JOURNAL OF COMPUTER AND SYSTEM SCIENCES 34, 236–265 (1987)

# Specification and Top-Down Design of Distributed Systems

MANFRED BROY*

*Fakultät für Mathematik und Informatik, Universität Passau, Postfach 25 40, 8390 Passau, Federal Republic of Germany*

Received October 1985; revised April 1986

Stream-processing functions and recursively defined streams provide an excellent semantic model for the abstract representation of systems ("networks") of nondeterministic concurrent communicating agents. Based on this model an "algebraic" (equation-oriented) formalism for the specification of such networks as stream-processing functions is suggested. This way a fully modular ("compositional") methodology for the specification and the design of distributed systems and their components is obtained. Concepts of correctness are defined and rules of inference are discussed that help to transform such specifications into a network of communicating agents. A combinatorial ("functional") notation for the sequential and parallel composition as well as for a feedback operator for those agents is introduced. © 1987 Academic Press, Inc.

## 1. INTRODUCTION

For the top-down design of concurrent communicating ("distributed") systems a specification formalism is an indispensible requisite. Only if one is able to give modular specifications, i.e., self-contained abstract specifications also for behaviours of subcomponents of a distributed system, then the decomposition of the system can be done properly and the subcomponents can be developed and verified separately. In such specifications we are not interested in the description of the internal structure of (components of) concurrent communicating systems that could be described, for instance, by event structures (cf. [24]) but rather in the "input/output" behaviour of these systems ("extensional behaviour").

In the following a simple language formalism is suggested that can be used as a formal framework for the specification, design, and verification of concurrent, communicating systems and their components. It is based on a semantic model for concurrent, communicating systems and is an attempt to combine ideas and concepts from denotational semantics for concurrent systems (cf. [4]), programming logic (such as temporal logic, cf. [21]), and predicative specifications, cf. [14], algebraic specifications (cf. [11]), program transformation (cf. [9]), and functional (multi-) programming (cf. [2]). Of course there are numerous approaches and proposals for

* This research was partly supported by the DFG Project Br 887 Parallelism.



0022-0000/87 $3.00

the treatment of concurrent and distributed systems, their description, analysis, and specification. For lack of space we do not give comprehensive references but rather refer to the respective literature (cf. also the references in the other papers on concurrency in this volume).

## 2. A Specification Formalism

In this section a specification formalism is introduced that allows specifying communicating agents with a finite tuple of input lines and a finite tuple of output lines. We start by giving some examples, then give a formal syntax, and finally define the semantics.

### 2.1. *Specifications of Communicating Agents*

A communicating agent has $n$ input lines and $m$ output lines, where $m$ and $n$ are arbitrary natural numbers (including 0). On every input line a finite or infinite sequence of data is transmitted to the agent and on every output line a finite or infinite sequence of data is generated by the agent. The input lines and output lines have internal (local) names that are used in a predicate for expressing the relationship between the input and output.

#### 2.1.1. *First Examples for Specifications of Agents*

We first give the example of a very simple agent called *store* which nevertheless shows the full power of the method.

**agent** store = **input stream data** $d$, **stream bool** $b$, **output stream data** $r$,

**first** $b$ = true $\Rightarrow$ $r$ = store(**rest** $d$, **rest** $b$),
**first** $b$ = false $\Rightarrow$ $r$ = **first** $d$ & store($d$, **rest** $b$) **end**

This specification defines an agent with two input lines and one output line; it can be graphically represented by Fig. 1. The identifiers $d$, $b$, $r$ for the input lines and output lines are only internal ("local" or "bound") names and **not** relevant to the outside. The operator & puts an element in front of a sequence, **first** $s$ returns the first element of a sequence $s$, **rest** $s$ returns the sequence $s$ without the first element.

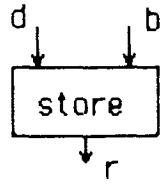


FIGURE 1

According to the specification taking, for instance, the sequences

$$d = 1 \,\&\, 2 \,\&\, 3 \,\&\, 4 \,\&\, \cdots$$
$$b = \text{false} \,\&\, \text{false} \,\&\, \text{true} \,\&\, \text{false} \,\&\, \text{true} \,\&\, \text{false} \,\&\, \cdots$$

as input the agent store produces the output sequence

$$r = 1 \,\&\, 1 \,\&\, 2 \,\&\, 3 \,\&\, \cdots$$

The agent store may be seen as a memory cell where $b$ can be interpreted as the read/write command sequence: If the input on input line $b$ is true, then a new data value on the input line $d$ is stored, if the input on input line $b$ is false, then this is interpreted as a read command: the current input on $d$ is copied as output.

As a second example the agent schedule is specified:

**agent** schedule =
**input stream data** $a$, **stream data** $b$, **stream bool** $s$, **output stream data** $r$,

$$\textbf{first}\ s = \text{true} \Rightarrow r = \textbf{first}\ a \,\&\, \text{schedule}(\textbf{rest}\ a, b, \textbf{rest}\ s),$$
$$\textbf{first}\ s = \text{false} \Rightarrow r = \textbf{first}\ b \,\&\, \text{schedule}(a, \textbf{rest}\ b, \textbf{rest}\ s) \quad \textbf{end}$$

The agent schedule receives three input streams: two data streams and one stream of booleans. The data streams are merged according to the boolean values in the third input stream. With the input

$$a = 0 \,\&\, 2 \,\&\, 4 \,\&\, \cdots$$
$$b = 1 \,\&\, 3 \,\&\, 5 \,\&\, \cdots$$
$$s = \text{true} \,\&\, \text{true} \,\&\, \text{false} \,\&\, \text{false} \,\&\, \text{true} \,\&\, \text{false} \,\&\, \cdots$$

the agent schedule produces

$$r = 0 \,\&\, 2 \,\&\, 1 \,\&\, 3 \,\&\, 4 \,\&\, 5 \,\&\, \cdots$$

Another example is the agent switch:

**agent** switch = **input stream data** $s$, **stream bool** $b$, **output stream data** $r1$, $r2$,

$$\forall\ \textbf{stream data}\ d1, d2\colon (d1, d2) = \text{switch}(\textbf{rest}\ s, \textbf{rest}\ b) \Rightarrow$$
$$(\textbf{first}\ b = \text{true} \Rightarrow (r1 = \textbf{first}\ s \,\&\, d1 \wedge d2 = r2)) \wedge$$
$$(\textbf{first}\ b = \text{false} \Rightarrow (d1 = r1 \wedge r2 = \textbf{first}\ s \,\&\, d2)) \quad \textbf{end}$$

The agent switch produces two output streams by sending the input on its input line $s$ either to the left or to the right output line depending on the boolean input in its input line $b$. One may prove

$$\text{switch}(s, b) = (d1, d2) \Rightarrow \text{schedule}(d1, d2, b) = s.$$

An agent even may have no input lines at all. An example reads

**agent** onestream = **output stream nat** $r$, $r = 1 \,\&\, r$ **end**

The agent onestream produces an infinite stream of 1's:

$$r = 1 \,\&\, 1 \,\&\, 1 \,\&\, \cdots$$

We can specify also agents that perform arithmetic operations on its input streams. A simple example is

**agent** addstream = **input stream nat** $a$, **stream nat** $b$, **output stream nat** $r$,

$r =$ (**first** $a +$ **first** $b$) & addstream(**rest** $a$, **rest** $b$) **end**

This agent adds the sequences of input streams elementwise; if

$$a = 0 \,\&\, 2 \,\&\, 4 \,\&\, \cdots$$

$$b = 1 \,\&\, 3 \,\&\, 5 \,\&\, \cdots$$

then one obtains

$$r = 1 \,\&\, 5 \,\&\, 9 \,\&\, \cdots$$

Agents may be specified based on other agents:

**agent** natstream =

**output stream nat** $r$, $r = 0$ & addstream($r$, onestream) **end**

The agent natstream produces the infinite stream of the natural numbers:

$$r = 0 \,\&\, 1 \,\&\, 2 \,\&\, \cdots$$

So far the specified agents looked deterministic, i.e., uniquely specified. They can be understood to describe precisely functions from the (tuples of) input streams to the (tuples of) output streams.

2.1.2. *Nondeterministic Agents*

For describing distributed systems it is important that nondeterministic agent specifications may be written in the formalism, too. Consider the agent infinite that is specified as

**agent** infinite = **output stream bool** $r$, $r =$ true & $r \wedge r =$ false & $r$ **end**

The agent infinite has two possible output streams:

$$r = \text{true} \,\&\, \text{true} \,\&\, \cdots \quad \text{or} \quad r = \text{false} \,\&\, \text{false} \,\&\, \cdots$$

The possibility of writing nondeterministic agent specifications leads to the possibility (and the problems) of the semantic interpretation of nondeterministic expressions. For instance, with the specification above, the agent infinite produces nondeterministically one of the two infinite sequences. Using this agent in an expression leads to the problem of nondeterministic terms and their meaning. Nondeterministic terms in equations bring a number of complications. What does it mean to write

$$t1 = t2$$

for nondeterministic terms $t1$ and $t2$? Does it mean that $t1$ and $t2$ must stand for the same set of possible values? To show some of the subtle differences let us consider the specification of the agent any which looks very similar to the agent infinite.

