Advanced Component Interface Specification[1)]

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

Abstract

We introduce a method for the specification of reactive asynchronous components with a concurrent access interface and outline its mathematical foundation. The method supports the specification of components that show a complex reactive behavior including timing aspects. Examples are the nonstrict fair merge or the arbiter. The method supports the specification of reactive systems and their modular composition into data flow networks. The specification approach is compositional. It supports the integrated specification and verification of both safety and liveness conditions in modular system descriptions. We outline particular specification styles that may be useful for the better readability of such specifications.

1. Introduction

A descriptive functional semantic model of distributed systems of interacting components is of major interest in many research areas and applications of computing science and systems engineering. For the modular systematic development of systems we need precise and readable interface descriptions of system components. We require that such interface descriptions contain all informations about its syntactic and semantic properties needed in order to use it properly. In the interface specification we also describe the time dependency of a component.

Although ignored in theoretical computer science for a while, the incorporation of time and its formal representation is of essential interest for system models. In time dependent

[1)] This work was partially sponsored by the Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen" and the industrial research project SysLab.

systems, the timing and the data values of the output depend upon the timing and the data values of the input. However, for certain components the timing of the input does not influence the data values of the output, but only their timing. Then we can describe the input/output behavior of a component without explicit reference to time.

We are interested in the description of components that react by output to input. Both input and output takes place within a frame of time. When dealing with the time properties of systems we distinguish three basic classes of components characterized by their time dependencies:

- *time independence:* the data output values of a component do not dependent on the timing of the input data but only on their values,
- *weak time dependency*: some of the nondeterministic decisions in the behavior of a nondeterministic component is controlled by the timing of the input such as the relative order of messages on different input channels or the relative order of messages on input and output channels. But this is done in a superficial way that does not allow control of the nondeterministic decisions in the behavior (the output) by the quantitative timing of the input,
- *strong time dependency:* the behavior (the timing as well as the data output) depends on the quantitative timing of the input data.

The class of weak time dependent components brings specific problems for their functional specification. Examples are fair nonstrict merge or the arbiter (see section 4.2 and 6.3 for a detailed treatment). This has already been observed in [Park 83].

The modular specification of the observable behavior of interactive systems is an important technique in system and software development. We speak of *black box specification* or *interface specification*. An adequate concept of interface specification does not only depend on a simple notion of observability, but also on the operators that we apply to compose components into systems.

In the following we introduce a semantic model of interface behavior and study composition operators. On the basis of this semantic model we introduce more pragmatic specification techniques for the description of reactive components.

The paper is divided into a more theoretical and a more practical part. In the theoretical part we introduce and discuss the mathematical foundations. In the more practical part we give examples of component specifications and specification styles.

In a first chapter we introduce our mathematical basis. Then we show how to describe the syntactic interfaces and the dynamic behaviors of interactive systems. We treat composition operators. Finally, we briefly demonstrate different specification styles by some examples.

2. Streams

In this section we introduce the basic mathematical concepts for the description of systems by functional techniques. We define the set of streams over a given set of messages. We introduce a number of functions on this set. Finally we define the notation of stream processing function.

In the following we concentrate on interactive systems that communicate asynchronously through channels. We use streams to denote histories of communications on channels. Given a set M of messages a stream over the set M is a finite or infinite sequence of elements from M. By M^* we denote the finite sequences over the set M. The set M^* includes the empty sequence that we denote by ‹›.

By the set M^∞ we denote the infinite sequences over the set M. The set M^∞ can be understood to be represented by the total mappings from the natural numbers $\mathbb{N}$ into M. We denote the set of all streams over the set M by M^ω. Formally we have

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

We introduce a number of functions on streams that are useful in system descriptions.

A classical operation on streams is the *concatenation* of two streams that we denote by the infix operator ^. The concatenation is a function that takes two streams (say s and t) and produces a stream s^t as result starting with the elements in s and continuing with those in the stream t. Formally the concatenation has the following functionality:

$$.\hat{}. : M^\omega \times M^\omega \to M^\omega$$

If a stream s is infinite, then concatenating s with a stream t yields the stream s again:

$$s \in M^\infty \Rightarrow s\hat{}t = s$$

Concatenation is associative and has the empty stream ‹› as its neutral element:

$$r\hat{}(s\hat{}t) = (r\hat{}s)\hat{}t, \qquad \langle\rangle\hat{}s = s = s\hat{}\langle\rangle,$$

For a message $m \in M$ we denote by ‹m› the one element stream. For keeping our notation simple we extend concatenation also to elements from the set M (treating them like one element sequences) and to tuples of streams (by concatenating the streams in the tuples elementwise).

On the set M^ω of streams we define a *prefix ordering* $\sqsubseteq$. We write $s \sqsubseteq t$ for streams s and t to express that the stream s is a *prefix* of the stream t. Formally we have for streams s and t:

$$s \sqsubseteq t \quad \text{iff} \quad \exists\, r \in M^\omega : s\hat{}r = t.$$

The prefix ordering defines a partial ordering on the set M^ω of streams. If $s \sqsubseteq t$ holds, then we also say that the stream s is an *approximation* of the stream t. The set of streams ordered

by $\sqsubseteq$ is complete in the sense that every directed set $S \subseteq M^\omega$ of streams has a *least upper bound* denoted by $\sqcup S$. A nonempty subset S of a partially ordered set is called *directed*, if

$$\forall x, y \in S: \exists z \in S: x \sqsubseteq z \wedge y \sqsubseteq z .$$

By least upper bounds of directed sets of finite streams we may describe infinite streams. A *stream processing function* is a function

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

that is *prefix monotonic* and *continuous.* The function f is called prefix *monotonic*, if for all streams s and t we have:

$$s \sqsubseteq t \Rightarrow f.s \sqsubseteq f.t .$$

For better readability we often write f.x instead of the more common notation f(x) for function application. Infinite streams are also of interest as (and can also be described by) fixpoints of prefix monotonic functions. The streams associated with feedback loops in interactive systems correspond to such fixpoints.

