Algebraic Specification of Reactive Systems[*]

Manfred Broy

Institut für Informatik, Technische Universität München
D-80290 München

Abstract. We present an algebraic method for the specification of reactive distributed systems. We introduce basic operators on specifications making the set of specifications into a specification algebra. This allows us to work with an algebra of system specifications in analogy to the process algebras that provide algebras of reactive programs. However, in contrast to process algebras we work with a concrete representation (a mathematical system model) of specifications and use algebaric equations to specify components and not programming languages. A specification is represented by a predicate that describes a set of behaviors. A deterministic component has exactly one behavior. A behavior is represented by a stream processing function. We introduce operations on behaviors and lift them to specifications. We show how algebraic system specifications can be used as an algebraic and logical basis for state automata specifications and state transition diagrams.

1 Introduction

There are many approaches to the formal description and specification of distributed interactive systems. We follow the idea of functional system modeling as it is explained in detail for instance under the keyword FOCUS in [FOCUS 94] or in [Broy 93] (for the theoretical background see also [Broy 86] and [Broy 87b]). There the basic idea is to describe the input/output relationship called the *black box behavior* of a system component by specifying predicates that characterize sets of deterministic behaviors. A deterministic behavior is represented by a stream processing function.

In the following, we extend and complement the functional approach by algebraic specification concepts. We specify a number of basic operations on specifications. This allows us to write elegant algebraic equations to characterize specifications.

Our motivation is to extend the functional approach to system specification by concepts and notations that provide the most simple and suggestive way of writing specifications for certain types of components. Algebraic specification techniques for reactive systems allow us to describe reactive systems by specifying equations very much along the line of algebraic specification techniques for data structures.

We show, in particular, the close relationship between finite automata and extended finite automata and algebraic techniques. Extended finite automata are automata with a finite control but an infinite data space. There are a number of system description concepts such as state transition diagrams that are used in practice (such as for instance in SDL, see [SDL 88]) that are based on this idea. Our approach provides an algebraic and logical foundation for these concepts.

There are components for which it is much easier and more elegant to describe their behavior by algebraic equations for their specification than by classical predicate logic. We introduce, in particular, a tuned notation for the algebraic specification of reactive systems.

Our paper is organized as follows. Section 2 briefly introduces the basic notion of component that we use. Section 3 introduces operations on behaviors called input and output transition. Section 4 defines the algebra of specifications based on transitions. Section 5 defines recursion for specifications. Section 6 considers the fundamental forms of

[*]) This work was partially sponsored by the Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen" and the industrial research project SysLab sponsored by Siemens Nixdorf and by the DFG under the Leibniz program.

compositions for building systems. Section 7 gives the algebraic laws for the algebra of transitions and system composition operations. Section 8 illustrates the relationship of this algebra to state transition diagrams. Section 9 outlines an extension of our approach to time dependent systems. An appendix repeats the basic notions of streams.

2 Components, Interfaces, Behaviors, Specifications

The concept of a *component* is fundamental in software and systems engineering. Distributed systems are composed of components that are connected in a way that allows them to exchange information and to cooperate. We use a very simple, but powerful, abstract, general, mathematical concept of a component.

A component has an interface described by a family of named input channels as well as a family of named output channels. By the channels a component communicates with its environment. By I we denote the set of input channel identifiers and by O we denote the set of output channel identifiers. Each channel has assigned a sort. For simplicity throughout this paper all channels carry messages of the same sort M. An extension to individually sorted channels is straightforward. Fig 1 shows a component with individually sorted channels as a data flow node.

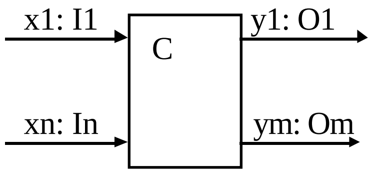

Fig. 1 Graphical representation of component C with sorted and named channels

Semantically, a component C is represented by a predicate defining a set of deterministic behaviors. A *deterministic behavior* is represented by a stream processing function

$$f : (I \rightarrow M^{\omega}) \rightarrow (O \rightarrow M^{\omega})$$

that maps every input history onto an output history. An input or output history is given by a valuation of the channels by streams. Here M^{ω} denotes the set of streams over the set M. It is defined by $M^{\omega} = M^{*} \cup M^{\infty}$. It consists of the finite and infinite sequences of elements from M. The concatenation of two sequences s and t is denoted by $s\hat{\ }t$. A detailed introduction to streams and stream processing functions is given in the appendix.

A component then is described by a predicate

$$C: ((I \rightarrow M^{\omega}) \rightarrow (O \rightarrow M^{\omega})) \rightarrow \mathbb{B}$$

that specifies a set of deterministic behaviors. Given such a component C we denote by In(C) its set I of input channels and by Out(C) its set O of output channels.

In many applications, it is useful to work with specifications that are parameterized. Mathematically, this means that we deal with a function

$$Q: \Xi \rightarrow (((I \rightarrow M^{\omega}) \rightarrow (O \rightarrow M^{\omega})) \rightarrow \mathbb{B})$$

where Ξ is an arbitrary set. For every element $\sigma \in \Xi$ we obtain by $Q(\sigma)$ a component specification. One way to think about Ξ is to see it as a state space.

Throughout this paper we write f.x for the function application f(x) whenever appropriate.