**agent** any = **output stream bool** $r$, $r \leftarrow$ true & any( ) $\vee$ $r \leftarrow$ false & any( )
**end**

Note that we use the arrow "$\leftarrow$" here instead of the "=" sign. The formula

$$t1 \leftarrow t2$$

indicates that for the possibly nondeterministic expressions $t1$ and $t2$ the set of values of $t1$ is included in the set of values of $t2$. If both $t1$ and $t2$ are deterministic, then $t1 \leftarrow t2$ of course is equivalent to $t1 = t2$.

We will choose a semantic interpretation that gives a different meaning to the agent any than to the agent infinite. According to this interpretation the agent any produces any infinite sequence of boolean values.

Another even more famous and more important example for nondeterministic agents is the agent merge:

**agent** merge = **input stream data** $a$, **stream data** $b$, **output stream** $r$

$$\exists \textbf{ stream bool } s\colon s \leftarrow \text{any}( ) \wedge r = \text{schedule}(a, b, s)$$

**end**

The agent merge is highly nondeterministic. For instance (assuming for a moment that **data** is **bool**), with the infinite streams

$$a = \text{true \& true \& } \cdots$$

$$b = \text{false \& false \& } \cdots$$

the term merge($a$, $b$) may stand for **any** infinite stream of boolean values. A nondeterministic agent is understood as a specification representing a predicate on a set of functions.

After having given a number of simple examples the syntax and the semantics of the specification language is now formally defined.

2.2. *Syntactic Form of Agent Specifications*

In this section a syntax for specifications of communicating agents is given in BNF-style notation.

*Syntax*

⟨system⟩::= ⟨agent-spec⟩*

⟨agent-spec⟩::= {**rec**} **agent** ⟨agent-id⟩ =
{**input** ⟨dec-tuple⟩,}{**output**⟨dec-tuple⟩,}
⟨formula⟩ **end**

⟨dec-tuple⟩ ::= {⟨dec⟩,}*

⟨dec⟩::= {**stream**} ⟨sort⟩⟨id⟩ { , ⟨id⟩}*

⟨formula⟩::= ⟨exp⟩ = ⟨exp⟩ | ⟨exp⟩ ← ⟨exp⟩ | ⟨function⟩ = ⟨function⟩ |
⟨function⟩ ← ⟨function⟩ | (⟨formula⟩) | true | false |
⟨formula⟩{ ∧ | ∨ | ⇒ |, }[1] ⟨formula⟩ | ¬ ⟨formula⟩ |
{∃|∀}[1] ⟨dec-tuple⟩ ⟨dec⟩: ⟨formula⟩ |
{∃|∀}[1] ⟨arity⟩ ⟨spf-id⟩ {, ⟨arity⟩ ⟨spf-id⟩}: ⟨formula⟩
{○, ◇, □}[1] ⟨id⟩{, ⟨id⟩}*: ⟨formula⟩

⟨arity⟩ ::= **funct**({{**stream**} ⟨sort⟩{, {**stream**} ⟨sort⟩}*) {**stream**} ⟨sort⟩

⟨exp⟩ ::= ⟨id⟩ | {**first** | **rest** | **isempty**}[1] ⟨exp⟩ | ⟨exp⟩ & ⟨exp⟩ |
⟨function⟩({⟨exp⟩{, ⟨exp⟩}*}) |
**if** ⟨exp⟩ **then** ⟨exp⟩ **else** ⟨exp⟩ **fi** | (⟨exp⟩{, ⟨exp⟩}*)

⟨function⟩ ::= ⟨agent-id⟩ | ⟨primitive function⟩ | ⟨spf-id⟩ |
({⟨dec⟩ {⟨dec⟩, }*}) {**stream**} ⟨sort⟩: ⟨exp⟩

In this syntax it is assumed that we have the following disjoint sets of identifiers:

- a set ⟨id⟩ of identifiers for data or streams of data,
- a set ⟨spf-id⟩ of identifiers for stream-processing functions,
- a set ⟨agent-id⟩ of identifiers for agents,
- a set ⟨primitive functions⟩ of identifiers for primitive (given) agents.

Of course a number of context conditions (such as well-formedness or type-correctness of terms) have to be presupposed for ensuring that an agent specification is meaningful. For convenience and lack of space these quite obvious context conditions are not given explicitly.

2.3. *Semantics of Agent Specifications*

In this section a semantic model is introduced, based on an algebra of primitive data and functions that are assumed given (such as booleans or arithmetic functions). The semantic model consists essentially of stream processing functions. Then the agent specification formalism is related to this semantic model.

2.3.1. *Streams and Stream-processing Functions*

As one of the most fundamental domains for communicating programs one may consider the *domain of streams* (cf. [2]). Given a flat domain $A^\perp$ (i.e., a set $A$, with $\perp \notin A$, $A^\perp = A \cup \{\perp\}$, ordered by $a1 \sqsubseteq a2$ iff $a1 = a2 \vee a1 = \perp$ for $a1, a2 \in A^\perp$) the domain STREAM($A$) of streams over $A$ is defined by

$$\text{STREAM}(A) = (A^* \times \{\perp\}) \cup A^* \cup A^\infty,$$

where $A^*$ denotes the finite sequences (words) over $A$, and $A^\infty$ denotes the infinite sequences (words) over $A$. For $s1, s2 \notin \text{STREAM}(A)$, a partial ordering is defined by

$$s1 \sqsubseteq s2 \text{ iff } s1 = s2 \text{ or } \exists\, s3 \in A^*,\ s4 \in \text{STREAM}(A): s1 = s3 \circ \langle \perp \rangle \wedge s2 = s3 \circ s4.$$

Here "$\circ$" denotes the usual concatenation where for $s \in A^\infty$ we define $s \circ s' = s$; for all $a \in A$ by $\langle a \rangle$ we denote the one-element sequence consisting of $a$. By $\mathscr{E}$ we denote the empty sequence.

With these definitions (STREAM($A$), $\sqsubseteq$) forms an algebraic domain where $(A^* \times \{\perp\}) \cup A^*$ is the set of finite elements, $A^\infty$ the infinite ones, $A^* \times \{\perp\}$ the partial ones, and $A^* \cup A^\infty$ the total ones. The stream $\langle \perp \rangle$ represents the least element.

Streams can be used for representing the sequence of communications of a program, for instance, the output on a specific channel. For communicating programs one has to distinguish two forms of nontermination: nontermination with infinite output and nontermination without any further output. The first is represented by an infinite stream, the second by a finite stream ending with the $\perp$-symbol. In this case one may speak of *divergence*. Of course, for the non-terminating programs the output may be an infinite sequence; for immediately diverging programs the output is $\langle \perp \rangle$.

The following four basic functions are used on streams:

$$\begin{aligned}
&\text{ap} && : A^\perp \times \text{STREAM}(A) \to \text{STREAM}(A)\\
&\text{rest} && : \text{STREAM}(A) \to \text{STREAM}(A)\\
&\text{first} && : \text{STREAM}(A) \to A^\perp\\
&\text{isempty} && : \text{STREAM}(A) \to \mathbb{B}^\perp
\end{aligned}$$

defined by

$$\text{ap}(a, s) = \begin{cases} \langle a \rangle \circ s & \text{if } a \in A,\ s \in \text{STREAM}(A) \\ \langle \perp \rangle & \text{otherwise.} \end{cases}$$

For ap$(a, s)$ we often write $a \,\&\, s$. Note that ap is a nonstrict function: the result of applying ap may be different from the least element $\langle \bot \rangle$ even if the second argument is $\langle \bot \rangle$. However, if the first argument is the least element of the domain (is $\bot$) then the result is the least element (is $\langle \bot \rangle$). The function ap is left-strict. The stream $a \,\&\, s$ can be seen as a sequence of communicated data (for instance, output). As soon as $a$ is $\bot$, then there cannot be any defined output afterwards: $\bot \,\&\, s = \langle \bot \rangle$. So the definition of ap mirrors the simple fact of communicating processes that after divergence there cannot be any further output. Note that STREAM$(A)$ is not closed with respect to usual concatenation since for streams $s \in A^* \times \{\bot\}$ and $s' \in A^* \setminus \{\mathscr{E}\}$: $s \circ s' \notin \text{STREAM}(A)$.

Let $a \in A$, $s \in \text{STREAM}(A)$, then the functions rest, first, isempty are defined by the equations

$$\begin{aligned} &\text{rest}(a \,\&\, s) = s, && \text{rest}(\mathscr{E}) = \text{rest}(\langle \bot \rangle) = \langle \bot \rangle, \\ &\text{first}(a \,\&\, s) = a, && \text{first}(\mathscr{E}) = \text{first}(\langle \bot \rangle) = \bot, \\ &\text{isempty}(a \,\&\, s) = \text{false}, && \text{isempty}(\mathscr{E}) = \text{true}, \qquad \text{isempty}(\langle \bot \rangle) = \bot. \end{aligned}$$

One simply proves that the functions ap, rest, first, and isempty are monotonic and continuous.