The function f on streams is called *continuous*, if for all directed sets $S \subseteq M^\omega$ of streams we have

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

By $\sqcup S$ we denote the least upper bound of a set. If a function is continuous, then we can already predict its results for infinite input from its results on all finite approximations of the input.

By $\perp$ we denote the pseudo element that represents the result of diverging computations. We write $M^\perp$ for $M \cup \{\perp\}$. Here we assume that $\perp$ is not an element of the set M. On the set $M^\perp$ we define also a simple partial ordering by:

$$x \sqsubseteq y \quad \text{iff} \quad x = y \vee x = \perp$$

We use the following functions on streams

$$ft: M^\omega \to M^\perp,$$

$$rt: M^\omega \to M^\omega.$$

We define their properties as follows: the function ft selects the first element of a stream, if the stream is not empty. The function rt deletes the first element of a stream, if the stream is not empty. We can express the properties of the functions by the following equations (let $m \in M$, $s \in M^\omega$):

$$ft.\langle\rangle = \perp, \quad rt.\langle\rangle = \langle\rangle, \quad ft(m^\frown s) = m, \quad rt(m^\frown s) = s.$$

All the introduced concepts and functions for streams such as the prefix ordering and the concatenation carry over to tuples of streams and functions on streams or tuples of streams by applying them pointwise.

By #x we denote the length of a stream. This is a natural number for finite streams. Infinite streams have infinite length.

Given a set $S \subseteq M$ and a stream $x \in M^{\omega}$ we denote by S©x the substream of x consisting of the elements in S.

We denote the function space of (n,m)-ary stream processing functions by:

$$[(M^{\omega})^{n} \rightarrow (M^{\omega})^{m}]$$

The operations ft and rt are prefix monotonic and continuous, whereas concatenation ^ as defined above is prefix monotonic and continuous only in its second argument.

3. Asynchronous Interactive Components

In this section we outline a domain theoretic approach to the modeling of the black box behavior of nondeterministic reactive systems. We use the fundamental observation that the concept of approximation chains as it appears in fixpoint computations is a central mean for modeling computations.

3.1 Parallel and Sequential Interfaces

In theoretical computer science there is a long dispute whether explicit concurrency (sometimes even called "true" concurrency) is necessary or at least helpful when treating interactive, distributed systems. This issue is mainly discussed on the grounds of an operational semantics of a closed system. The discussion has led to a controversy between the proponents of the so-called interleaving semantics and the so-called "true" concurrency semantics. However, there was not much discussion, so far, how these issues are reflected in modular system descriptions. In this section we briefly discuss sequential interfaces in contrast to parallel (or concurrent) interfaces. Let us start with a basic informal definition.

An interactive (or reactive) system is an information processing object that interacts with its environment by sending or receiving messages. Its *syntactic interface* is defined by its set of input and output channels and by the sort of the messages it may send or receive on its channels. Its semantic interface that we also call its *black box behavior* is defined by the

causal relationship between the messages that it receives or sends. We call every instance of the sending or receiving of a message an *event*.

For a reactive component we call its interface *sequential*, if it can only process one event at a time. As a consequence, in every behavior of a sequential component there is a linear order for the execution of its events. Moreover, its output events may of course depend on the specific linear order of its input events.

A typical example of a component with a sequential interface is a deterministic state machine with the set of input messages I, the set of output messages O and a set of states S. The dynamic behavior of the state machine can be described by a transition function:

$$\delta: S \times I \to S \times O$$

With such a state machine we can associate traces with a straightforward linear order of events. Here we assume that every element in the trace taken from the set I or the set O stands for one event of sending and receiving messages. The behavior of a state machine can be easily represented by stream processing functions associated with its states

$$f: S \to (I^{\omega} \to O^{\omega})$$

The function f is specified by the following equation (for input messages $i \in I$, input streams $x \in I^{\omega}$, states $\sigma \in S$, output messages $o \in O$):

$$(f.\sigma).\langle\rangle = \langle\rangle$$

$$(f.\sigma)\ (i \ \& \ x) = o \ \& \ (f.\sigma').x \text{ where } (\sigma', o) = \delta(\sigma, i)$$

Since our state machine has such a simple "ping pong" behavior we immediately obtain the following equation (for every state σ and every input stream x):

$$\#x = \#(f.\sigma).x$$

The equation expresses the obvious fact that for every input message exactly one output message is generated.

A state machine nevertheless shows a kind of "parallel" behavior since input and output are processed in one step although we may assume that the output occurs only after the input has been received. A strictly sequential view is obtained by state transition systems defined by transition relations (cf [Lynch, Stark 89])

$$R \subseteq S \times (I \cup O) \times S$$

where for simplicity we assume that the sets of input messages I and output messages O are disjoint.

A transition step represented by $(\sigma, a, \sigma') \in R$ may either be an input or an output step. We obtain a strictly sequential order for each of the steps (each event) in a computation. We maintain the character of input steps and assume that input events are completely determined by the environment of the system. Therefore we assume that all input transitions can be

executed in every state without any restrictions. We call this the principle of *input enabledness*. It is formally expressed by the following proposition:

$$\forall\, \sigma \in S,\ a \in I: \exists\, \sigma' \in S: (\sigma, a, \sigma') \in R$$

Stream processing functions with one input stream and one output stream model reactive systems with sequential interfaces such as given by state machines with input, output and an initial state.

For a component with a concurrent interface several input and output events may take place concurrently. We can model a component with n input channels and m output channels by a stream processing function

$$f: (M^{\omega})^n \rightarrow (M^{\omega})^m$$

A graphical representation of f is given in Fig. 1.

This model of component behavior works well for a large variety of specifications. However, for certain components we run into difficulties that are caused by the following fact: according to the monotonicity assumption the relative order of the messages on different input channels cannot be used to determine the result of the function. We demonstrate this by the following basic example.