3 The Algebra on Behavior Functions

In this section we introduce a mathematical concept of operations applied to the black box behavior of deterministic components. We start with the introduction of a basic operation on streams. Let $x \in M^{\omega}$ be a stream and $m \in M$ be a message. We write

$$m \lessdot x$$

for the stream $\langle m\rangle\hat{\ }x$ that starts with message m and then continues with the stream x of messages. We extend this notation to finite sequences $s \in M^*$ and write

$$s \lessdot x$$

for the stream $s\hat{\ }m$ that starts with the messages in s and continues with the stream x.

We extend this notion also to families of named streams. Let A be a set of names for streams which we call *channels*. Let $x \in A \to M^\omega$ be a family of streams with names from A, $c \in A$ be a channel and $m \in M$ be a message. We extend the concatenation on streams to valuations $x: A \to M^\omega$ and $y: A \to M^\omega$ elementwise by the definition

$$(x\hat{\ }y).a = (x.a)\hat{\ }(y.a)$$

We specify the family of streams

$$c{:}m \lessdot x$$

by the following equations:

$$(c{:}m \lessdot x).c' = x.c' \qquad \Leftarrow c \neq c',$$

$$(c{:}m \lessdot x).c = \langle m\rangle\hat{\ }(x.c),$$

$$c{:}m \lessdot x = x \Leftarrow c \notin A.$$

The behavior of a deterministic component with the set I of input channels and the set O of output channels is described by stream processing functions

$$f : (I \to M^\omega) \to (O \to M^\omega)$$

mapping each valuation of the input channels I by streams over M onto valuations of the output channels O by streams from M^ω.

We define for messages $m \in M$, channels $c \in I$ and input histories $x \in I \to M^\omega$ the following operation on the stream processing function f:

$$f \lessdot c{:}m$$

By this term we denote the stream processing function that behaves like the function f on the communication history $x \in (I \to M^\omega)$ after we add the message m as the first message on channel c to the input x. Formally, this explanation can be expressed by an equation as follows:

$$(f \lessdot c{:}m).x = f(c{:}m \lessdot x)$$

For every output channel $c \in O$ we define in a similar way the stream processing function denoted by

$$c{:}m \lessdot f$$

as the function that represents the same behavior as the function f but always adds the message m as its first output on channel c to the output produced by the function f. Again we can express this description formally by an equation as follows:

$$(c{:}m \lessdot f).x = c{:}m \lessdot (f.x)$$

These operations on functions are called *input transitions* and *output transitions*. They can easily be generalized from stream processing functions to sets of stream processing functions and therefore to specifications.

With the transition operators introduced so far we can already specify behaviors by algebraic equations.

Example: Simple Memory Component
A memory component that can store data values has one input channel i and one output channel o. Let D be a set of data and ® denote the request signal. We define a behavior f for such a component with one input channel i and one output channel o of sort

$$M = D \cup \{®\}$$

by the equations (let $e, d \in D$):

$$f \lhd i{:}d \lhd i{:}® = o{:}d \lhd f \lhd i{:}d \qquad \text{(Read value equation)}$$

$$f \lhd i{:}d \lhd i{:}e = f \lhd i{:}e \qquad \text{(Write value equation)}$$

By these algebraic equations the behavior function f is only loosely specified since nothing is fixed about the behavior of the function

$$f \lhd i{:}®.$$

A state diagram description of our example is given in Fig. 2. The node of the state transition diagram denotes a state represented by the behavior $f \lhd i{:}d$. Each equation corresponds to a transition with an input pattern separated by the symbol "/" from the corresponding output pattern followed by a statement that specifies the associated state change. A more detailed explanation of state transition diagrams will be given in section 8. □

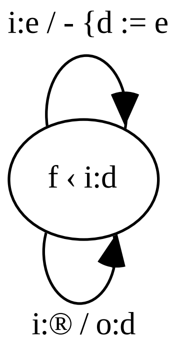

Fig. 2 State diagram description of the memory component

The example shows a classical algebraic specification style for the description of reactive systems.

4 The Algebra of Specifications

A system behavior is specified by a predicate Q that characterizes a set of behaviors represented by stream processing functions. Formally, we have

$$Q : \big((I \to M^{\omega}) \to (O \to M^{\omega})\big) \to \mathbb{B}$$

We can think about the predicate Q in the following also as a set of functions. As it is well-known, all operations on functions can be extended to sets of functions and therefore to specifications by applying them pointwise to the elements in this set. We write

$$Q \lhd x{:}m$$

for the specification of the set of functions that we obtain from the set of functions described by the predicate Q by applying the operation to each of these functions. This corresponds to the following formal definition of the predicate

$$Q \lhd x{:}m \equiv \lambda\, f : \exists\, f' : Q.f' \wedge f = f' \lhd x{:}m$$

According to this definition the specification

$$Q \lhd x{:}m$$

characterizes all behaviors f for which there is a behavior f' with Q.f' such that f behaves like f' after it has received the message m on channel x.

In analogy we write

$$x{:}m \lhd Q$$

for the specification (the predicate) characterized by the following equation:

$$x{:}m \,‹\, Q \equiv \lambda\, f : \exists\, f' : Q.f' \wedge f = x{:}m \,‹\, f'$$

According to this definition the specification

$$x{:}m \,‹\, Q$$

characterizes all behaviors f for which there is a behavior f' with Q.f' such that f behaves like the function f' after producing the message m on the output channel x.