For obvious reasons, streams can be considered as one of the most fundamental domains when dealing with systems of communicating processes. For procedural concurrent programs with shared memory one may consider streams of states; for processes with explicit communication primitives one can think of streams of communication actions (cf. [3, 67]).

For giving meaning to agent specifications, a fixed set DATA of atomic data objects is assumed. For writing examples, we assume $\mathbb{N} \subseteq \text{DATA}$ and true, false $\in$ DATA. In a more complete specification framework, one may assume some abstract data type specification method for specifying the atomic data objects.

The set $D$ of all objects on which agents operate is defined by

$$D =_{\text{def}} \text{DATA}^{\bot} \cup \text{STREAM}(\text{DATA}).$$

An agent with $m$ input lines and $n$ output lines is a continuous mapping. The set of those continuous mappings is defined by

$$\text{AGENT}_n^m = [D^m \to D^n].$$

The set of all agents is defined by

$$\text{AGENT} = \{f \in [D^m \to D^n] : n, m \in \mathbb{N}\}.$$

Having fixed the data universe $D$ and the universe of agents $\text{AGENT}_n^m$, now meaning can be assigned to the specifications.

2.3.2. *Assigning Meaning to Agent Specifications*

For assigning meaning to agent specifications the well-known technique of environments is used:

$$\mathrm{ENV} =_{\mathrm{def}} ((\langle \mathrm{id} \rangle \cup \langle \text{spf-id} \rangle \cup \langle \text{agent-id} \rangle) \to (D^* \cup \mathrm{AGENT} \cup P(\mathrm{AGENT}) \setminus \{\varnothing\})).$$

An agent environment associates with every agent identifier a set of agents, with every data identifier a stream or a data object, and with every stream-processing function identifier a stream processing function.

Note that for every particular reasons (cf. concluding remarks) we have chosen to associate with every agent identifier a set of continuous stream processing functions instead of using functions mapping (tuples of) streams into sets of (tuples of) streams.

As usual the updating of environments $\sigma$ is denoted by $\sigma[d/x]$:

$$\sigma[d/x](y) = \begin{cases} d & \text{if } x = y \\ \sigma(y) & \text{otherwise.} \end{cases}$$

The meaning of an expression is defined by the semantic function

$$B\colon \langle \mathrm{exp} \rangle \to \mathrm{ENV} \to P(D^*).$$

We write $B_\sigma[E]$ for $B(E)(\sigma)$. Note that due to the fact that agent identifiers stand for sets of stream processing functions, this particular semantic model is chosen.

$$B_\sigma[f(E_1, \ldots, E_n)] \underset{\mathrm{def}}{=} \{h(h_1, \ldots, h_n)\colon h_1 \in B_\sigma[E_1] \wedge \cdots \wedge h_n \in B_\sigma[E_n] \wedge h \in F_\sigma[f]\}.$$

The semantic function $F_\sigma$ will be defined below.

$$B_\sigma[\textbf{rest } E] \underset{\mathrm{def}}{=} \{\mathrm{rest}(g)\colon g \in B_\sigma[E]\}$$

$$B_\sigma[\textbf{first } E] \underset{\mathrm{def}}{=} \{\mathrm{first}(g)\colon g \in B_\sigma[E]\}$$

$$B_\sigma[\textbf{isempty } E] \underset{\mathrm{def}}{=} \{\mathrm{isempty}(g)\colon g \in B_\sigma[E]\}$$

$$B_\sigma[E1 \,\&\, E2] \underset{\mathrm{def}}{=} \{ap(h1, h2)\colon h1 \in B_\sigma[E1] \wedge h2 \in B_\sigma[E2]\}$$

$$B_\sigma[x] \underset{\mathrm{def}}{=} \{\sigma(x)\} \qquad \text{for } x \in \langle \mathrm{id} \rangle$$

$$B_\sigma[\textbf{if } E0 \textbf{ then } E1 \textbf{ else } E2 \textbf{ fi}] \underset{\mathrm{def}}{=}$$

$$\{\mathrm{if}(h0, h1, h2)\colon h0 \in B_\sigma[E0] \wedge h1 \in B_\sigma[E1] \wedge h2 \in B_\sigma[E2]\},$$

where if: $D^3 \to D$ is defined by

$$\text{if}(d0, d1, d2) = \begin{cases} d1 & \text{if } d0 = \text{true} \\ d2 & \text{if } d0 = \text{false} \\ \perp & \text{otherwise} \end{cases}$$

$$B_\sigma[(E_1,..., E_n)] = \underset{\text{def}}{=} \{(h_1,..., h_n): h_1 \in B_\sigma[E_1] \wedge \cdots \wedge h_n \in B_\sigma[E_n]\}.$$

The meaning of a term denoting a function is defined by the semantic mapping

$$F: \langle \text{function} \rangle \to \text{ENV} \to P(\text{AGENT}),$$

where

$$F_\sigma[f] \underset{\text{def}}{=} \{\sigma(f)\} \quad \text{for} \quad f \in \langle \text{spf-id} \rangle$$

$$F_\sigma[f] \underset{\text{def}}{=} \sigma(f) \quad \text{for} \quad f \in \langle \text{agent-id} \rangle$$

$$F_\sigma[(m_1 x_1,..., m_n x_n): E] \underset{\text{def}}{=} \{g \in [M_1 \times \cdots \times M_n \to M_{n+1}]:$$

$$\forall x_1 \in M_1,..., x_n \in M_n: g(x_1,..., x_n) \in B_\sigma[E]\}.$$

Here $M_1,..., M_n$ are assumed to denote the carrier sets associated with the sorts $m_1,..., m_n$, and $M_{n+1}$ is assumed to be the carrier set associated with the sort of the expression $E$.

The meaning of formulas is defined by the semantic function

$$M: \langle \text{formula} \rangle \to \text{ENV} \to \mathbb{B}.$$

Here $\mathbb{B}$ denotes the set {true, false} of logical values. We write $M_\sigma[H]$ for $M(H)(\sigma)$. For expressions $E1$ and $E2$ we define

$$M_\sigma[E1 = E2] = (B_\sigma[E1] = B_\sigma[E2])$$

$$M_\sigma[E1 \leftarrow E2] = (B_\sigma[E1] \subseteq B_\sigma[E2]).$$

For terms $t1$ and $t2$ denoting functions we define

$$M_\sigma[t1 = t2] = (F_\sigma[t1] = F_\sigma[t2])$$

$$M_\sigma[t1 \leftarrow t2] = (F_\sigma[t1] \in F_\sigma[t2]).$$

For boolean expressions $E$ we often write just $E$ instead of $E = \text{true}$,

$$M_\sigma[H1 \wedge H2] \underset{\text{def}}{=} M_\sigma[H1] \wedge M_\sigma[H2].$$

Analogous definitions are assumed for the remaining logical connectives. We often write "," instead of " $\wedge$ ,"

$$M_\sigma[\forall \mathbf{m}\, x : H] \underset{\text{def}}{=} \forall d \in R: M_{\sigma 1}[H], \text{ where } \sigma 1 = \sigma[d/x],$$

where $R$ denotes the subset of $D$ that is indicated by the sort **m**:

$$M_\sigma[\exists\, \mathbf{m}\, x\colon H] \underset{\text{def}}{=} M_\sigma[\neg \forall \mathbf{m} x\colon \neg H]$$

In agent specifications agent identifiers stand for (sets of) stream processing functions. Every time an agent identifier occurs in a specification another instantiation may be taken, i.e., another stream processing function out of the set of stream processing functions can be chosen.

Given a formula $H$ and an environment $\sigma$ we say $H$ is *valid* for $\sigma$ and write

$$\sigma \models H \qquad \text{if} \qquad M_\sigma[H] = \text{true}.$$

In the following we use $\bar{x}$ (and $\bar{y}$, respectively) as an abbreviation for tuples of declarations, i.e., for phrases of the syntactic unit ⟨dec-tuple⟩. Let $\bar{x}$ stand for

$$s_1 x_1, \ldots, s_n x_n,$$

where the $s_i$ specify the sort of the identifiers $x_i$. Furthermore let $x$ (and $y$ respective) stand for $(x_1, \ldots, x_n)$. Now we define under which circumstances an agent specification is fulfilled by an environment. Given an agent specification

$$\textbf{agent}\, f = \textbf{input}\, \bar{x}, \textbf{output}\, \bar{y}, \qquad H \quad \textbf{end}$$

where we assume that in $H$ only agent identifiers and the identifiers from $\bar{x}$ and $\bar{y}$ occur freely, we say for a given agent environment $\sigma$ "$\sigma$ fulfills the specification for $f$" if

$$\sigma \models \forall \bar{x}, \bar{y}\colon (y \leftarrow f(x) \Leftrightarrow H).$$

Given a family DEF of definitions of the agent, $f_1, \ldots, f_k$,

$$\textbf{agent}\, f_1 = \cdots H_1\, \textbf{end} \qquad \cdots \qquad \textbf{agent}\, f_k = \cdots H_n\, \textbf{end}$$

an agent environment

$$\sigma\colon \{f_1, \ldots, f_k\} \to P(\text{AGENT})$$

is called *consistent* w.r.t. DEF and we write $\sigma \models \text{DEF}$ if $\sigma$ fulfills all the agent specifications for $f_1, \ldots, f_k$.