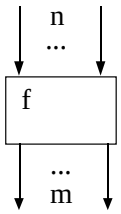

Fig. 1 Graphical representation of a component with n input and m output channels

Example (Independence of the output of the relative order of input on different channels): Let

$$n = 2$$

hold and assume that x, x_1, x_2 are pairs of streams where

$$x_1 \sqsubseteq x \wedge x_2 \sqsubseteq x$$

holds and where the pair x_1 contains an additional message in its first component while x_2 contains an additional message in its second component. Monotonicity implies the validity of the following formula:

$$f(x_1) \sqsubseteq f(x) \wedge f(x_2) \sqsubseteq f(x)$$

So the result of the function f on input history x cannot depend on the fact that the message on channel 1 comes first or that the message on channel 2 comes first. Such differences in the timing of messages cannot even be expressed for pairs of streams x. In other words, messages on the different channels may be received in parallel.

Allowing that messages arrive in parallel on different channels has the consequence that we cannot observe a relative order between these messages. Thus the behavior of a component cannot depend on this relative ordering. The function f models a nonsequential interface for components with a number of input channels $n > 1$. □

Recall that monotonicity models the causality between input and output histories.

Unfortunately, the interface modeled by prefix monotonic stream processing functions is so highly parallel that we cannot represent any relative order of messages on different input channels and therefore cannot use it to determine the result.

For specific components we need to have information about the relative order of input messages on its different input channels. A simple solution to this problem reads as follows: we consider streams

$$z \in (M^*)^\omega$$

instead of streams $x \in M^\omega$. We understand the stream z as a representation of a communication history of a channel in a time frame formed by an infinite sequence of time intervals. The kth entry in the stream represents the sequence of messages that are communicated along the channel in the time interval k.

If we model the behavior of a component by a function

$$g: ((M^n)^*)^\omega \rightarrow ((M^m)^*)^\omega$$

mapping streams of tuples of finite sequences onto streams of tuples of finite sequences then we obtain a function representing the behavior of a parallel interface which allows nevertheless to speak about the relative order of messages on different input or output channels. A stream

$$x \in ((M^*)^n)^\omega \qquad \text{where} \qquad x = x_1 \,\&\, x_2 \,\&\, x_3 \ldots$$

with $x_i \in (M^*)^n$ can be understood as the representation of an input history given in some discrete time frame. The tuple x_i of sequences of messages represents the sequences of messages received on the channels in parallel in the time interval i.

3.2 Theoretical Foundation: Approximation Chains

A complete communication history for n channels is given by an infinite stream which is an element of the set

$((M^*)^n)^\infty$.

We can understand every stream in this set as an infinite sequence of tuples of sequences of messages. The tuple represents the sequences of messages received on the individual channels during the time interval i.

Note that the sets $((M^*)^n)^\infty$ and $((M^*)^\infty)^n$ are isomorphic. This allows us to construct an input history for the n channels from the input histories for each of the n channels a.

In the following we give a theoretical foundation for stream processing functions operating on streams from $(M^*)^\infty$. However, to keep the theory as abstract as possible we represent $(M^*)^\infty$ by chains of streams from M^ω.

In domain and fixpoint theory we model the stepwise progress in computations of the output of a component by the concept of approximation. The basic idea is that during a computation step by step more information about the output of the component is generated. A partial order $\sqsubseteq$ is used to express for the partial (information about the) output x and y by $x \sqsubseteq y$ that the output y gives more (or at least as much) information about the output of the computation as x does and that the information of x is consistent with the perhaps more specific information of y.

We call the partially ordered set $(D, \sqsubseteq)$ an *algebraic domain*, if there is a subset $D_{fin} \subseteq D$ containing the "finite" elements of D such that every element $x \in D$ is the least upper bound of a directed set $S \subseteq D_{fin}$.

An element $e \in D$ is called *finite* if for every directed set $S \subseteq D$ we have

$$e \sqsubseteq \sqcup S \Rightarrow \exists\, d \in S:\ e \sqsubseteq d$$

In the case of the streams the finite elements are exactly the finite sequences.

Let $(D, \sqsubseteq)$ be an algebraic domain with approximation ordering $\sqsubseteq$ and the set $D_{fin} \subseteq D$ its set of finite elements. Any mapping

$$x: \mathbb{N} \to D_{fin}$$

with

$$x_i \sqsubseteq x_{i+1}$$

is called an *approximating chain* for the possibly infinite element $\sqcup x \in D$, where

$$\sqcup\ x = \sqcup\ \{x_i \in D_{fin}: i \in \mathbb{N}\}$$

Such a chain represents an *infinite stepwise approximation* of the element $\sqcup\ x \in D$ by finite elements. Such approximation chains clearly are closely related to computations and in particular to the timing of a computation.

By a chain we represent a computation of the element $\sqcup\ x$ that has computed the partial output represented by x_i until time point i.

We denote the set of chains of finite elements from the set D by

$\mathbf{C}(D)$

Of course there exists a surjective mapping from the set $\mathbf{C}(D)$ of chains over the domain D to the set D that maps every chain $x \in \mathbf{C}(D)$ to its least upper bound $\sqcup x$.

For streams M^{ω} and tuples of streams $(M^{\omega})^n$ the set of finite elements is given by M^* and $(M^*)^n$ respective. Every chain

$x \in \mathbf{C}((M^{\omega})^n)$

represents also a stream

$y \in ((M^*)^n)^{\infty}$

by the following definition which defines for given finite stream x_i the stream y_i uniquely.

$x_0 = y_1$

$x_{i+1} = x_i \hat{} y_i$

Obviously we can also always represent a chain in $\mathbf{C}((M^{\omega})^n)$ by a stream $((M^*)^n)^{\infty}$.

This shows that the sets $\mathbf{C}((M^{\omega})^n)$ and $((M^*)^n)^{\infty}$ are isomorphic. We prefer to work with $\mathbf{C}((M^{\omega})^n)$ instead of $((M^*)^n)^{\infty}$ in this section, since the concept of a chain is more general and therefore we can apply our construction for arbitrary domains.