The transitions

$$\lambda\, Q{:}\ Q \,‹\, x{:}m$$

and

$$\lambda\, Q{:}\ x{:}m \,‹\, Q$$

define algebraic operators on specifications. We call these operators again *input* and *output transitions*.

If a component described by a specification has only one input and one output channel (which need not be named by channel identifiers then) we write

$$Q \,‹\, m$$

and

$$m \,‹\, Q$$

without explicitly referring to the channel with the meaning as explained above[1].

Our notation of input and output transitions is simply extended from messages $m \in M$ to finite nonempty sequences $s \in M^*$ of messages by the definitions

$$Q \,‹\, (‹m›\hat{}s) = (Q \,‹\, m) \,‹\, s$$

$$Q \,‹\, c{:}(‹m›\hat{}s) = (Q \,‹\, c{:}m) \,‹\, c{:}s$$

$$Q \,‹\, ‹› = Q \,‹\, c{:}‹› = Q$$

where $m \in M$. In analogy we extend our notation to sequences of output transitions and write $s \,‹\, Q$ and $c{:}s \,‹\, Q$ resp.

5 Recursive Equations for Specifications

Using the operators on specifications introduced above we can write axiomatic equations for specifications. Besides the operators introduced explicitly above we can also use all logical connectors for composing specifications since specifications formally are nothing but predicates. Since the set of predicates forms a complete lattice the treatment of recursion is straightforward.

Often we are interested in recursive specifications for predicates Q using equations like

$$Q \equiv \psi(Q)$$

where ψ is an implication monotonic (inclusion monotonic if we think about predicates as sets) operator on specifications. Then we can work with well-known concepts from μ-calculus.

We write

$$Q{:}\ [\ \psi(Q)\ ]$$

for the weakest specification (the weakest predicate) called Q that fulfills the equation

$$Q \equiv \psi(Q).$$

This property of the specification of Q can be formally expressed by the following two axioms:

---

1) This notation can also be used in cases of sorted channels for messages the sorts of which identify the channel uniquely.

$$Q: [\psi(Q)] \Rightarrow Q \equiv \psi(Q) \qquad \text{(fixpoint property)}$$

$$Q: [\psi(Q)] \wedge (R \Rightarrow \psi(R)) \Rightarrow (R \Rightarrow Q) \qquad \text{(least fixpoint)}$$

These two axioms can be used to prove properties about the specification of the predicate Q by Q: [ψ(Q)]. In classical λ-notation the proposition

$$Q: [\psi(Q)]$$

stands for

$$Q = \textbf{fix}\ \lambda\ Q: \psi(Q)$$

Now we are ready to give a first example of a specification using our algebraic specification formalism.

Example: Unbounded Buffer
We describe the buffer by a specification. A buffer has one input line and one output line. It receives data messages that are to be buffered and request signals ® that indicate that a buffered data element is to be sent back. A buffer (an interactive queue) is specified by Q as described in the following formula:

$$Q: [\forall\ x \in D^*, d \in D: Q ‹ d ‹ x ‹ ® = d ‹ Q ‹ x]$$

The specification essentially expresses that if a buffer receives the data message d and afterwards a finite number of data messages x and then the request signal ® this is equivalent to a buffer that sends at first the data message d and then receives the sequence of messages x.

Note the difference between the specification above and the following one which works with an equation for the deterministic behaviors:

$$R.f \equiv \forall\ x \in D^*, d \in D: f ‹ d ‹ x ‹ ® = d ‹ f ‹ x$$

Of course, we have

$$R \Rightarrow Q$$

since

$$R.f \Rightarrow \forall\ x \in D^*, d \in D: f ‹ d ‹ x ‹ ® = d ‹ f ‹ x$$

and by the least fixpoint property of Q we obtain Q.f. However, we do not have

$$Q \Rightarrow R$$

since there exist functions f with Q.f for which we do not have

$$f ‹ d ‹ ® ‹ ® = d ‹ f ‹ ®$$

which certainly holds for all functions in R. So R is less underspecified than Q is. A specification R' more liberal than R is obtained by the declaration

$$R': [R'.f \equiv \forall\ x \in D*, d \in D: f ‹ d ‹ x ‹ ® \in d ‹ R' ‹ x]$$

The weakest predicate that fulfills this specification is equivalent to Q. However, the description of R' is again recursive, while the specification of R is not. ☐

To demonstrate the flexibility of our specification method we also describe a stack using algebraic techniques.

Example: A Reactive Stack
A stack is specified by the specification S defined by the following formula:

$$S: [\forall\ x \in D^*, d \in D: S ‹ x ‹ d ‹ ® = d ‹ S ‹ x] \qquad ☐$$

Both specifications S and Q of the examples above are highly underspecified[2], since nothing is said about the behavior in cases where the input does not conform to the given

[2]) A component with specification Q is called *underspecified* or *nondeterministic* if there exist more than one function that fulfills Q.

input pattern. Thus in both cases the behavior is not fixed when the input stream starts with a request. In such cases the component may show any behavior. We speak of a *chaotic behavior*, which is an extreme case of underspecification.

It is more difficult to specify nondeterministic systems that do not have a chaotic but a more specific but nevertheless highly nondeterministic behavior for certain input by our algebraic specification technique. For instance, for a given input, a component may react by sending one of two possible messages. Let us study this in a small example.