In the context of equational ("algebraic") specifications of families of agents similar questions arise for the algebraic specification of abstract (data) types in hierarchies. One may introduce notions like persistency, hierarchy completeness (sufficient completeness), or hierarchy consistency in (algebraic) specifications of agents.

2.4. *Temporal Logics*

A highly developed notation and calculus for reasoning on (sets of) sequences is (linear time) temporal logic. Temporal logic mainly has been advocated and used for reasoning about complete concurrent systems cooperating via shared memory. Since agents are functions on sequences, temporal logic should provide a framework also for agent specifications. Since one has to reason in agent specifications about several sequences in one formula, the temporal logic framework has to be slightly generalized. Three temporal operators are introduced.

Let $s_i$ be stream identifiers and $H$ be a predicate where the $s_i$ are used as streams. Then we define

$$\bigcirc\, s_1,\ldots, s_k\colon H \underset{\text{def}}{=} H[(\mathbf{rest}\, s_1)/s_1,\ldots, (\mathbf{rest}\, s_k)/s_k]$$

$$\Box\, s_1,\ldots, s_k\colon H \underset{\text{def}}{=} \forall i \in \mathbb{N}\colon H[(\mathbf{rest}^i\, s_1)/s_1,\ldots, (\mathbf{rest}^i\, s_k)/s_k]$$

$$\Diamond\, s_1,\ldots, s_k\colon H = \underset{\text{def}}{=} \exists i \in \mathbb{N}\colon H[(\mathbf{rest}^i\, s_1)/s_1,\ldots, (\mathbf{rest}^i\, s_k)/s_k].$$

Here $\mathbf{rest}^i$ is assumed to be defined inductively by the equations

$$\mathbf{rest}^0 s = s, \qquad \mathbf{rest}^{i+1} s = \mathbf{rest}(\mathbf{rest}^i\, s).$$

By $H[E/s]$ we denote the expression that is obtained by replacing all occurrences of $s$ in $H$ by the expression $E$.

In terms of our semantic definitions, the meaning of the temporal operators is given by

$$M_\sigma[\bigcirc\, s_1,\ldots, s_k\colon H] \underset{\text{def}}{=} M_{\sigma 1}[H] \quad \text{where } \sigma 1 = \sigma[\mathrm{rest}(s_1)/s_1,\ldots, \mathrm{rest}(s_k)/s_k],$$

$$M_\sigma[\Box\, s_1,\ldots, s_k\colon H] \underset{\text{def}}{=} \forall i \in \mathbb{N}\colon M_{\sigma 1}[H],$$

where $\sigma 1 = \sigma[\mathrm{rest}^i(s_1)/s_1,\ldots, \mathrm{rest}^i(s_k)/s_k]$,

$$M_\sigma[\Diamond\, s_1,\ldots, s_k\colon H] \underset{\text{def}}{=} \exists i \in \mathbb{N}\colon M_{\sigma 1}[H],$$

where $\sigma 1 = \sigma[\mathrm{rest}^i(s_1)/s_1,\ldots, \mathrm{rest}^i(s_k)/s_k]$.

With these temporal operators one can give specifications without any use of "recursion" (implicit equations). For instance, the example onestream then reads

**agent** onestream = **output stream nat** $r$, $\quad \Box r$: **first** $r = 1$ **end**

Temporal formulas stand for infinite formulas and thus can be seen to allow often

more explicit specifications. For some of the examples of Section 2 one actually can obtain more explicit specifications by temporal logics:

**agent** infinite = **output stream bool** $r$,
$(\Box r$: **first** $r$ = true) $\vee$ ($\Box r$: **first** $r$ = false) **end**

**agent** addstream = **input stream nat** $a$, **stream nat** $b$, **output stream nat** $r$,
$\Box a, b, r$: **first** $r$ = **first** $a$ + **first** $b$ **end**

**agent** any = **output stream** $r$, $\Box r$: (**first** $r$ = true $\vee$ **first** $r$ = false) **end**

For other examples such as the agent merge or the agent schedule, more explicit (nonalgebraic) specifications using temporal logic are less obvious. An appropriate, more powerful temporal logic for specifying communicating agents without algebraic (recursive) equations seems one of the interesting questions to be looked at.

Possible candidates for such an extended temporal logic are, for instance, versions of the until operator, the (iterated) combine operator, and the fixed point operator for logical formulas as suggested in [1]. However, it does not seem clear so far, whether an algebraic style of specifications or a more logic-oriented style of specifications is more appropriate. For the moment *I* prefer a formalism which supports both styles.

## 3. Agent Specifications Describing Algorithms

The formalism introduced so far was explained as a specification tool. Nevertheless, agent specifications of particular syntactic forms can be seen as programs. Such specifications are called *algorithmic agents.*

### 3.1. *Algorithmic Agents*

An algorithmic agent is an agent specification of the following syntactic form:

$$\textbf{agent } f = \textbf{input } \bar{x}, \textbf{output } \bar{y}, \exists \bar{z}\colon H_1 \wedge \cdots \wedge H_j \textbf{ end}$$

The set $X$ of identifiers of $\bar{x}$ are called *input ports*, the set $Y$ of identifiers in $\bar{y}$ are called the *output ports*, the set $Z$ of identifiers in $\bar{z}$ are called the *internal ports.*

The formulas $H_i$, $1 \leqslant i \leqslant j$ are assumed to be of the form

$$(a_i \leftarrow T_1^{(i)} \vee \cdots \vee a_i \leftarrow T_p^{(i)}),$$

where $a_i$ is called the left-hand side of the clause $H_i$ and the $a_i$ are tuples of output ports and internal ports, i.e., $a_i \in (Y \cup Z)^p$ with some $p \in \mathbb{N}$. The $T_k^{(i)}$ are assumed to be expressions. Every identifier in $Y \cup Z$ must occur in exactly one left-hand side $a_i$. A system of agents is called *algorithmic*, if all agents of the system are *algorithmic.*

Algorithmic agent specifications can be trivially translated into applicative multiprograms written in programming languages such as AMPL (cf. [2]) for which

an operational (data driven reduction) semantics is available. This way algorithmic specifications define algorithms. In a system of agent specifications it is said that "the agent $f$ uses the agent $g$" if

(1) $g$ occurs in the body of the specification of agent $f$, or

(2) $g$ is used by an agent $h$ that is used by $f$.

If an agent $f$ uses itself, then it is called *recursive agent specification.* If an algorithmic agent specification contains equations with stream identifiers occurring on both the left-hand side and the right-hand side of equations between streams then we speak of *recursive (algebraic) equations for streams.*

Every algorithmic agent defines a network (a data flow graph), if we take $X \cup Y \cup Z$ as arcs and introduce a node for every clause $H_i$. The arcs have as sources the nodes of the clauses where they appear on the left-hand side. If the agent specification is recursive, then the defined network is infinite; if the agent specification is not recursive but contains recursive stream equations, then the network is finite and cyclic. If the algorithmic specification neither is recursive nor contains recursive stream equations, then the network is a finite, acyclic (directed) graph. Examples are given below.

## 4. Design Issues

After having introduced a particular language for the specification of agents, in this section a number of design issues are treated that give some insights on how the language for specifying agents can be used.

### 4.1. *A Combinatorial/Functional Notation for Agents*

So far the specifications are written using (internal) identifiers for data objects. In this section a functional style notation is introduced that allows combining given agent specifications by sequential composition, parallel composition, and feedback.

In the following we assume the two agent specifications to be given:

$$\textbf{agent } a1 = \textbf{input } \overline{x1}, \textbf{ output } \overline{y1}, H1 \quad \textbf{end}$$

$$\textbf{agent } a2 = \textbf{input } \overline{x2}, \textbf{ output } \overline{y2}, H2 \quad \textbf{end}$$

where $a1$ has $n1$ input lines and $m1$ output lines and $a2$ has $n2$ input lines and $m2$ output lines. We assume that $\overline{x1}, \overline{x2}, \overline{y1y1}, \overline{y2}$ are pairwise disjoint w.r.t. the (sets of) identifiers they contain.