We can understand a chain x of tuples of finite streams as a collection of observations about a system at time points $t_0 < t_1 < t_2 < \ldots$. We do not use a specific concrete physical time such as seconds or microseconds, but quite abstractly just assume that the elements of the tuple x_k represent the streams of observed messages up to time point t_k. In analogy, we interpret chains of input or output elements as observations about communication histories up to certain time points t_k.

Given a chain $x \in \mathbf{C}(D)$ we write $x|_i$ to denote the sequence of the first i elements of the chain x.

A function

$F: \mathbf{C}(D_1) \to \wp(\mathbf{C}(D_2))$

mapping chains onto sets of chains is called a *behavior*, if the first i elements (more precisely the messages received till timepoint i) of the input chain determine the first i elements in the output chain. In mathematical terms, for all $i \in \mathbb{N}$ we require:

$x|_i = z|_i \Rightarrow \{y|_i : y \in F(x)\} = \{y|_i : y \in F(z)\}$

The definition expresses that for a behavior the output history till time point i is determined exclusively by the input history till time point i.

A behaviour is called *time guarded*, if the first i elements in the input chain determine the first i+1 elements in the output chain. Time guardedness is equivalent to a delay property of a component.

We call a set-valued function

$$F: \mathbf{C}(D_1) \to \wp(\mathbf{C}(D_2))$$

limit closed, if for every chain $x \in \mathbf{C}(D_1)$ and every chain $y \in \mathbf{C}(D_2)$ we have

$$(\forall\, k \in \mathbb{N}: \exists\, z \in F(x): y|_k = z|_k) \Rightarrow y \in F(x)$$

We assume that every behavior function F is the union of a set of limit closed behaviors.

We call a component with the behavior F *timing independent*, if

$$\sqcup\, x = \sqcup\, z \Rightarrow \{\sqcup\, y: y \in F(x)\} = \{\sqcup\, y: y \in F(z)\}$$

We call a behavior F *consistent*, if it has at least one possible output history for every input history. Mathematically expressed, if for all input histories $x \in \mathbf{C}(D_1)$

$$F.x \neq \emptyset$$

We use the set $\mathbf{C}(D)$ to model the behavior of system components by set-valued functions

$$F: \mathbf{C}(D_1) \to \wp(\mathbf{C}(D_2))$$

that are behaviors and associate with every approximation of an input element in D_1 a set of approximations for the possible output histories of the component.

Example (The arbiter): An arbiter is a component that receives elements form a message set M. It reproduces all its input elements as output, but adds to its output stream an infinite number of clock signals that we can interpret as time out signals. Given a set of data elements M and the element √ called a *clock signal* we specify the function

$$arb: \mathbf{C}(M^\omega) \to \wp(\mathbf{C}((M \cup \{\surd\})^\omega))\backslash\{\emptyset\}$$

by the following formulas

$$y \in arb(x) \Rightarrow \sqcup\, x = M©(\sqcup\, y) \wedge \#\{\surd\}©(\sqcup\, y) = \infty$$

The specification is consistent, since there exists a behavior function that fulfills this specification. For instance, we may define for a chain $x = \{x_i \in D_{fin}: i \in \mathbb{N}\}$ the function arb by

$$arb(x) = \{y \in \mathbf{C}((M \cup \{\surd\})^\omega): \quad y_0 = \langle\rangle \wedge \; \forall\, i \in \mathbb{N}: M©y_{i+1} = x_i \wedge \; M©(\sqcup\, x) = M©(\sqcup\, y) \wedge \#\{\surd\}©(\sqcup\, y) = \infty\ \}$$

There is an essential difference between time and clock signals in our approach. We consider the timing as a physical frame modeled by the intervals of the streams while clock signals are technical means to synchronize behaviors. □

The output of functions on approximation chains x may depend, such as the output of I/O-machines, on the particular information *how* the input streams $\sqcup$ x are approximated by the chain x (which can be used to model its "timing") and thus on the approximation granularity of the input and not only on the least upper bound $\sqcup$ x. Under certain conditions we can, however, obtain a more abstract view of such functions that work on elements of D rather than on elements of $\mathbf{C}$(D) when we abstract away the information given by the approximation.

4.2 Abstractions from Functions on Approximations

In this section we describe properties of functions on approximations that allow to abstract from set-valued functions on approximation chains to functions on the approximated elements. We start by introducing a number of notions and definitions that are useful for reaching our goal.

The construction of appropriate abstractions is an important requisite for the treatment of complex systems. Here abstraction means the elimination of (unimportant) details. We begin with the introduction of a number of definitions that define a relationship between behaviors and stream processing functions.

A function

$$f: \mathbf{C}(D_1) \to \mathbf{C}(D_2)$$

is called a *descendant* of a function

$$F: \mathbf{C}(D_1) \to \wp(\mathbf{C}(D_2))$$

if for all chains $x \in \mathbf{C}(D_1)$ we have

$$f(x) \in F(x).$$

For a timing independent function

$$F: \mathbf{C}(D_1) \to \wp(\mathbf{C}(D_2))$$

which by definition has the property

$$\sqcup x = \sqcup z \Rightarrow \{\sqcup y: y \in F(x)\} = \{\sqcup y: y \in F(z)\}$$

we may construct an abstraction of the timing aspects contained in the behavior F by functions on D_1 as follows. Given a set valued F function

$$F : \mathbf{C}(D_1) \to \wp(\mathbf{C}(D_2))$$

we call a continuous function

$$f: D_1 \to D_2$$

a *weak abstraction* of the behavior F and we write

$$f \ \mathbf{abs}\ F$$

if for all input elements $x \in D_1$ we have:

$$\exists\, z \in \mathbf{C}(D_1): x = \sqcup\, z \wedge \exists\, y \in F(z): \forall\, i \in \mathbb{N}: f(z_i) \sqsubseteq y_i$$

A weak abstraction f is partially correct for the behavior F. In other words, f is correct with respect to the safety properties of F, but not necessarily with respect to the liveness properties of F.