Example: Unreliable Lossy Buffer with Acknowledgments
If we send a message to the unreliable buffer it answers with an acknowledgment that may be positive (indicated by the signal @) or negative (indicated by the signal ®). If it is negative this expresses that the message was lost. This behavior is described by the following equations:

$$Q ‹ d \equiv ® ‹ Q \vee @ ‹ R(d)$$

$$R(d) ‹ ® \equiv ® ‹ R(d) \vee d ‹ Q$$

Here R(d) is an auxiliary specification of a component parameterized by d. We do not express any fairness assumptions that way, however. □

So far, we have worked with equations between specifications defining sets of behaviors. Sets of behaviors are used to model nondeterminism as well as underspecification. Through an execution one behavior is selected. There are two extreme techniques to select such a behavior. One extreme is to choose one behavior in advance before the first input arrives. The other extreme is to do the nondeterministic choices step by step only when reaction by output enforces such choices. This second strategy may be called *delayed choice* while the first one is called a *prophecy strategy*.

The prophecy technique can also be used in specifications. Then we write formulas that refer to single deterministic behaviors.

Example: Fairness Specified with the Help of Prophecies
We may express the fairness conditions for the unreliable buffer using prophecies by negative conditions as follows:

$$Q.f \Rightarrow \neg\forall\ i \in \mathbb{N}: f ‹ d^i \in ®^i ‹ Q$$

$$R(d).f \Rightarrow \neg\forall\ i \in \mathbb{N}: f ‹ ®^i \in ®^i ‹ R(d)$$

These two equations express that the buffer cannot always react with a reject signal to input but eventually accepts input. These negative condition can be replaced by positive conditions using an existential quantifier:

$$Q.f \Rightarrow \exists\ i \in \mathbb{N}: f ‹ d^{i+1} \in ®^i ‹ @ ‹ R(d)$$

$$R(d).f \Rightarrow \exists\ i \in \mathbb{N}: f ‹ ®^{i+1} \in ®^i ‹ d ‹ Q$$

This is, in combination with the equations above for Q and R(d), logically equivalent to the formulation of the fairness properties above with the negative conclusion. □

The problem of fairness occurs only when describing underspecified or nondeterministic components, of course. In the case of nondeterministic behaviors, we do not only use equations but also implications. We demonstrate this technique by specifying a lossy one element buffer.

Example: Specification of the Unreliable Buffer by Implication
The unreliable buffer can be specified with the help of an implication as follows:

$$\tilde{Q}\ : [\forall\ d \in D: \exists\ i, j \in \mathbb{N}: Q ‹ d^{i+1} ‹ ®^{j+1} \Leftarrow ®^i ‹ @ ‹ ®^j ‹ d ‹ \tilde{Q}\ ]\ (*)$$

Unfortunately this specification does not really express what we intend to. Let us consider $Q ‹ d^{i+1} ‹ ®^{j+1}$; for some behaviors of Q we actually get $®^i ‹ @ ‹ ®^j ‹ d$ as output for this specification but for other behaviours this is not true. So the specifying formula (∗) is too narrow. We have to use prophecies, again.

For every behavior f with Q.f we assume that there exist numbers $i, j \in \mathbb{N}$ such that

$$f ‹ d^{i+1} ‹ ®^{j+1} \in ®^{i} ‹ @ ‹ ®^{j} ‹ d ‹ Q$$

This is perhaps the most concise specification of Q if we in addition specify that every input triggers exactly one output. From this specification we can conclude the equations for Q and R(d) used in the specification above by the prefix monotonicity of the functions involved. □

Using logical operators we may compose a specification of a number of specifications expressing the required properties. Of course, all logical operators are understood to be applied pointwise.

Example: Unreliable Buffer and Driver
In the following F(d) is a parameterized auxiliary specification. We combine equational behavior specifications and prophecies as described above and get the following specification.

$$\begin{aligned} Q:\ [\exists F: F: [\forall d \in D:\ & Q ‹ d \equiv @ ‹ F(d) \vee ® ‹ Q \wedge \\ & F(d) ‹ ® \equiv \big(d ‹ Q \vee ® ‹ F(d)\big) \wedge \\ & ®^{\infty} \notin Q ‹ d^{\infty} \wedge \\ & ®^{\infty} \notin F(d) ‹ ®^{\infty}]] \end{aligned}$$

Another example for the usage of algebraic techniques is the specification of a driver component V. The driver has two input channels x and y. On x it receives data messages that it repeatedly sends on its output channel as long as it gets on its input line y the negative acknowledgment ®. If it gets a positive acknowledgment @ it continues by sending the next message received on the channel x:

$$\begin{aligned} V: [\exists W: W: [\forall m \in M:\quad & V ‹ x{:}m \equiv m ‹ W(m) \\ & W(m) ‹ y{:}® \equiv m ‹ W(m) \\ & W(m) ‹ y{:}@ \equiv V\ \ ] \end{aligned}$$

For the driver, we need no fairness assumptions. □

As we demonstrated, we can express fairness with the help of prophecies and negative conditions that rule out certain unintended behaviors. Of course, negative conditions are more difficult to handle. In particular, they may lead to inconsistencies. To avoid such inconsistencies, we may also include fairness by encoding prophecies into the state. However, this is much more difficult to express in all details and less abstract.

