Executable specifications for discrete event systems

Executable specifications for discrete event systems

by

K.M. van Hee, G.J.Houben,
L.J.Somers, M.Voorhoeve

88/17

October, 1988

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author or the editor.

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands

Editors:

EXECUTABLE SPECIFICATIONS FOR DISCRETE EVENT SYSTEMS [1]

by

K.M. van Hee, G.J. Houben, L.J. Somers and M. Voorhoeve

ABSTRACT

A formal framework for the specification of discrete event systems is introduced. The precise description of such a system, either for analysis or design, is a major problem in systems engineering. The underlying theoretical model is based on automata theory, and subsumes the Petri net approach. For modeling real systems one has to consider a network of interacting components and therefore three aspects of a system have to be specified: the structure of the state spaces of the components, the state transformations of the components and the interaction structure. Within the framework a tool has been developed with a language for an integrated specification of those three aspects of a system. After the introduction of the language an example illustrates our approach.

[1] This research is partly supported by IBM Nederland N.V.

1. INTRODUCTION

One of the major problems of systems engineering is the precise description of the system under consideration. The need for such a description manifests itself in two different situations: either one wants to analyse an existing system or one wants to design a new system. In the first case one has to map a part of the physical world into a description, in the second case one has to map concepts in our head into a description. If we compare these aspects of systems engineering to architectural design then we observe that the description language of the architect has an alphabet consisting of line segments, the descriptions are drawings and their semantics are abstract three dimensional objects. We all know that the abstract three dimensional objects are idealizations of the objects in the physical world they represent: there are no flat walls and rectangular corners. However, the abstract objects are useful guidelines for the building contractor to realize the physical object. The contractor may deviate from the abstract object within some tolerance bounds: the implementor's freedom.
For the analysis and design of systems we are looking for similar tools: a description language with semantics such that the described objects are realizable within some bounds.

The systems that we consider are the ones that are called discrete event systems (des). The characteristic of such systems is that they have a finite or countable state space and their behaviour can be described by sequences of successive states. Hence, systems that can be described by (partial) differential equations are out of scope.
The class of systems we focus on contains distributed information systems such as airline reservation systems and production control systems. However, other systems such as logistic systems fit also into the framework of des's. In mathematics and computing science may formalisms to describe des's are developed, however, most of them are only suitable to describe some aspects of a des, but the systems engineer needs a formalism that covers all relevant aspects in an integrated way. A simple formalism is given by automata theory. There a system is considered in combination with an (unknown) environment that sends actions to the system and waits for reactions. An automaton is usually characterized by a state space, input/output alphabet and a transition function. However, often the model of one automaton exchanging information with its environment is not adequate, for instance to describe a distributed database of an airline-reservation system. Then one needs to consider a network of interacting components and then one has to specify an interaction structure in addition.
In fact there are three major aspects of a des:

- the structure of the state spaces of components,
- the state transformations of components,
- the interaction structure.

We will review some formalisms for specifying these aspects; we do not claim to give a complete survey.
A state space is in fact a set and there are many ways to specify them: any language to define data types provides one. For complex state spaces one often uses a data model such as the entity-relationship model [Chen 76] or the relation model [Ullman 82]. A

database scheme specifies in fact a data type.
For the specification of state transformations we often use logic to define pre- and postconditions, but functional languages are also used. The latter approach gives a constructive specification: functions are specified by lambda expressions involving more elementary functions, and in the end by primitive functions. Typed functional languages have also facilities for specification of data types, so they cover the first two aspects of a des. Two approaches for specification of state structure and transformations, that cover both the functional way and the way of pre- and postconditions, are the languages Z [Hayes 87] and VDM [Jones 86]. They are very suitable for specifying the first two aspects of a des, however, at the moment they have no way to specify the third one. Algebraic specifications [Goguen et al 77], [Guttag, Horning 78] also allow only to specify the first two aspects.
There are many formalisms to specify an interaction structure in a distributed system. By interaction we mean the way components transfer data and the way they activate (or trigger) each other. The first way of interaction is also called data flow and the second way may be called control flow. Often it is not possible to distinguish both types of interaction. There are two well-known algebraic process theories, CSP [Hoare 85] and CCS [Milner 80]. Closely related are trace theory [Rem 83], [Mazurkiewicz 84] and process algebra [Bergstra, Klop 84]. In these approaches parallelism is modeled by interleaving of actions and they are suitable to specify communication between components of a system. They are poor in their capabilities of specifying data structures and data transformations. Another way to describe interaction structure is the use of Petri nets [Petri 76] where conditions and events are modeled by bipartite graphs. A generalization is found in predicate/transition nets [Genrich, Lautenbach 81].
An approach based on finite state machines with a graphical language and formal semantics is called statecharts [Harel 86]. Petri nets and statecharts are not very well suited to describe data structures and transformations.
There are several informal frameworks to describe data flow, using graphical languages. In practice ISAC [Lundeberg 79], DFD [Ward, Mellor 85] and SADT [Ross 77] are used frequently. Most of these frameworks have also methods for specifying data structures, using the entity-relationship model or Bachman diagrams [Bachman 69]. However, transformations are specified by sugared imperative languages.