We call the function f a weak abstraction, since it guarantees only output consistent with output contained in F, but it might not produce any output at all. If in addition $f(z_i) = y_i$ holds in the formula above, we call the function f a *strong abstraction.* We then write

$$f \ \mathbf{abs}^+ F.$$

Weak abstractions always exist. Strong abstractions do not always exist according to the monotonicity requirement for the function f.

Example (Behavior without strong abstractions): The arbiter of the example above does not have strong abstractions. According to the specification of the arbiter for a weak abstraction f of the arbiter specification we assume

$$M©f.x \sqsubseteq x$$

For a strong abstraction we, in addition we require

$$\#\{\surd\}©f.x = \infty$$

So we may conclude

$$f.\langle\rangle = \surd^\infty$$

$$\exists\, k: f.\langle m\rangle = \surd^k \,\hat{}\, \langle m\rangle \,\hat{}\, \surd^\infty$$

There does not exist a monotonic function that fulfills these equations, since

$$\langle\rangle \sqsubseteq \langle m\rangle$$

and therefore

$$f.\langle\rangle \sqsubseteq f.\langle m\rangle$$

is required which does not hold for the function specified above. □

We call a behavior function a *strongly time independent*, if its behavior can be described by its set of strong abstractions. In mathematical terms, then we have

$$F(x) = \{y \in \mathbf{C}(D_2): \exists\, f: \sqcup\, y = f(\sqcup\, x) \wedge f\ \mathbf{abs}^{+}\ F\}$$

As we have seen already, not all behaviors have strong abstractions. This is not surprising, since we cannot expect strong abstractions for strongly time dependent specifications. However, for a subclass of the time dependent specifications we can find abstractions by so-called *input choice specifications* (for a detailed description see [Broy 93]).

An *input choice specification* for a component with input domain D_1 and an output domain D_2 is a relation

$$R: [D_1 \to D_2] \times D_1 \to \mathbb{B}$$

R is called *abstract specification* of a behavior F, if the following formula is valid

$$R(f, x) \equiv f\ \mathbf{abs}\ F \wedge \exists\, z \in \mathbf{C}(D_1): x = \sqcup\, z \wedge \exists\, y \in F(z): \forall\, i \in \mathbb{N}: f(z_i) = y_i$$

The proposition R(f, x) expresses that f fulfills the safety properties required by F and for input x also the liveness properties of F. In general, the predicate R cannot characterize the behavior of F. But for time independent behavior R characterizes F. So R can be seen as a description of F in a model without time.

Theorem: Every time independent behavior can be defined by an input choice specification.

Proof: Given a time independent behavior

$$F: \mathbf{C}(D_1) \to \wp(\mathbf{C}(D_2))$$

We define an input choice specification

$$R: [D_1 \to D_2] \times D_1 \to \mathbb{B}$$

by the definition

$$R(f, x) \equiv f\ \mathbf{abs}\ F \wedge \forall\, z \in \mathbf{C}(D_1): x = \sqcup\, z \Rightarrow f.x \in \{\sqcup\, y: y \in F.z\}$$

We define a behavior F' based on R by

$$F'(z) = \{y \in \mathbf{C}(D_2): \exists\, f: R(f, \sqcup\, z) \wedge \sqcup\, y = f(\sqcup\, z)\}$$

By the definition of R and the time independence of F we obtain F = F'. □

We can use the input choice specification R of F as a representation or a specification of F at a more abstract level.

5. Composition Operators

When modeling systems and system components the composition of larger systems from smaller ones is a basic structuring technique. We consider only three basic composition operators, namely *sequential* composition, *parallel* composition and *feedback.*

As well known these three composition operators suffice to formulate all kinds of networks of reactive information processing components, provided we have simple components available for permuting and copying input and output lines.

5.1 Parallel Composition

To define parallel composition we first introduce an operation that allows to form a chain of tuples (of streams) out of two chains of tuples (of streams). Let D_1 and D_2 be domains. Given two chains of streams

$$x \in \mathbf{C}(D_1),\ y \in \mathbf{C}(D_2)$$

we construct a chain

$$x \oplus y \in \mathbf{C}(D_1 \times D_2)$$

by elementwise composition as follows:

$$(x \oplus y).i = x.i \oplus y.i$$

where $x.i \oplus y.i$ is the tuple obtained by concatenating the tuples x.i and y.i.

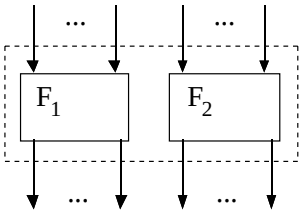

Fig. 2 Parallel composition

Given two behaviors

$$F_1 : \mathbf{C}(D_{1,1}) \to \wp(\mathbf{C}(D_{2,1}))$$

$$F_2 : \mathbf{C}(D_{1,2}) \to \wp(\mathbf{C}(D_{2,2}))$$

we define the parallel composition

$F_1||F_2 : \mathbf{C}(D_{1,1} \times D_{1,2}) \rightarrow \wp(\mathbf{C}(D_{2,1} \times D_{2,2}))$

by the behavior defined by the following formula (let $x_1 \in \mathbf{C}(D_{1,1})$, $x_2 \in \mathbf{C}(D_{1,2})$):

$(F_1||F_2)\ (x_1 \oplus x_2) = \{y_1 \oplus y_2 : y_1 \in F_1(x_1) \wedge y_2 \in F_2(x_2)\}$

It is a straightforward proof to show that the parallel composition $F_1||F_2$ leads to a behavior provided F_1 and F_2 are behaviors.