Example: Fairness by Prophecies as Part of the State
To express fairness conditions we add prophecy parameters to our specifications. As an example we specify the unreliable one element buffer. We specify the component P(k) that formalizes the behavior of the empty buffer. Here k is the prophecy that determines how often the buffer replies by the reject signal ® to data input. R(d, n) denotes the one-element buffer that is full. It contains the data element d. The number n determines how often the buffer responds to a read signal ® by the reject signal until it produces the data element d as output.

$$P(0) ‹ d \equiv \exists n: R(d, n)$$

$$P(n+1) ‹ d \equiv ® ‹ P(n)$$

$$R(d, 0) ‹ ® \equiv d ‹ \exists n: P(n)$$

$$R(d, n+1) ‹ ® \equiv ® ‹ R(d, n)$$

Here n is the prophecy part of the state that fixes the number of reject signals that are sent. We can hide the prophecy n in the specification P(n) by existential quantification. By

$$\exists n: P(n)$$

we denote the specification of the empty unreliable one element buffer. □

We have to choose a specific mechanism to encode fairness by prophecies in any case. This is demonstrated by the example above. We have basically three options to deal with fairness. The first two work with prophecies. We either include prophecies into the state or talk about deterministic behaviors. In both cases we use existential quantification. The third option is to work with negation.

Example: Merge component
Using the algebraic specification technique the specification of the nondeterministic merge that has two input channels x and y and one output channel and maps its input stream onto an output stream can be written as follows

$$Q \,‹\, x{:}m \,‹\, y{:}n \equiv n \,‹\, Q \,‹\, x{:}m \vee m \,‹\, Q \,‹\, y{:}n$$

This equation does not introduce any fairness conditions. We can express fairness for channels x by an additional property

$$f \in Q \Rightarrow \forall\, s \in M^{\infty}: \exists\, i \in \mathbb{N}:\ f \,‹\, x{:}m \,‹\, y{:}s \in s[1{:}i] \,‹\, m \,‹\, Q \,‹\, y{:}s[i+1{:}\infty[$$

Here for $s \in M^{\infty}$ we denote by s[i:j] the sequence of elements s_i, s_{i+1}, ..., s_j. This way nonfair behaviors are excluded. ☐

The examples above show the power and the flexibility of the algebraic specification technique for reactive systems.

6 Composition

So far we have studied basic operations on specifications that correspond to communication (input and output transitions) and to the logical connectives that allow us to combine properties. In this section, we define two basic operations on components, namely *parallel composition* and *feedback* (for a more careful treatment of these forms of composition see [Broy 87a]). These operations are sufficient to form all kinds of data flow nets. Although we consider these operations rather as combining forms for components but as connectors for combining properties, they can nevertheless be defined by logical connectors on the predicates representing the specifications.

Besides parallel composition and feedback, we introduce a hiding operator for the channels of a component. Formally, it is also an operation on predicates.

So far we have only used classical logical connectives such as conjunction, disjunction, implication and quantification besides the basic operations on components that work with message exchange as introduced above. Now we show how we may compose components by parallel composition. Given two components C_1 and C_2 with (recall that Out was defined in section 2 and denotes the set of output channels of a component) disjoint sets of output channels as indicated by the formula

$$\text{Out}(C_1) \cap \text{Out}(C_2) = \emptyset$$

we denote their *parallel composition* by the formula:

$$C_1 \parallel C_2$$

The requirement that the sets of output channels are disjoint simplifies the algebraic treatment of components. In a more sophisticated approach we can drop this restriction. Then we assume that in the parallel composition of components with overlapping sets of output channels the output on those channels is merged.

We define the input and output channel of the component obtained by parallel composition by the following equations:

$$\text{Out}(C_1 \parallel C_2) = \text{Out}(C_1) \cup \text{Out}(C_2)$$

$$\text{In}(C_1 \parallel C_2) = \text{In}(C_1) \cup \text{In}(C_2)$$

Besides parallel composition we work with feedback. It allows us to use the output of a component on its output channel y as input for its input channel x. For channels $x \in \text{In}(C)$

and $y \in Out(C)$ we write

$$\mu_x^y \ C$$

for the *feedback* of the output from the channel y to the input channel x (without hiding the channel y, but of course hiding the channel x). A graphical explanation of feedback is given in Fig. 3.

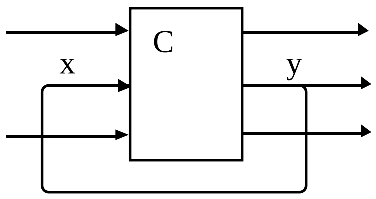

Fig. 3 Feedback for the component C from the output channel y to the input channel x

We define therefore

$$In(\mu_x^y \ C) = In(C) \backslash \{x\}$$
$$Out(\mu_x^y \ C) = Out(C)$$

We may drop an input or output line x by hiding the corresponding channel. To do that we write

$$C \backslash \{x\}$$

We define

$$In(C\backslash\{x\}) = In(C) \backslash \{x\}$$
$$Out(C\backslash\{x\}) = Out(C) \backslash \{x\}$$

Sometimes it is useful to rename channels. We write $\rho_y^x \ C$ to rename the channel y in the component described by C to x. We define

$$In(\rho_y^x \ C) = (In(C) \backslash \{y\}) \cup \{x\}$$
$$Out(\rho_y^x \ C) = (Out(C) \backslash \{y\}) \cup \{x\}$$

If $x \in In(C)$ then

$$\rho_y^x \ C$$

is a component which "contains the input channel x twice". In other words, it takes two copies of the stream x as input. This is no problem as long as the sorts of the channels coincide. If $x \in Out(C)$ then

$$\rho_y^x \ C$$

is a component with two different output channels with the same name. This leads to an inconsistency and is therefore forbidden.