The *parallel composition* of the two agents $a1$ and $a2$ is written by

$$a1 \parallel a2,$$

where $a3 = a1 \parallel a2$ has $n1 + n2$ input lines and $m1 + m2$ output lines and it is specified by

$$\textbf{agent } a3 = \textbf{input } \overline{x1}, \overline{x2}, \textbf{ output } \overline{y1}, \overline{y2}, H1 \wedge H2 \quad \textbf{end}$$

For simplicity let us assume that now $m1 = n2$. The *sequential composition* of the two agents $a1$ and $a2$ is written by

$$a1 \cdot a2,$$

where $a4 = a1 \cdot a2$ is an agent specification with $n1$ input lines and $m2$ output lines. It is defined by

$$\textbf{agent } a4 = \textbf{input } \overline{x1}, \textbf{ output } \overline{y2}, \exists\, \overline{x2}, \overline{y1}: x2 = y1 \wedge H1 \wedge H2 \quad \textbf{end}$$

The *feedback* of an agent is defined by

$$C_j^i a,$$

where the agent $a$ is assumed to be given by the specification (with $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant m$)

$$\textbf{agent } a = \textbf{input } \mathbf{s}_1 x_1, \ldots, \mathbf{s}_n x_n, \textbf{ output } \mathbf{r}_1 y_1, \ldots, \mathbf{r}_m y_m, H \textbf{ end}$$

and the agent $a5 = C_j^i a$ has $n-1$ input lines and $m$ output lines. It is defined by the agent specification

$$\begin{aligned}\textbf{agent } a5 = \textbf{input } & \mathbf{s}_1 x_1, \ldots, \mathbf{s}_{i-1} x_{i-1}, \mathbf{s}_{i+1} x_{i+1}, \ldots, \mathbf{s}_n x_n, \\ \textbf{output } & \mathbf{r}_1 y_1, \ldots, \mathbf{r}_m y_m, \quad \exists\, \mathbf{s}_i x_i: x_i = y_j \wedge H \qquad \textbf{end}\end{aligned}$$

Note that the composition of algorithmic agents always leads to algorithmic agents again.

The agent store may be turned into an algorithmic agent (assuming algorithmic agents for switch and schedule) by

$$\textbf{agent } \text{store} = \textbf{input stream data } d, \textbf{ stream bool } b, \textbf{ output stream data } r,$$

$$\exists\, \textbf{stream data } s, z: (z, r) = \text{switch}(s, b),\ s = \text{schedule}(d, r, \text{true } \&\ b) \textbf{ end}$$

This agent store may be represented by the data flow diagram in Fig. 2. It can be also expressed by

$$C_1^2((I \| I \| a) \cdot (\text{schedule} \| I) \cdot \text{switch} \cdot (\text{forget} \| I)),$$

where $I$ is the identity function and the agents forget and $a$ are the trivial agents specified by

$$\begin{aligned}&\textbf{agent } \text{forget} = \textbf{input stream data } d, \text{ true } \textbf{end} \\ &\textbf{agent } a = \textbf{input stream bool } b, \textbf{ output stream bool } b1, b2, \\ &\qquad\qquad\qquad\qquad\qquad\qquad b1 = \text{true } \&\ b,\ b2 = b \qquad\qquad \textbf{end}\end{aligned}$$

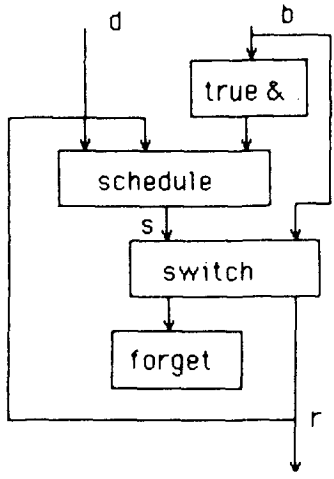


FIGURE 2

4.2. *Correctness of Agents*

During a design process of a distributed system, a sequence or even a tree of agent specifications are produced. The final agent specification should be a program, i.e., an algorithmic agent, that is correct w.r.t. the initial agent specification. In the most restrictive approach one could define correctness in the following way: An agent specification $A1$ is *totally correct* w.r.t. an agent specification $A0$, iff the set of functions denoted by $A1$ is identical to the set of functions denoted by $A0$. However, obviously this approach is too restrictive. More liberal notions of correctness have to be used and can be used much more flexibly in the design process.

4.2.1. *Partial Correctness*

Partial correctness of concurrent, communicating systems is a so-called *safety property*: a program is partially correct (w.r.t. some requirement specification) iff it never produces wrong results. This does not exclude that it diverges always immediately and therefore does not produce any results. Given an agent specification $A1$, that defines a set $F1$ of stream-processing functions, and a requirement specification $A0$, that defines a set $F0$ of stream processing functions, $A1$ is called *partially correct* w.r.t. $A0$, iff

$$\forall f1 \in F1\, \exists f0 \in F0 : f1 \sqsubseteq f0.$$

This formula defines a preordering on sets that is also used in power domains. It is abbreviated by $F1 \sqsubseteq_E F0$. Accordingly for a partially correct agent, every output stream is a prefix of an output stream included in the requirement specification.

4.2.2. *Robust Correctness*

Robust correctness is a *typical liveness* property: if a certain amount of output is guaranteed by the specification, then at least the same amount of information is to be guaranteed by a robustly correct program. Given an agent specification $A1$ defining a set $F1$ of stream processing functions and an agent specification $A0$ defining a set $F0$ of stream processing functions, then $A1$ is called *robustly correct* w.r.t. $A0$, iff

$$\forall f1 \in F1\, \exists f0 \in F0 : f0 \sqsubseteq f1.$$

This is again a preordering on sets that is also used in power domains. It is abbreviated by $F0 \sqsubseteq_M F1$. Accordingly for a robustly correct agent every output contains a prefix of some output included in the requirement specification. If according to the specification there may be some divergence and no more output, then a robust correct implementation may include some error message at that point.

4.2.3. *Correct Implementations*

During the design process of a program for a given specification $A0$ typically, particular design decisions are taken: specific algorithmic solutions are envisaged and certain nondeterministic alternatives are excluded. Accordingly for an agent specification $A1$ defining the set of functions $F1$ and the agent specification $A0$ defining the set of agents $F0$, the agent $A1$ is called a (*correct*) *implementation* of $A0$ if

$$F1 \subseteq F0.$$

This definition reflects a very essential point of view of nondeterministic concurrent programs: a correct implementation may not include all nondeterministic possibilities but restrict itself to certain subsets, i.e., by choosing a particular scheduling strategy. Immediately one obtains the following lemma.

LEMMA. *If $A1$ is a correct implementation of $A0$, then*

- $A1$ is partially correct w.r.t. $A0$
- $A1$ is robustly correct w.r.t. $A0$.

The converse statement does *not* hold: robust and partial correctness do not imply the correctness of an implementation.

Note that the inconsistent agent specification is not excluded. The inconsistent agent specification is a correct implementation for every agent specification; it is always robustly and partially correct. However, we will never be able to find an algorithmic agent implementing the inconsistent agent.

4.3. *Recursively Defined Algorithmic Agents*

For recursively defined algorithmic agent specifications the semantic definitions are very liberal: the semantics is not restricted to least (defined) fixed points but

considers the class of all fixed points. Trivially if one restricts the meaning of a recursively defined algorithmic agent by transition to a least (defined) fixed point semantics, then this represents a correct implementation of the given agent.

An algorithmic agent may trivially be translated into a program written in an applicative language for multiprocessing like AMPL (cf. [4]). Then, however, a particular fixed point theory is used, while in the specification all fixed points are included. This restriction can be also expressed in our specification language by prefixing algorithmic specifications by **rec**. The system of agents

$$\textbf{rec agent } f_1 = \cdots H_1 \quad \textbf{end} \qquad \cdots \qquad \textbf{rec agent } f_n = \cdots H_n \quad \textbf{end}$$

has the meaning $\sigma_{\text{rec}}$, where

$$\sigma_{\text{rec}} \in \{\sigma\colon (f_1,\ldots,f_n) \to P(\text{AGENT})\backslash\{\varnothing\}\}$$

is the $\subseteq$-least ($\subseteq$taken pointwise) function which is a $\sqsubseteq_M$-least and $\sqsubseteq_{EM}$-least fixed point of the equations $E_i$ with

$$\sigma_{\text{rec}} \models \forall \bar{x}_i \exists \bar{y}\colon y \leftarrow f_i(x) \wedge H_i$$

for $1 \leqslant i \leqslant n$. Here $\sqsubseteq_M$ is the preordering defined in the previous section. The preordering $\sqsubseteq_{EM}$ is defined by $\sqsubseteq_E \cap \sqsubseteq_M$.

The mathematical soundness and consistency of these definitions is implied, for instance, by Broy [8].

Trivially, if one prefixed a given family DEF of agent specifications by rec, then one obtains a correct implementation of DEF: the least fixed points in the sense above provide a subset of the set of fixed points. Therefore the final step of a program development, viewing an algorithmic specification as a program, is trivially correct.

### 4.4. *Transformations of Agent Specifications*

The mathematical semantics of agent specifications and the definition of correct implementations clearly define what may be called a (partially, robustly) correct transformation step for a specification: the transformed specification must be a (partially, robustly) correct implementation of the initial program. On this basis a calculus for transformation rules can be developed. It comprises essentially the rules of inference of predicate logic, the algebraic axioms of streams, and the basic rules of the used programming constructs (cf. [13]).

A calculus for transforming and manipulating formulas specifying agents and even relating them to the programming language concepts for describing nondeterminism like the nondeterministic choice operator can be found in [10].

Now a simple example for the transformation of an agent specification is given. It essentially shows a development starting with recursive agents towards recursive stream equations.