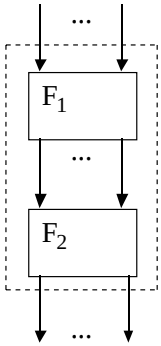

Fig. 3 Sequential composition

5.2 Sequential Composition

Sequential composition is simple to define by functional composition. Give two behaviors

$F_1: \mathbf{C}(D_0) \rightarrow \wp(\mathbf{C}(D_1))$

$F_2: \mathbf{C}(D_1) \rightarrow \wp(\mathbf{C}(D_2))$

we define the sequential composition

$(F_1 ; F_2) : \mathbf{C}(D_0) \rightarrow \wp(\mathbf{C}(D_2))$

by

$(F_1 ; F_2).x = \{y : \exists\ z : z \in F_1(x) \wedge y \in F_2(z)\}$

Again it is a straightforward proof to show that the sequential composition $(F_1 ; F_2)$ is a behavior provided F_1 and F_2 are behaviors.

5.3 Feedback Operator

Given a behavior

$$F: \mathbf{C}(D_1 \times D_0) \to \wp(\mathbf{C}(D_2 \times D_0))$$

we define the behavior

$$\mu F: \mathbf{C}(D_1) \to \wp(\mathbf{C}(D_2))$$

by the equation

$$(\mu F).x = \{(z, y) \in D_2 \times D_0 : (z, y) \in F(x, y)\}.$$

Again it is a straightforward proof to show that the feedback μF is a time guarded behavior provided F is a time guarded behavior.

Theorem: A time guarded consistent behavior

$$F: \mathbf{C}(D) \to \wp(\mathbf{C}(D))$$

has a (least) fixpoint $c \in F(c)$.

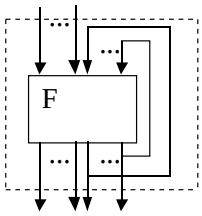

Fig. 4 Feedback operator

Proof: Define the elements $z_i \in D$ by

$$z_0 = \bot$$

and

$$z_{i+1} \in \{y_{i+1} : \exists\, x \in \mathbf{C}(D): y \in F.x \wedge \forall\, j: j \le i \Rightarrow x_j = z_j\}$$

The elements z_i form a chain z. Furthermore for this chain z we have

$$z \in F(z)$$

since F is limit closed. So z is a fixed point. □

If a behaviour F is time independent, then μF is time independent. If F is strongly time independent, then μF is strongly time independent.

6. Specifications

The model introduced above is very powerful. It supports the specification of reactive systems with and without timing aspects. Many small case studies have shown the power and flexibility of the specification, development and verification methods based on this model. However, writing such specifications is a difficult task. From the mathematical and logical basis it is a long way to a applicable specification method that can be used by an engineer. In the following section we demonstrate how a more practical specification method can be based on the model introduced so far.

We introduce a number of specification techniques for the description of reactive components. As we have demonstrated above we can specify systems with a parallel interface and a possibly time dependent behavior by mathematical formulas describing functions on chains of tuples of streams that yield sets of chains of tuples of streams. However, for many applications it is more appropriate to use more specific and better readable specification techniques. We demonstrate such specification techniques in the following by a number of simple examples.

6.1 Relations Between Input and Output Histories

For practical purposes one might be interested in a more formalized syntax. It is not difficult to provide a logic based syntax for our formalism. A simple form of such a syntax could be predicate logic to specify the relation between the input and output histories.

We have already seen a simple relational specifications (the arbiter). We give an additional example that illustrates the important concept of an interrupt.

Many concepts in systems programming, in process control, and in telecommunication show a behavior of the following type: do something until a certain event happens. For message passing components we can express this often as: send (or receive) messages until a particular message is received. Such a behavior corresponds to what we call an interrupt. A typical example is the sender in a protocol of the type of the alternating bit protocol or many other similar communication protocols.

Example (Sender protocol): We consider a component with the behavior

$$\mathrm{SP}: \mathbf{C}(M^{\omega} \times A^{\omega}) \to \wp(M^{\omega})$$

where M is a set of messages and A = {ack}. We define the sender protocol SP by the following formula

$$y \in SP(x, z) \equiv \exists\, r \in \mathbb{N}^\omega: \#r = \#\sqcup\ z \wedge \sqcup\ y = do(\sqcup\ x, r^\frown\langle\infty\rangle)$$

where

$$do: M^\omega \times (\mathbb{N} \cup \{\infty\})^\omega \to M^\omega$$

with (let $m \in M$, $n \in \mathbb{N}$, $x \in M^\omega$, $r \in (\mathbb{N} \cup \{\infty\})^\omega$):

$$do(\langle m\rangle^\frown x, n^\frown r) = m^{n\frown} do(x, r).$$ □

We can understand many of our example problems given so far as instances of such an interrupt behavior:

(1) Fair nonstrict merge: process messages from the left input line until a message arrives on the right input line.

(2) Arbiter: Send signals √ until a message arrives on the input line.

Interrupt behavior is a very typical and important behavior when dealing with reactive systems. Of course, we have given very abstract specifications of interrupts. More specific interrupt specifications may take the timing into consideration, too, and model behaviors that react "immediately" (more precisely, within a certain amount of time) for any interrupt.

6.2 State Transition Specifications

In a state transition specification we specify the behavior of a component by a state transition system with input and output. In a state transition specification we use a set of states as an auxiliary structure.

For a state transition specification we introduce a set of states. In the example of the arbiter we may choose as states the messages received but not sent so far.

We can choose any appropriate set as the set of states. Let State be the set of states. We define a relation:

$$R: State \times D_1^* \to \wp(State \times D_2^*)$$

and an initial state set $State_0$. We can define this relation in many ways using logic, tables or just plain mathematics. We then define a behavior

$$F: \mathbf{C}(D_1^\omega) \to \wp(\mathbf{C}(D_2^\omega))$$

as follows: we define that a history y is a possible result of the behavior F applied to the input history x, in mathematical terms

$$y \in F(x),$$

if and only if there exists a sequence of states σ_i that form a computation for the relation R. The sequence of states σ_i that forms a computation if for every $i \in \mathbb{N}$ there exists a state transition from state σ_i to σ_{i+1} such that the input and output of this transition define the messages that have to be added to x_i and y_i to obtain x_{i+1} and y_{i+1}. Formally we have

$$\exists\, a, b: x_{i+1} = x_i \hat{} \langle a \rangle \wedge y_{i+1} = y_i \hat{} \langle b \rangle \wedge (\sigma_{i+1}, b) \in R(\sigma_i, a)$$

and σ_0 is in the set of initial states $State_0$:

$$\sigma_0 \in State_0.$$

This way a state machine defines a function on chains (of tuples) of streams.