The algebraic laws of renaming are very simple and straightforward: Let z, z', y, x be pairwise distinct channel identifiers and $x \notin Out(C) \cup Out(C_1) \cup Out(C_2)$:

$$\rho_y^x \ C = C \qquad \Leftarrow x \notin In(C) \cup Out(C)$$
$$(\rho_y^x \ C) \lhd z{:}m = \rho_y^x (C \lhd z{:}m)$$
$$\rho_y^x (z{:}m \lhd C) = z{:}m \ (\rho_y^x \ C)$$
$$\rho_y^x (C_1 \parallel C_2) = (\rho_y^x \ C_1) \parallel (\rho_y^x \ C_2)$$
$$(\rho_y^x \ C) \lhd x{:}m = \rho_y^x (C \lhd y{:}m) \qquad \Leftarrow x \notin In(C)$$
$$(\rho_y^x \ C) \lhd x{:}m = \rho_y^x (C \lhd x{:}m \lhd y{:}m) \qquad \Leftarrow x \in In(C)$$

$$\begin{aligned}
\rho^x_y\,(y{:}m \triangleleft C) &= x{:}m \triangleleft \rho^x_y\ C \\
\rho^x_y\ \mu^y_z\ C &= \mu^x_z\ \rho^x_y\ C \\
\rho^x_y\ \mu^z_y\ C &= \mu^z_x\ \rho^x_y\ C \\
\rho^x_y\ \mu^z_{z'}\ C &= \mu^z_{z'}\ \rho^x_y\ C
\end{aligned}$$

These rules are rather straightforward. Therefore we do not explain and motivate them.

7 The Calculus of Specifications

Algebraic formalisms can nicely be described by equational axioms. We use the following rules (let x, y and z be distinct identifiers for channels and C_1 and C_2 have disjoint sets of output channels) as axioms:

$$\begin{array}{lcl}
C \triangleleft x{:}m & = & C \qquad \Leftarrow x \notin \mathrm{In}(C) \\
(C_1 \parallel C_2) \triangleleft x{:}m & = & (C_1 \triangleleft x{:}m) \parallel (C_2 \triangleleft x{:}m) \\
(x{:}m \triangleleft C_1) \parallel C_2 & = & x{:}m \triangleleft (C_1 \parallel C_2) \\
C_1 \parallel C_2 & = & C_2 \parallel C_1 \\
(\mu^y_x\ \ C) \triangleleft z{:}m & = & \mu^y_x\,(C \triangleleft z{:}m) \\
\mu^y_x\,(z{:}m \triangleleft C) & = & z{:}m \triangleleft \mu^y_x\ \ C \\
\mu^y_x\ \ y{:}m \triangleleft C & = & y{:}m \triangleleft \mu^y_x\,(C \triangleleft x{:}m) \\
\mu^x_y(Q \parallel D) & = & (\mu^x_y\ \ Q) \parallel D \text{ if } x \notin \mathrm{In}(D) \text{ and } y \notin \mathrm{Out}(D) \\
C \triangleleft x{:}m_1 \triangleleft z{:}m_2 & = & C \triangleleft z{:}m_2 \triangleleft x{:}m_1 \\
x{:}m_1 \triangleleft z{:}m_2 \triangleleft C & = & z{:}m_2 \triangleleft x{:}m_1 \triangleleft C
\end{array}$$

The last two equations are called the rules of *asynchrony*. There is no causal relationship for messages on different channels. These rules of asynchrony have to be dropped as laws if we consider timed streams where the timing of the messages and thus the relative timing of messages on different input streams can be expressed.

The equations define a process algebra. We do not give a more careful analysis of this algebra. Such an analysis can be found in [Broy, Stefanescu 95]. The algebraic laws give an axiomatization of our operations in terms of communication actions. They allow us to do proofs about specifications. Furthermore they can be used as a basis for an operational semantics and therefore for an interpreter that executes such specifications.

8 Relation to State Transition Diagrams

State transition systems are a well-known and well-accepted concept for describing the behavior of components in systems engineering (see for instance [Lynch, Tuttle 87] and [Lynch, Stark 89]). Every algebraic equation or logical implication as introduced above can be seen as the description of a transition of a state machine. It is interesting that in a state machine, equations and implications are represented both by one arc. It is helpful in the understanding of systems to visualize equations by clear means. Therefore we represent for instance the transition equation

$$Q \triangleleft m \equiv y_1 \triangleleft Q_1 \vee y_2 \triangleleft Q_2$$

by the multitarget arc as shown in Fig. 4. Vice versa we may translate diagrams into equations by this correspondence.

Using this technique of interacting state transition diagrams we may translate algebraic specifications of reactive systems into state transition diagrams and vice versa. The translation of state transition diagrams to algebraic equations is straightforward according to the rules mentioned above.

In diagrams we use states represented by little circles that correspond to specifications. Therefore we label states by specifications. We label arrows by input and output events corresponding to channels and messages.

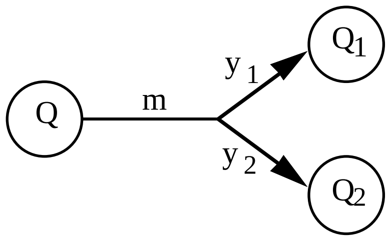

Fig. 4 Multitarget Arc

It is more common to represent the transitions by two independent arcs in this case as shown in Fig. 5.