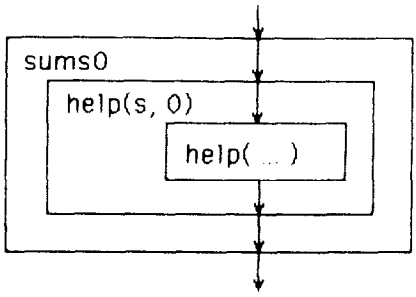


FIGURE 3

EXAMPLE. To demonstrate the transformation from recursion on functions to recursion on streams, we consider the agent sums0 that computes the partial sums of a stream:

**agent** sums0 = **input stream nat** $s$, **output stream nat** $r$, $r$ = help($s$, 0) **end**

**agent** help = **input stream nat** $s$, **nat** $n$, **output stream nat** $r$,
$r = n$ & help(**rest** $s$, $n$ + **first** $s$) **end**

The agent sums0 or, more precisely, the agent help is defined recursively. It can be graphically represented by Fig. 3. The agent sums0 is extensionally equivalent to the agent sums1:

**agent** sums1 = **input stream nat** $s$, **output stream nat** $r$, $r = 0$ & addstream($r$, $s$) **end**

The agent sums1 is defined by recursion on streams. It can be seen as defining a data flow diagram (see Fig. 4).

The agents sums0 and sums1 may also be represented by the term

$$C_1^1(\text{addstream} \cdot \text{zero}),$$

where zero is defined by

**agent** zero = **input stream nat** $s$, **output stream nat** $r$, $r = 0$ & $s$ **end**

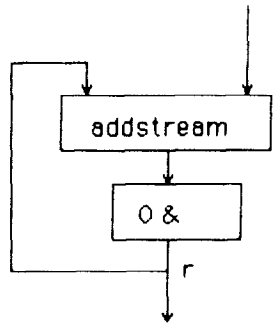


FIGURE 4

The example may be generalized to the following development rule: If an agent contains an equation with streams $r$, $s$ of the form (with some given agent $f$):

$$r = f(s) \qquad (*)$$

and one can prove the formulas (with some given agent $g$):

$$f(s) = g(f(s), s) \qquad (1)$$

$$\forall r1, r2: r1 = g(r1, s) \wedge r2 = g(r2, s) \Rightarrow r1 = r2 \qquad (2)$$

then (*) can be replaced by the equation

$$r = g(r, s).$$

Condition (1) states the requirement that $f(s)$ is a fixed point of $g$; the requirement (2) in addition implies that $g$ has a unique fixed point.

EXAMPLE (continued). For verifying the applicability condition (1) in our example above, one has to prove

$$\text{help}(s, n) = n \,\&\, \text{addstream}(\text{help}(s, n), s).$$

Condition (1) can be proved by structural induction on $s$:

(i) if $s = \langle \bot \rangle$, then (and similarly for $s = \mathscr{E}$):

$$\begin{aligned}
&\text{help}(s, n)\\
&\quad = n \,\&\, \text{help}(\text{rest } s, n + \text{first } s)\\
&\quad = n \,\&\, \bot \,\&\, \text{help}(\cdots)\\
&\quad = n \,\&\, \langle \bot \rangle\\
&\quad = n \,\&\, \bot \,\&\, \text{addstream}(\cdots)\\
&\quad = n \,\&\, \text{addstream}(\cdots);
\end{aligned}$$

(ii) if $s = x \,\&\, sn$ (where $x \in \mathbb{N}$) and (1) holds for $sn$:

$$\begin{aligned}
&\text{help}(s, n)\\
&\quad = n \,\&\, \text{help}(sn, n + x) && \text{(ind. hypoth.)}\\
&\quad = n \,\&\, n + x \,\&\, \text{addstream}(\text{help}(sn, n + x), sn) && \text{(def addstream)}\\
&\quad = n \,\&\, \text{addstream}(n \,\&\, \text{help}(sn, n + x), x \,\&\, sn) && \text{(def help, } s\text{)}\\
&\quad = n \,\&\, \text{addstream}(\text{help}(s, n), s).
\end{aligned}$$

For infinite $s$, the continuity of the involved functions proves the result. Finally, one has to prove that the equation

$$r = n \,\&\, \text{addstream}(r, s)$$

has a unique fixed point for any $s$. Again, this can be proved by induction on the length of $s$:

(i) if $s = \langle \bot \rangle$ then the equation implies

$$\text{addstream}(r, \langle \bot \rangle) = \langle \bot \rangle, \qquad \text{and hence} \qquad r = n \,\&\, \langle \bot \rangle;$$

(ii) if for $s$ the solution of the equation above is unique, then with $x \in \mathbb{N}$,

$$r = n \,\&\, \text{addstream}(r, x \,\&\, s)$$

one gets

$$r = n \,\&\, (\textbf{first}\ r + x) \,\&\, \text{addstream}(\textbf{rest}\ r, s)$$

which now can be replaced by the equations

$$r = n \,\&\, t, \qquad t = (n + x) \,\&\, \text{addstream}(t, s).$$

According to our induction hypothesis, $t$ is uniquely determined for finite streams $s$. For infinite streams the uniqueness of the fixed point of the equation follows from a continuity argument: the fixed point depends continuously on $s$.

Of course, the transformation rule above is only one simple example for transformations of agent specifications. It can be seen immediately that there is a rich class of transformation rules for agent specifications. Many of the transformation rules for sequential programs (cf. [9]) can be adapted also for agent specifications.

### 4.5. *Verification of Agent Specifications*

Proving for two given agents $A0$, $A1$ that $A1$ is a (partially, robustly) correct implementation of $A0$ represents the classical case of verification. After respectively renaming the input and output identifiers of $A1$, one basically has to show that the specifying predicate of $A1$ implies the specifying predicate of $A0$. Formally this again can be done by transforming $A0$ to $A1$ or, more precisely, deducing $A0$ from $A1$. Moreeover, one may also think of particular calculi for the verification of agents. For instance, calculi dealing just with particular aspects of communicating agents may be developed such as calculi for partial or robust correctness.

### 4.6. *Structured Programming with Agent Specifications*

Trivially, many different specifications can be given for one agent, i.e., one set of stream processing functions. Apart from different ways of writing equivalent predicates and renaming of local identifiers one could use different ways of structuring a specification.

A complete system of agent specifications consists of a family of agent specifications where for all occurring agent identifiers agent specifications are provided. The relation "agent $a_i$ is used in agent $a_j$" then defines a quasi-ordering,

and therefore induces a partial ordering on a quotient structure on the set of agents. This quasi-ordering is called the *macro structure of the system.*

Every agent itself contains some structure: the way the specifying predicate is written. Especially for algorithmic agents this form of the predicate, the number of local identifiers is interesting because it defines a finite network of communicating agents. This structure is called the *micro structure of* (*the components of*) *the system.*

Clearly the micro structure of an agent may again be seen as the macro structure of a (sub-)system with some further micro structure. Due to the use of recursive agents, such a transition from macro to micro views may be done arbitrarily often.

An important transformation rule that connects the micro structure with the macro structure is the unfold/fold rule for agents: Given an agent

$$\textbf{agent } a0 = \textbf{input } \overline{x0}, \textbf{ output } \overline{y0}, E0 \quad \textbf{end}$$

and an occurrence of the agent $a0$ in an equation

$$t1 = a0(t2) \tag{3}$$

then one may "unfold" the agent specifications, i.e. one may replace Eq. (3) by the formula

$$\exists\, \overline{x0}, \overline{y0}: t1 = y0 \wedge x0 = t2 \wedge E0. \tag{4}$$

Of course it is assumed that name clashes do not appear.

Even the reverse of the rule above is possible: the formula (4) may be replaced by Eq. (3), provided (4) is occurring outside of the body of the agent $a0$. If $a0$ may be graphically represented by a finite data flow network (Fig. 5) and an agent $a$ uses $a0$ (Fig. 6) then by the rule, one obtains for the agent $a$, the network shown in Fig. 7. This way algorithmic agents are transformed into algorithmic agents again.

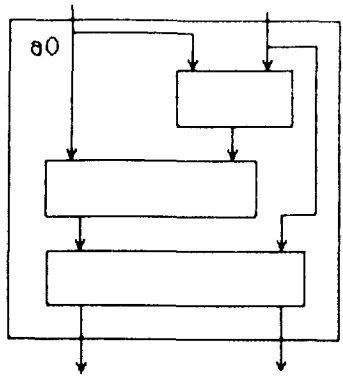


FIGURE 5

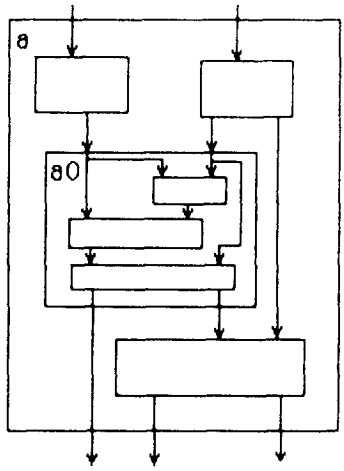


FIGURE 6

## 5. TOWARDS A DESIGN METHODOLOGY FOR DISTRIBUTED SYSTEMS

The design of distributed systems can follow the same patterns as the design of sequential systems. Roughly, we may use the following steps:

— specification of the basic data structures and their characteristic functions (say by algebraic specifications),

— specification of additional functions,

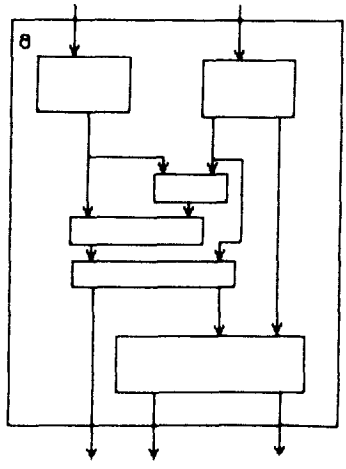


FIGURE 7

— decomposition of the functions into more simple functions,

— decomposition of the data objects into more basic ones,

— derivation of algorithmic representations for the functions and data objects,

— derivation of efficient versions.