Example (Locked Access to Data): A component that guarantees locked access to a number of memory cells can be specified as follows. We use the following sets:

Data	data values to be read or written,
Key	identifier for the memory cells,
Id	identifier for the processes that access the memory cells.

We define the state set as follows

$$State: Key \to Data \times (Id \cup \{nil\})$$

For a state σ and a key k the value $(\sigma.k).2 \in Id$ indicates for which process the key k is locked. The special value nil (where we assume $nil \notin Id$) expresses that the key is not locked for a process. We use the following set In of input messages.

$$\begin{aligned} In = \quad & \{Lock(k, i): k \in Key \wedge i \in Id\} \cup \\ & \{Read(k, i): k \in Key \wedge i \in Id\} \cup \\ & \{Write(k, i, d): k \in Key \wedge i \in Id \wedge d \in Data\} \cup \\ & \{Unlock(k, i): k \in Key \wedge i \in Id\} \end{aligned}$$

and the following set Out of output messages.

$$\begin{aligned} Out = \{ & Ack(x): x \in In \backslash Re\} \cup \\ & \{Fail(x): x \in In\} \cup \\ & \{Ack(x, v): x \in Re \wedge v \in Data\} \end{aligned}$$

Where Re is the set of messages in the set of input messages In labeled with a read.

Now we define the transition relation R by the following proposition:

$$(\sigma', o) \in R(\sigma, x) \equiv$$

$$\begin{array}{lll}
(x = Lock(k, i) \wedge & ((o = Ack(x) \wedge \sigma(k).2 = nil \wedge \sigma' = \sigma[i/k.2]) \vee \\
 & (o = Fail(x) \wedge \sigma(k).2 \neq nil \wedge \sigma' = \sigma)) \vee \\
(x = Write(k, i, d) \wedge & ((o = Ack(x) \wedge \sigma(k).2 = i \wedge \sigma' = \sigma[d/k.1]) \vee \\
 & (o = Fail(x) \wedge \sigma(k).2 \neq i \wedge \sigma' = \sigma)) \vee \\
(x = Read(k, i) \wedge & ((o = Ack(x, \sigma(k).1) \wedge \sigma(k).2 = k \wedge \sigma' = \sigma) \vee \\
 & (o = Fail(x) \wedge \sigma(k).2 \neq i \wedge \sigma' = \sigma)) \vee \\
(x = Unlock(k, i) \wedge & ((o = Ack(x) \wedge \sigma(k).2 = i \wedge \sigma' = \sigma[nil/k.2]) \vee \\
 & (o = Fail(x) \wedge \sigma(k).2 \neq i \wedge \sigma' = \sigma))
\end{array}$$

Here we write

$$\sigma[i/k.2]$$

and

$$\sigma[d/k.1]$$

to denote the update of the second or first respective component of $\sigma(k)$ by the vakue d.

For such a problem a state-oriented description seems much more appropriate than one in terms of pure history relations. By the relation R we describe the behavior of a locked access component we call LAC. □

State based specifications often are helpful to express the data dependencies between messages in a more suggestive if the state space is chosen carefully.

6.3 Relational Specifications

In a relational specification we specify a component by a formula referring to the input and output streams written in first order predicate logic.

A relational specification of a weakly time dependent specification can also be given by a relation

$$R \subseteq D_1 \times D_2$$

The relation R defines a function

$$F: \mathbb{C}(D_1) \rightarrow \wp(\mathbb{C}(D_2))$$

by the following formula

$$F(x) = \{y : (\sqcup\ x, \sqcup\ y) \in R\}$$

In addition we can indicate that the relation is time independent.

Example (Fair Merge): We define the relation R by the formula

$$((x_1, x_2), y) \in R \Leftrightarrow \exists\ b \in \mathbb{B}^{\infty}: x_1 = sched(y, b) \wedge x_2 = sched(y, \neg^* b)$$

where the function sched is defined by the following formula

$$sched(x, b) = \textbf{if}\ ft.b\ \textbf{then}\ ft.x\ \&\ sched(rt.x, rt.b)\ \textbf{else}\ sched(rt.x, rt.b)\ \textbf{fi}$$

$$\neg^* b = \neg ft.b\ \&\ \neg^* rt.b$$ □

We give a second example of a time independent component.

Example (Delayed Permutation Component): A behavior that we can observe in many examples of the type of remote procedure control mechanisms are delayed permutation function. Given a set M a behavior

$$DPC: \mathbf{C}(M^{\omega}) \rightarrow \wp(\mathbf{C}(M^{\omega}))$$

is called a delayed permutation component provided DPC fulfills the following specification:

$$y \in DPC(x) \equiv \forall\ m \in M: \{m\} \copyright_{\sqcup}\ y = \{m\} \copyright_{\sqcup}\ x$$

A delayed permutation accepts messages from the set M as input and reproduces them as output after a while. DPC produces only messages as output that have been received as input (safety). All messages are delayed only for a finite amount of time and then produced as output (liveness). It is one of the advantages of the presented specification method that an artificial separation of properties into safety and liveness properties is not required, but possible, if wanted. We can combine the component LAC with the delayed permutation component by

$$DPC\ ;\ LAC\ ;\ DPC$$

This component may rearrange the order of its input messages and the order of its output messages. □

All the specification techniques we have available for defining relational specifications can this way be used for defining behavior functions.