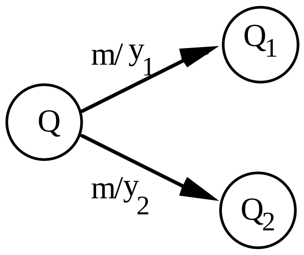

Fig. 5 State Transition Diagram with Two Transitions

This state transition diagram corresponds rather to two specifying formulas that have to be written by implications instead of equations

$$Q ‹ m \Leftarrow y_1 ‹ Q_1$$

$$Q ‹ m \Leftarrow y_1 ‹ Q_2$$

Note, however, that a state transition diagram always includes a closed world assumption corresponding to the statement "and no further transitions".

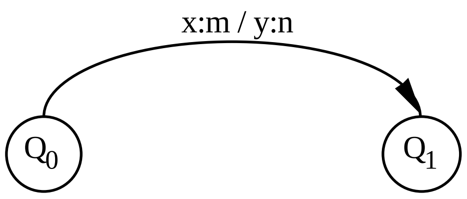

Fig. 6 State Transition

The state transition given in Fig. 6 corresponds to the specification

$$Q_0 ‹ x{:}m \Leftarrow y{:}n ‹ Q_1$$

or expressed with the help of prophecies:

$$\forall f_1{:}\ Q_1.f_1 \Rightarrow \exists f_0{:}\ Q_0.f_0 \wedge f_0 ‹ x{:}m = y{:}n ‹ f_1$$

If we know, in addition, that there is no other arrow labeled by x:m as input we even work with the equation

$$Q_0 ‹ x{:}m = y{:}n ‹ Q_1$$

This expresses basically the closed world assumption that for the input m on the channel x only one reaction is possible.

Example: One-element Buffer
The algebraic specification of the one-element buffer is given in section 5 by the equations

$$Q \text{ ‹ } d \equiv R(d)$$

$$R(d) \text{ ‹ ®} \equiv d \text{ ‹ } Q$$

This corresponds to the state transition diagram given in Fig. 7. ☐

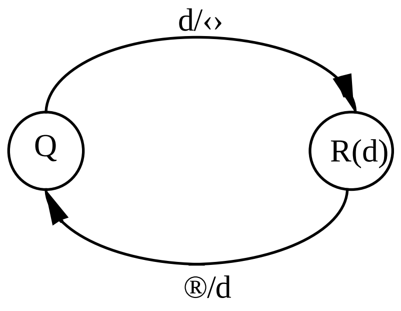

Fig. 7 Transition Diagram

If there are several transitions labeled with the input pattern x:m we may translate the transitions into the equation:

$$Q_0 \text{ ‹ } x{:}m \equiv \bigvee_{i=1}^{k} y_i{:}m_i \text{ ‹ } Q_i \qquad (*)$$

If for an input pattern a transition is missing, the behavior is unspecified for such an input. Therefore it is not appropriate to interpret transitions by the formula

$$\bigwedge_{i=1}^{k} (Q_0 \text{ ‹ } x{:}m \Leftarrow y_i.m_i \text{ ‹ } Q_i)$$

Since in the case of this formula the weakest specification for Q_0 is true. This does not hold for the interpretation above. This illustrates that a state transition description always corresponds to a number of implications and a closed world assumption. A more detailed treatment is given in [Broy 96].

9 Incorporation of Real Time

In this section, we extend our approach to the specification of real time properties by using a specific message √ called a time tick. Since we assume a global time for all channels we assume that a time tick message comes "at the same time" on all channels. For a specification Q we write

$$Q \text{ ‹ } \surd$$

to denote the behavior described by the predicate Q provided in the first time interval (the end of which is indicated by the time tick √) there do not arrive any messages. In analogy, we use the notation

$$\surd \text{ ‹ } Q$$

If we write

$$Q \text{ ‹ } c{:}m \text{ ‹ } \surd$$

this denotes the behavior of the component where in the first interval only on channel c the message m is received and no further messages.

Example: Timer
A timer is a component T(k) that receives the messages set(n) with $k, n \in \mathbb{N}$ to set the timer and the message ® to reset the timer. T(0) is the unset timer. If the time set is over, the timer sends a timeout message:

$$T(0) \,‹\, \surd = \surd \,‹\, T(0)$$

$$T(n) \,‹\, set(k) = T(k)$$

$$T(1) \,‹\, \surd = \surd \,‹\, timeout \,‹\, T(0)$$

$$T(k+1) \,‹\, \surd = \surd \,‹\, T(k) \qquad \Leftarrow k > 0$$

$$T(k) \,‹\, ® = T(0) \qquad \square$$

Since we assume that time tick signal √ arrives on all channels the rule of asynchrony does not hold for messages separated by √. In other words, in general, we have

$$Q \,‹\, x{:}m_1 \,‹\, \surd \,‹\, y{:}m_2 \neq Q \,‹\, y{:}m_2 \,‹\, \surd \,‹\, x{:}m_1$$

even in cases where x and y denote different channels. We show by a second example how to specify time dependent components.