This scheme can be used for sequential as well as distributed (nonsequential) systems. Here we concentrate on two aspects in this development which are most interesting for obtaining distributed systems.

One aspect concerns the derivation of (specifications of) stream-processing functions, the other one the decomposition of stream-processing functions into networks.

### 5.1. *From Functions to Stream-Processing Functions*

Some problems lead to specifications of stream-processing functions in a straightforward way, since such problems, for instance, arise in connection with interactive program components, or the problem already arises in connection with a distributed system. Then the first specification of the programming task is already a stream-processing function. Note that we consider a (finite or infinite) network of stream-processing functions as a distributed system. Other problems do not contain distribution aspects and for them stream-processing functions and networks can be developed in a systematic way from simple functions.

We now give some simple patterns for obtaining stream-processing functions from simple functions.

#### 5.1.1. *Interactive Iteration*

Given a monotonic function

$$f: (D_1 \times \cdots \times D_m) \to (D_{m+1} \times \cdots \times D_{m+n}),$$

where the $D_i$ are flat domains, then $f$ can immediately be associated with a stream-processing function

$$pf^*: (\mathrm{STREAM}(D_1) \times \cdots \times \mathrm{STREAM}(D_m)) \to (\mathrm{STREAM}(D_{m+1}) \times \cdots \times \mathrm{STREAM}(D_{m+n}))$$

defined by the least fixed point of the equations:

$$f^*(s_1,..., s_m) = (\mathscr{E},..., \mathscr{E}) \quad \text{if} \quad s_i = \mathscr{E} \quad \text{for some} \quad i,\ 1 \leqslant i \leqslant m,$$

$$f^*(a_1 \,\&\, s_1,..., a_m \,\&\, s_m \,\&\, s_m) = (b_1 \,\&\, r_1,..., b_n \,\&\, r_n)$$

if

$$f(a_1,..., a_m) = (b_1,..., b_n) \wedge f^*(s_1,..., s_m) = (r_1,..., r_n)$$

This way stream-processing functions without "internal states" are obtained. Stream-processing functions with internal states can be obtained in the following way.

5.1.2. *Internal States by Feedback Loops*

Now given a monotonic function

$$f: (D_1 \times \cdots \times D_m) \to (D_{m+1} \times \cdots \times D_{m+n}),$$

where $D_1 = D_{m+1}$ we may immediately associate a stream processing function with $f$ for each given $x \in D_1$:

$$f_x: (\mathrm{STREAM}(D_2) \times \cdots \times \mathrm{STREAM}(D_m)) \to (\mathrm{STREAM}(D_{m+2}) \times \cdots \times \mathrm{STREAM}(D_{m+n}))$$

defined by

$$f_x(s_2, \ldots, s_m) = (r_2, \ldots, r_n),$$

where $(r_1, \ldots, r_n)$ is the least fixed point of

$$(r_1, \ldots, r_n) = f^*(x \,\&\, r_1, s_2, \ldots, s_m).$$

Note that $f_x$ can be understood as an agent with an internal state from $D_1$, where $x$ represents the initial state. The stream $x \,\&\, r$ in the definition above can be understood as the stream of "internal states" of the agent $f_x$ given the input $(s_2, \ldots, s_m)$.

Note that on these techniques the construction of interactively usable modules for given (say algebraically specified) data structures can be based.

5.2. *From Stream-processing Functions to Networks*

Given a stream processing function we may interested to decompose it into a network of simpler (predefined) agents. This way a distribution of the function into a distributed system can be obtained. The decomposition can be performed completely within a calculus by using the rules given above. If the specifying formula of an agent can be transformed (by the rules of predicate logic) into a form such that the agent can be considered as an algorithmic agent (that can be understood as a network) or decomposed into parallel agents, sequential agents, or feedback agents, then networks of agents are generated.

In all these techniques what we need are deductive theories for equational logic as well as logical rules dealing with the arrow "$\leftarrow$." In this way a design calculus for distributed systems may be provided.

## 6. Application to Other Models for Concurrency

Specifications of concurrent communicating agents are based on the model of stream processing functions. This is essentially a model with implicit buffering message passing. Other important models are handshaking message passing or

shared memory systems. An immediate question is, whether the specification methods for stream-processing agents can be applied to the other models, too.

The answer is rather simple: there exist semantic models both for shared memory as well as for handshake communication in terms of stream-processing functions (cf. [3–7]). Therefore one may simply write agent specifications that specify stream-processing functions that represent the semantics of programs written in those programming languages according to those semantic models.

EXAMPLE. Specification of producer and consumer working with shared memory.

According to the semantic model for shared memory as given in [6] components of a system of parallel programs working on some shared memory, the access to which is protected by conditional critical regions, may semantically be modelled by (sets of) functions mapping streams of states to streams of states and "rejections." For instance, the specification of the producer/consumer problem may read as follows (here $x$, $y$, $z$, $q$ denote programming variables):

**agent** producer = **input stream state** $s$, **output stream**(**state** $\cup$ {REJECTED}) $r$,

$$\begin{aligned}&\textbf{first } r = (\textbf{first } s)[x_0/x] \wedge\\ &\Box r, s: \bigcirc r, s: \textbf{first } r = (\textbf{first } s)[\text{next}((\textbf{first } s)(x))/x,\\ &\qquad\qquad \text{app}((\textbf{first } s)(q), (\textbf{first } s)(x))/q] \qquad \textbf{end}\end{aligned}$$

**agent** consumer = **input stream state** $s$, **output stream** (**state** $\cup$ {REJECTED}) $r$,

$$\begin{aligned}&\textbf{first } r = (\textbf{first } s)[y_0/y] \wedge\\ &\quad \Box r, s: \bigcirc r, s:\\ &\qquad \neg\, \text{isempty}((\textbf{first } s)(q)) \Rightarrow \textbf{first } r = (\textbf{first } s)[\text{top}(q)/z,\\ &\qquad\qquad\qquad\qquad\qquad\qquad \text{pop}(q)/q,\\ &\qquad\qquad\qquad\qquad\qquad\qquad \text{consume}(y, \text{top}(q))/y],\\ &\qquad \text{isempty}((\textbf{first } s)(q)) \Rightarrow \textbf{first } r = \text{REJECTED} \qquad\qquad \textbf{end}\end{aligned}$$

Here the notation $s[d/x]$ is used as the notation for updating the state $s$ by replacing the value for the identifier $x$ by $d$. Procedural programs that fulfill this specification then read, for instance,

*producer*:

$$\begin{aligned}&x := x_0;\\ &\textbf{while } \text{true } \textbf{do await } \text{true } \textbf{then } q := \text{app}(q, x) \textbf{ endwait};\\ &\qquad\qquad x := \text{next}(x) \qquad\qquad \textbf{od}\end{aligned}$$

*consumer*:

$$\begin{aligned}&y := y_0;\\ &\textbf{while } \text{true } \textbf{do await } \neg\text{isempty}(q) \textbf{ then } q, z := \text{pop}(q), \text{top}(q) \textbf{ endwait};\\ &\qquad\qquad y := \text{consume}(y, z) \qquad\qquad \textbf{od}\end{aligned}$$

Here the specification does not look very convincing: It is textually larger and (since less familiar) seems harder to understand than the programs. However, a tuned notation (following [14]) for the specification might change this view. Writing $x'$ for the input value of a program variable (i.e., for (**first** $s$)($x$)) and $x^{\backprime}$ for its output value (i.e., for (**first** $r$)($x$)) and dropping $r$, $s$ from temporal formulas one obtains: sm-agents (shared memory agents). Moreover, we just write REJECTED instead of **first** $r$ = REJECTED. It is just reasonable to add the information which variables are assumed to be shared:

**sm-agent** producer ≡ **shared** $q$,

$$x' = x_0 \wedge \Box \bigcirc (x' = \text{next}(x^{\backprime}) \wedge q' = \text{app}(q^{\backprime}, x')) \qquad \textbf{end}$$

**sm-agent** consumer ≡ **shared** $q$,

$$y' = y_0 \wedge \Box \bigcirc ((\neg \text{isempty}(q^{\backprime}) \Rightarrow (z' = \text{top}(q^{\backprime}) \wedge q' = \text{pop}(q^{\backprime}) \wedge y' = \text{consume}(y^{\backprime}, z')) \wedge (\text{isempty}(q^{\backprime}) \Rightarrow \text{REJECTED})) \qquad \textbf{end}$$

