain from

$$\begin{aligned} R_1;A_1 &= I \\ R_2;A_2 &= I \end{aligned}$$

from this

$$R_1;\hat{C};A_2 \Rightarrow C$$

A more general notion of refinement is obtained for components by so-called refining contexts. Figure 5 shows a graphical representation of a refining context.

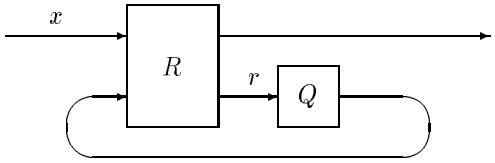

Figure 5: Refining Context R

We write

$$R[Q]$$

for the specification defined by

$$R[Q].f \equiv \exists \rho, q : R.\rho \ \wedge \ Q.q \ \wedge \ \forall x : \exists r : \rho(x, q.r) = (r, f.x)$$

For a refining context R we require an abstraction context A such that for any specification Q of appropriate syntactic interface the network given in Figure 6 is identical to Q.

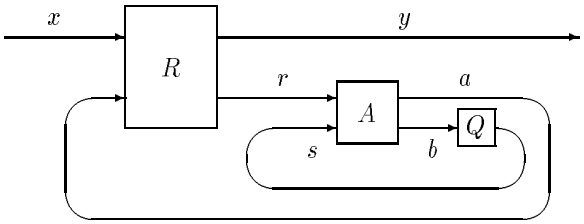

Figure 6: A network identical to Q

In mathematical terms we require for every function f of appropriate functionality

$$\begin{array}{ll} R.\rho \wedge A.\alpha \Rightarrow & \forall x, y, a, b, r, s : \\ & (y, r) = \rho(x, a) \wedge \\ & (a, b) = \alpha(r, s) \wedge \\ & s = f.b \Rightarrow f(x) = s \end{array}$$

This requirement basically means that R and A have inverse effects. In other terms we have

$$R\big[A[Q]\big] \equiv Q$$

for all specifications Q.

References

[Abadi, Lamport 90] M. Abadi, L. Lamport: Composing Specifications. Digital Systems Research Center, SRC Report 66, October 1990

[deBakker et al. 90] J. W. de Bakker, W.-P. de Roever, G. Rozenberg (eds): Stepwise Refinement of Distributed Systems. Lecture Notes in Computer Science 430, Springer 1990

[Broy 90] M. Broy: Functional Specification of Time Sensitive Communicating Systems. REX Workshop. In: [deBakker et al. 90], 153-179

[Broy 92] M. Broy: Compositional Refinement of Interactive System. DIGITAL Systems Research Center, SCR Report 89, July 1992

[Broy 93] M. Broy: Interaction Refinement – The Easy Way. In: M. Broy (ed.): Programm Design Calculi. Springer NATO ASI Series, Series F: Computer and System Sciences, Vol. 118, 1993

[Broy 94] M. Broy: A Functional Rephrasing of the Assumption/Commitment Specification Style. Technische Universität München, Fakultät für Informatik, Technical Report, TUM-I9417, June 1994

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

[Lamport 83] L. Lamport: Specifying concurrent program modules. ACM Toplas 5:2, April 1983, 190-222

[Stølen et al. 92] K. Stølen, F. Dederichs, R. Weber: Assumption/Commitment Rules for Networks of Agents. Technische Universität München, Institut für Informatik, SFB–Bericht Nr. 342/2/93A