Example: Time Out
A component with one input channel and one output channel that transmits a message provided it arrives twice before a timeout occurs is specified by the following equations (let $m \neq \surd$)

$$Q \,‹\, m \,‹\, m \,‹\, \surd = \surd \,‹\, m \,‹\, Q$$

$$Q \,‹\, m \,‹\, \surd = Q \,‹\, \surd$$

$$Q \,‹\, \surd = \surd \,‹\, Q$$

So far nothing is said what happens if other patterns of input occur. They are covered by (let $m_1, m_2, m_3 \neq \surd$) the following equations

$$Q \,‹\, m_1 \,‹\, m_2 \,‹\, \surd = \surd \,‹\, ® \,‹\, Q \qquad \Leftarrow m_1 \neq m_2$$

$$Q \,‹\, m_1 \,‹\, m_2 \,‹\, m_3 = Q \,‹\, m_1 \,‹\, m_3$$

The second equation indicates that a message is also forwarded if the last message that is received before the time tick repeats it. □

Of course we need additional assumptions about the time flow to get a proper formalization of time properties. We do not go deeper into the question of formalizing time. For a more careful treatment see [Broy 83] and [Broy 93].

10 Conclusion

Algebraic equations for components provide a powerful and useful technique for the specification of systems. They can be used in addition to the specification techniques of FOCUS that are based on predicate logic. They complement and extend these specification techniques that are based on classical higher order predicate logic. They, on the other hand, are the mathematical basis for state transition diagram descriptions.

Algebraic equations for specifications of reactive systems can often be interpreted as recursive definitions of systems. This gives some operational flavor and allows us to express and even to generate algorithmic executions. In addition, algebraic techniques form a bridge to state transition systems and state transition diagrams.

The approach that we have introduced above can be extended to treat dynamic systems with channel creation and channel deletion and with component ("object") creation and deletion along the lines of [Milner et al. 92], [Grosu 94], [Broy 95] and [Grosu et al. 95].

FOCUS provides a powerful theory and method of refinement (see [Broy, Stølen 94]) which has a very algebraic flavor that can be combined with the algebraic specification techniques.

Appendix A0: Mathematical Basis

Throughout this paper interactive systems are supposed to communicate asynchronously through unbounded FIFO channels. Streams are used to denote histories of communications on channels. Given a set M of messages, a *stream* over M is a finite or infinite sequence of elements from M. By M^* we denote the finite sequences over M. M^* includes the empty stream that is denoted by ‹›.

By M^∞ we denote the infinite streams over the set M. M^∞ can 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

$$M^\omega = M^* \cup M^\infty.$$

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

A classical operation on streams is the *concatenation* that we denote by ˆ. The concatenation is a function that takes two streams (say s and t) and produces a stream sˆt as result, starting with the stream s and continuing with the stream t. Formally the concatenation has the following functionality:

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

If the stream s is infinite, then concatenating the stream 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 any message $m \in M$ we denote by ‹m› the one element stream consisting of the element m.

On the set M^ω of streams we define a *prefix ordering* ⊑. We write s ⊑ t for streams s and t if s is a *prefix* of t. Formally we have

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

The prefix ordering defines a partial ordering on the set M^ω of streams. If s ⊑ t, then we also say that s is an *approximation* of t. The set of streams ordered by ⊑ is complete in the sense that every directed set $S \subseteq M^\omega$ of streams has a *least upper bound* denoted by lub 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. 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.

A *stream processing function* is a function

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

that is *prefix monotonic* and *continuous*. Stream processing functions model data flow components (see [Kahn, MacQueen 77], [Park 80] and [Park 83]). 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 for the function application f.x instead of f(x). A prefix monotonic function f is called *prefix continuous*, if for all directed sets $S \subseteq M^\omega$ of streams we have

$$f.\text{lub } S = \text{lub } \{f.s: s \in S\} .$$

If a function is prefix continuous, then its results for infinite input can be already determined from its results on all finite approximations of the input.

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

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

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

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. ACM Transactions on Software Engineering and Methodology 2:1, Januar 1993, 1-46

[Broy 95]
M. Broy: Equations for Describing Dynamic Nets of Communicating Systems. In: E. Astesiano, G. Reggio, A. Tarlecki (eds): Recent Trends in Data Types Specification, 10th Workshop on Specification of Abstract Data Types joint with the 5th COMPASS Workshop, S.Margherita, Italy, May/June 1994, Lecture Notes in Computer Science 906, Springer 1995, 170-187

[Broy 96]
M. Broy: The Specification of System Components by State Transition Diagrams. Technical Memo 1996

[Broy, Stefanescu 95]
M. Broy, G. Stefanescu: Algebra of Stream Processing Functions. Forthcoming paper

[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

[Dybier, Sander 88]
P. Dybier, H. Sander: A Functional Programming Approach to the Specification and

Verification of Concurrent Systems. Chalmers University of Technology and University of Göteborg, Department of Computer Sciences 1988

[FOCUS 94]
M. Broy, M. Fuchs, T. F. Gritzner, B. Schätz, K Spies, K Stølen: Summary of Case Studies in FOCUS - a Design Method for Distributed Systems. Technische Universität München, Institut für Informatik, TUM-I9423, Juni 1994

[Grosu 94]
R. Grosu: A Formal Foundation for Concurrent Object Oriented Programming. Ph. D. Thesis, Technische Universität München, Fakultät für Informatik, 1994

[Grosu et al. 95]
R. Grosu, K. Stølen, M. Broy: A Denotational Model for Mobile Data Flow Networks. To appear

[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

[Milner et al. 92]
R. Milner, J. Parrow, D. Walker: A Calculus of Mobile Processes. Part i + ii, Information and Computation, 100:1 (1992) 1-40, 41-77

[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

[SDL 88]
Specification and Description Language (SDL), Recommendation Z. 100. Technical Report, CCITT 1988