6.4 Pragmatic Specification Techniques

In principle we can write specifications by predicates that characterize set valued functions that we call behaviors. However, such specifications are hardly readable by software engineers that are not familiar with predicate logic and the foundations of fixpoint theory. For them it is certainly helpful if more pragmatic specification techniques are available.

Useful pragmatic specification techniques, that are familiar to engineers, may be graphics, tables and diagrams.

Example (Table of a relation): For the example given in section 6.2 a table is more readable than the rather lenghty logical formula. In the following tabele every line corresponds to a formula.

$(\sigma', o) \in R(\sigma, x) \equiv (x = ... \wedge \sigma(k).2 \; ... \Rightarrow o = ... \wedge \sigma' = ...) \wedge ...$			
x =	$\sigma(k).2$	o =	σ'
Lock(k, i)	= nil	Ack(x)	$\sigma[i/k.2]$
	$\neq$ nil	Fail(x)	σ
Write(k, i, d)	= i	Ack(x)	$\sigma[d/k.1]$
	$\neq$ i	Fail(x)	σ
Read(k, i)	= i	Ack(x, $\sigma(k).1$)	σ
	$\neq$ i	Fail(x)	σ
Unlock(k, i)	= i	Ack(x)	$\sigma[nil/k.2]$
	$\neq$ i	Fail(x)	σ

□

Of course we can use tables both for describing relations and for describing state machines. This way we obtain more practical techniques both for relational and for state oriented specifications.

The semantic model of a reactive system may be difficult to handle for a systems engineer, since it is tuned too much towards mathematics. Nevertheless, we may give rather intuitive explanations for our model. Furthermore we can give more syntactic sugar for the specification of a component. We shortly outline (without going into a concrete syntax) how such a specification technique might look like.

We describe a component by introducing identifiers for each of its input or output channels and a sort for the messages of its identifiers. The sorts of the messages can be defined by any appropriate specification mechanism for describing data types.

Then the relation between the input and the output streams is described. For doing so for an input or output stream x we can refer by x.t (where $t \in \mathbb{N}$) to the input or output produced

up to time point t. By $x[t_1 : t_2]$ we refer to the input or output in the time interval $[t_1 : t_2]$. We write x[t] instead of x[t : t]. Both x.t and $x[t_1 : t_2]$ are always finite sequences.

If appropriate we may introduce a local state in a component to write a state-based specification.

7. Conclusions

The specification of interactive system has to be done in a time/space frame. A specification should indicate which events (communication actions) can take place where, when and how they are causally related. Such specification techniques are an important prerequisite for the development of safety critical systems.

Modeling information processing systems appropriately is basically a matter of choosing the adequate abstractions in terms of the corresponding mathematical models. Giving operational models that contain all the technical computational details of interactive nondeterministic computations is relatively simple. However, for system engineering purposes operational models are not very helpful. Only, if we manage to find good abstractions is it possible reach a tractable basis for system specifications.

Abstraction means forgetting information. Of course, we may forget only information that is not needed. Which information is needed does not only depend upon the explicit concept of observation, but also upon the considered forms of the composition of systems.

Finding appropriate abstractions for operational models of distributed systems is a difficult but nevertheless important task. Good abstract nonoperational models are the basis of a discipline of system development.

Acknowledgment

I am grateful to Ketil Stølen for a number of discussions that were helpful to clarify the basic concepts.

References

[Broy 83]
M. Broy: Applicative real time programming. In: Information Processing 83, IFIP World Congress, Paris 1983, North Holland Publ. Company 1983, 259-264

[Broy 85]
M. Broy: Specification and top down design of distributed systems. In: H. Ehrig et al. (eds.): Formal Methods and Software Development. Lecture Notes in Computer Science 186, Springer 1985, 4-28, Revised version in JCSS 34:2/3, 1987, 236-264

[Broy 86]
M. Broy: A theory for nondeterminism, parallelism, communication and concurrency. Habilitation, Fakultät für Mathematik und Informatik der Technischen Universität München, 1982, Revised version in: Theoretical Computer Science 45 (1986) 1-61

[Broy 87a]
M. Broy: Semantics of finite or infinite networks of communicating agents. Distributed Computing 2 (1987), 13-31

[Broy 87b]
M. Broy: Predicative specification for functional programs describing communicating networks. Information Processing Letters 25 (1987) 93-101

[Broy 93]
M. Broy: Functional Specification of Time Sensitive Communicating Systems. REX Workshop. ACM Transactions on Software Engineering and Methodology 2:1, Januar 1993, 1-46

[Broy, Stølen 94]
M. Broy, K. Stølen: Specification and Refinement of Finite Dataflow Networks – a Relational Approach. In: Langmaack, H. and de Roever, W.-P. and Vytopil, J. (eds): Proc. FTRTFT'94, Lecture Notes in Computer Science 863, 1994, 247-267

[Kahn, MacQueen 77]
G. Kahn, D. MacQueen: Coroutines and networks of processes, Proc. IFIP World Congress 1977, 993-998

[Lynch, Stark 89]
N. Lynch, E. Stark: A proof of the Kahn principle for input/output automata. Information and Computation 82, 1989, 81-92

[Lynch, Tuttle 87]
N. A. Lynch, M. R. Tuttle: Hierarchical correctness proofs for distributed algorithms. In: Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing, 1987

[Park 80]
D. Park: On the semantics of fair parallelism. In: D. Björner (ed.): Abstract Software Specification. Lecture Notes in Computer Science 86, Berlin-Heidelberg-New York: Springer 1980, 504-526

[Park 83]
D. Park: The ``Fairness'' Problem and Nondeterministic Computing Networks. Proc. 4th Foundations of Computer Science, Mathematical Centre Tracts 159, Mathematisch Centrum Amsterdam, (1983) 133-161

[Pannangaden, Shanbhogue91]
P. Pannangaden, V. Shanbhogue: The expressive power of indeterminate dataflow primitives. Information and Computation 98, 1992, 99-131