In a similar way, one may develop tuned notations for CSP-like programs or CCS-like programs by using the stream-based semantic models for CSP/CCS as given in [3, 6]. We give a simple example for a specification of a CSP-like program. We use a version of CSP, where the components of a distributed system communicate by channels that are identified by names. Every channel is input channel for exactly one process and can be used as output channel by all other processes. A CSP-program with the behaviour of a queue reads as follows

**begin var queue** $q$ := emptyqueue;
**do** ¬isempty($q$): $c$! first($q$) **then** $q$ := rest($q$)
□ $d$? **then** $q$ := app($q$, $d$) **od**
**end**

The semantic model for CSP is given by stream-processing functions of the form

$$\text{STREAM(OFFER)} \rightarrow \text{STREAM(REACTION)},$$

where

$$\text{OFFER} = (\{\text{in}\} \times \{\text{CHAN-ID}\} \times \{\text{DATA}\}) \cup (\{\text{out}\} \times \{\text{CHAN-ID}\})$$
$$\text{REACTION} = (\text{DATA} \cup \{\text{REJECTED, ACCEPTED}\}).$$

Accordingly, the specification of the program above reads

**agent** queue = **input stream offer** of, **output stream reaction** $re$,
$re$ = hqueue (of, emptyqueue) **end**

**agent** hqueue = **input stream offer** of, **queue** $q$, **output stream react** $re$,
$\forall$ **chan-id** $i$, **data** $x$: first(of) = $(in, i, x) \Rightarrow$
$((i = d \wedge re = \text{ACCEPTED} \;\&\; \text{hqueue}(rest(of), \text{app}(q, x)))$
$\vee\ (i \neq d \vee re = \text{REJECTED} \;\&\; \text{hqueue}(rest(of), q)) \wedge$
first(of) = (out, $i$) $\Rightarrow$
$((i = c \wedge \text{isempty}(q) = \text{false} \wedge re = \text{first}(q) \;\&\; \text{hqueue}(\text{rest}(of), rest(q)))$
$\vee\ ((i \neq c \vee \text{isempty}(q) = \text{true}) \wedge re = \text{REJECTED} \;\&\; \text{hqueue}(rest(of), q))$
**end**

Again one may think of a better tuned notation for getting shorter specifications here, too.

## 7. Concluding Remarks

The approach of agent specifications reflects an attempt to bring together distinct research directions: algebraic (equational) and predicative specifications, denotational models for concurrent, communicating systems, temporal logic, functional programming, and program development by transformations. The presented approach to the specification of concurrent, communicating systems includes a number of design decisions that will now be shortly recapitulated and some justification should be given, too.

The chosen *semantic model* is sets of continuous functions mapping tuples of streams to tuples of streams. Why tuples of streams are considered is obvious. Why we consider sets of functions instead of relations or set-valued functions might be less clear. It is done to avoid some subtle problems in connection with definitions of streams by fixed points over nondeterministic functions (for a more detailed justification see, for instance, [2]). Whether the restriction to continuous functions is actually always appropriate, however, seems less clear.

The chosen *logical framework* is basically algebraic: All properties are specified by equations or by the operator "$\leftarrow$." The inclusion of temporal logic can just be seen as a notational variant being appropriate since sequence-like structures like streams are used. Maybe it is important to point out that the logic is two-valued in spite of the existence of partial functions (in the disguise of total functions with $\perp$ as result) and of nondeterministic agents.

The chosen concept of the *validity* of specifications represents a very subtle point. The validity of formulas containing nondeterministic terms can be defined in several ways (by several modalities). Which of these possibilities is actually most appropriate can only be qualified after gaining some further experience.

It may be that the most important properties of a specification method are not only the underlying theoretical concepts but the more pragmatic issues such as readability, tractability, support for structuring, possibilities of visual aids, and machine support. In this light the presented approach seems rather attractive. The essential principle that every family of agents can be also considered (and formally

specified) as a network and vice versa, and the integration of pure specification constructs and algorithmic views into one framework make it reasonable to expect that the given approach can be further developed into a flexible and practically helpful tool.

What has been presented in the previous sections can be seen as an example of a first attempt to develop a modular specification and design method for communicating concurrent systems rather than a fully worked out methodology. Much remains to be done until such an approach actually will work practically.

## ACKNOWLEDGMENTS

This is the revised and (w.r.t. the semantics) simplified version of a paper that appeared at the TAPSOFT conference. It is a pleasure to acknowledge a number of helpful discussions with the members of IFIP Working Group 2.3, and with Thomas Streicher.

## REFERENCES

1. H. BARRINGER, R. KUIPER, AND A. PNUELI, A compositional temporal approach to a CSP-like language, unpublished, 1985.
2. M. BROY, A fixed point approach to applicative multiprogramming, *in* "Theoretical Foundations of Programming Methodology" (M. Broy, G. Schmidt, Eds.), pp. 565–623, Reidel, Dordrecht, 1982.
3. M. BROY, Denotational semantics of communicating processes based on a language for applicative multiprogramming, *Inform. Process. Lett.* **17**, No. 1 (1983), 29–38.
4. M. BROY, Fixed point theory for communication and concurrency, *in* "IFIP TC2 Working Conference on Formal Description of Programming Concepts II, Garmisch, June 1982" (D. Björner, Ed.), pp. 125–147, North-Holland, Amsterdam/New York/Oxford, pp. 125–147, 1983.
5. M. BROY, Applicative real time programming, *in* "Information Processing 83" (R. E. A. Mason, Ed.), pp. 259–264.
6. M. BROY, Semantics of communicating processes, *Inform. and Control* **61**, No. 3 (1984), 202–246.
7. M. BROY, Denotational semantics of concurrent programs with shared memory, *in* "STACS 84" (M. Fontet and K. Mehlhorn, Eds.), Lecture Notes in Computer Science Vol. 166, pp. 163–173, Springer, Berlin/Heidelberg/New York, 1984.
8. M. BROY, On the Herbrand Kleene universe of nondeterministic computations, *in* "Mathematical Foundations of Computer Science 1984" (M. P. Chytil and V. Koubeck, Eds.), Lecture Notes in Computer Science Vol. 176, pp. 214–222, Springer, Berlin/Heidelberg/New York/Tokyo, 1984.
9. M. BROY, Algebraic methods for program construction: The project CIP, *in* "Program Transformation and Programming Environments" (P. Pepper, Ed), NATO ASI Series. Series F: 8, pp. 199–222, Berlin/Heidelberg/New York/Tokyo, 1984.
10. M. BROY, Predicative specifications for functional programs describing communicating networks, *Inform. Process. Lett.* **25**, No. 2 (1987).
11. M. BROY AND M. WIRSING, Partial abstract types, *Acta Inform.* **18** (1982), 47–64.
12. J. B. DENNIS, First version of a data flow procedure language, *in* "Colloque sur la Programmation" (B. Robinet, Ed.), Lecture Notes in Computer Science Vol. 19, pp. 362–367, Springer, Berlin/Heidelberg/New York, 1974.
13. P. DYBIER, Reasoning about streams in intuitionistic logic, unpublished, 1985.
14. E. C. R. HEHNER, Predicative programming. Parts I + II, *Comm. ACM* **27**, No. 2 (1984), 134-151.
15. C. A. R. HOARE, S. D. BROOKES, AND A. W. ROSCOE, "A Theory of Communicating Sequential Processes," Technical Monograph PRG-21, Oxford University Computing Laboratory, Programming Research Group, Oxford, 1981.

16. G. KAHN AND D. MACQUEEN, Coroutines and networks of parallel processes, *in* "Proceedings, of the IFIP Congress 77, pp. 994–998, North-Holland, Amsterdam, 1977.
17. R. M. KELLER, Denotational models for parallel programs with indeterminate operators, in "Formal Description of Programming Concepts" (E. J. Neuhold, Eds.), pp. 337–366, North-Holland, Amsterdam, 1978.
18. R. MILNER, "A Calculus of Communicating Systems," Lecture Notes in Computer Science Vol. 92, Springer, Berlin/Heidelberg/New York, 1980.
19. D. PARK, On the semantics of fair parallelism, *in* "Abstract Software Specification," (D. Björner, Ed.), Lecture Notes in Computer Science Vol. 86, pp. 504–526, Springer, Berlin/Heidelberg/New York, 1980.
20. G. PLOTKIN, A powerdomain construction, *SIAM J. Comput.* **5** (1976), 452–486.
21. A. PNUELI, The temporal logic of programs, *in* "Proceedings, 13th FOCS, Providence, 1977," pp. 46–57.
22. V. R. PRATT, Process logic, *in* "Proceedings, 6th Annual Sympos. Principles of Programming Lang. 1979," pp. 83–100.
23. M. SMYTH, Power domains, *J. Comput. System Sci.* **16** (1978), 23–36.
24. G. WINSKEL, Event structure semantics of CCS and related languages, *in* "ICALP 82" (M. Nielsen and E. M. Schmidt, Eds.), Lecture Notes in Computer Science, Vol. 140, pp. 561–576, Springer, Berlin/Heidelberg/New York, 1982.