We have developed a formal framework based on a mathematical model of discrete event systems and a language to specify the components of a des. The model is related to Petri nets. We use here Petri net terminology to explain our model. We have attached a value to each token. Each place has a type and values of tokens in a place belong to that type. Transitions are endowed with a transformation that transforms input tokens into output tokens. We do not require that all output places get a token nor that an output place gets only one token. A transition may fire only if from all its input places a token can be consumed. In our terminology a token is called a trigger, a place a channel and a transition a processor. Besides a des we also consider a real-time des where each token/trigger has besides the value a time stamp. The meaning of it is that it cannot be consumed before that time. The real-time des is modeled as a des with some additional properties. It is more powerful than the DEVS-model [Concepcion, Zeigler 88]: the basic components of DEVS can be made easily from ours. In Section 2 we introduce our model.
We introduce a language for the specification of a des. This language has been devel-

oped with the following aims in mind:

- it must be executable, and can thus be used for prototyping,
- it must allow the comparison of alternative specifications,
- it must encourage the reuse of specified components,
- it must have an open interface to allow the incorporation of external software.

Our language consists of two parts: a functional part and an dynamic part. The functional part is used to define types and functions, in other words to define a many sorted algebra. The type system consists of some primitive types and a few type constructors to define new types. In this way one specifies a type for each channel. A sugared version of lambda calculus is used to define new functions from a set of primitive functions. In this way one specifies for each processor its transformation. The dynamic part of the language is used to specify the network of processors and channels and therefore the interaction structure.
The state of a des is the configuration of triggers in the channels. The structure of the state space of a channel is characterized by its type: the set of all bags over this type. In this way one is able to specify all three aspects of a des:

- the structure of state spaces of components is specified using the type-system,
- the state transformations are specified by functions using lambda calculus,
- the interaction structure is specified by the network structure.

The type system is specified by the network structure. The type system we use is simple but more powerful than the relational data model because it allows nested structures. To model a database one may define a channel that is connected to a processor both as input channel and output channel, and that contains always exactly one trigger that represents the database state. This way of modeling a database corresponds to SADT and ISAC. The language is treated in Section 3.
The fact that our networks may be considered as Petri nets allows us to use Petri net theory to verify structural properties [Genrich 81]. Since we use a typed-functional language to specify types and functions we specify by high level construction. This has two advantages:

- we have an existence proof for each specified system (it is, for instance, not possible to specify a processor that computes the largest prime number),
- we are able to generate a simulation model or prototype of the specified system.

Having a prototype of a specified system is very important because potential end-users are seldom able to check formal specifications, however, they may test the functionality of the specified system by playing with the prototype. A simulation model of a specified system can also be used to test performance. This can be done using the real-time variant of the des-model: the throughput of triggers can be measured.

We have developed a software tool to support the specification process. It consists of a type checker that verifies the rules of the type system and an interpreter that generates a simulation of the specified system. We have specified many different systems such as: a token ring, a distributed data base, an access control system, a distributed inventory control system and a banking system. We have also done examples from [Shridar, Hoare 85] and [Hayes 87] and these comparisons give us the confidence that our framework is powerful and not too complicated to apply. In Section 4 we treat an example.

2. FRAMEWORK

A very elementary notion of a system is a graph $\langle S,T\rangle$, called a ***basic system***. Here S is a finite or countable set, called the *state space*, and T is a binary relation over S, called the *transition relation*. Elements of $\{s \in S \mid \neg\, \exists t \in S \;:\; \langle s,t\rangle \in T\}$ are called *terminal states*. Any finite sequence of states ending with a terminal state or any finite sequence of states, such that each pair of successive states belongs to T is called a *path* and any finite, serried subsequence of a path is called a *trace*. The set of all paths of a basic system is called the *process* of the system.
Although it is, in principle, possible to map complex systems into the framework of a basic system it is practically infeasible. Therefore we introduce a more structured framework, called *discrete event system* (DES). It is much easier to map a complex system into this framework. A discrete event system (des) has also a state space and a transition relation and is therefore a basic system. Hence, concepts such as path and process have a meaning for discrete event systems too. A des consists of two kinds of components: *processors* and *channels*. (They correspond to transitions and places in Petri nets.) A processor is connected to one or more input channels and one or more output channels. With each channel a type is associated and with each processor a function. The signature of the function of a processor corresponds with the types of in- and output channels. A channel may be shared by several processors as input or output channel. At each moment the channels may contain *triggers* (tokens in Petri nets). A trigger has a value that belongs to the type of the channel. There may be more than one trigger with the same value. So a channel contains a bag over its type. At each moment there may be a *transition*, which means that the configuration of triggers, called the *state*, in the channels may change. Such a transition occurs instantaneously and is *executed* by the processors. A processor may execute if it is able to select a trigger in each of its input channels. A trigger can be selected by only one processor. The execution of a processor means that the selected triggers are consumed (deleted) and that new triggers for the output channels of the processor are produced. To represent a des we use a diagram technique like for Petri nets.
Now we formalize the DES-framework. We use the following notations. If A is a set, then $\mathbb{P}(A)$ denotes the set of all finite subsets of A. If Y is a set of sets, then $\bigcup Y$ denotes the union of all elements of Y. For a set-valued function F, ΠF denotes the set of all (total) functions f over $\mathrm{dom}(F)$ and $\Pi^* F$ the set of all partial functions f over $\mathrm{dom}(F)$, both with $\forall x \in \mathrm{dom}(f) \;:\; f(x) \in F(x)$. for set A, $\mathbb{B}(A)$ denotes the set of all multisets (bags) over A, i.e. the set of all functions from A to $\mathbb{N}_0$, the set of nonnegative integers. $\mathbb{N}_1$ denotes the set of all positive integers. For $x \in \mathbb{B}(A)$ and $a \in A$:

$$a \in x \quad \text{iff} \quad x(a) > 0\ .$$

For $x \in \mathbb{B}(A)$ and $s \in \mathbb{P}(A)$:

$$\begin{array}{lcll} s \subset x & \text{iff} & \forall a \in s & : \; a \in x \\ x \setminus s & = & \lambda a \in A & : \; \text{if } a \in s \wedge a \in x \text{ then } x(a) - 1 \text{ else } x(a) \\ x \cup s & = & \lambda a \in A & : \; \text{if } a \in s \text{ then } x(a) + 1 \text{ else } x(a)\ . \end{array}$$

For $x, y \in \mathbb{B}(A)$:

$$x \cup y = \lambda a \in A \ : \ x(a) + y(a) \ .$$

For notational clarity we often write the function application $f(x)$ as f_x. We denote function restriction by $\restriction$ and the set of all partial functions from A to B by $A \nrightarrow B$.

Definition 1.
A *discrete event system* (des) is a four tuple $\langle R, C, I, O \rangle$ where R is a function-valued function and C, I and O are set-valued functions, such that

- $\mathrm{dom}(I) = \mathrm{dom}(O) = \mathrm{dom}(R)$, finite or countable sets,
- $\forall i \in \mathrm{dom}(R) : I_i \subset \mathrm{dom}(C) \wedge O_i \subset \mathrm{dom}(C) \wedge I_i \neq \emptyset \wedge O_i \neq \emptyset$,
- $\forall i \in \mathrm{dom}(R) : R_i \in \Pi(C \restriction I_i) \rightarrow \mathbb{B}(\{\langle k, x \rangle \mid k \in O_i \wedge x \in C_k\})$,
- $\forall i \in \mathrm{dom}(C) : C_i$ is finite or countable.

$\mathrm{dom}(R)$ is called the set of *processor indices* and is denoted by P.
$\mathrm{dom}(C)$ is called the set of *channel indices* and is denoted by K.

For $i \in P$: I_i is called the set of input channels of i,
O_i is called the set of output channels of i,
R_i is called the raction function of i.
For $k \in K$:C_k is called the type of channel k.

We will use these symbols strictly for the concepts defined. If we consider different des's we distinguish them by super- or subscripts.

Definition 2.
Let a des be given. Then

$$Q := \{\langle k, x \rangle \mid k \in K \wedge x \in C_k\}$$
$$S := Q \rightarrow \mathbb{N}_0$$
$$E \subset P \nrightarrow \Pi^*(C) \text{ such that } \forall e \in E :$$
$$\forall m \in \mathrm{dom}(e) \ : \ e(m) \in \Pi(C \restriction I_m).$$

Q is called the *trigger set*, S the *state space* and E the *event set.*

Note that a state is a bag over Q and that an event is an assignment of a set of triggers to a processor such that for each input channel exactly one trigger is chosen.

Definition 3.
The *event function* F of a des is a function with

$$F \in S \to I\!P(E) \text{ and}$$
$$\forall s \in S \ : \ \forall e \in E \ : \ e \in F(s) \leftrightarrow$$
$$e \neq \emptyset \wedge$$
$$\forall y \in Q \ : \ \sum(m \in \mathrm{dom}(e) \ : \ g_{e,m}(y)) \leq s(y)$$

where:

$$g_{e,m} := \lambda(y \in Q \ : \ \text{if } y \in e(m) \text{ then 1 else 0}) \ .$$

Hence $e \in F(s)$ holds only if s contains enough triggers to supply all the processors of $\mathrm{dom}(e)$. Note that $g_{e,m}$ is the characteristic function of the set $e(m)$ as subset of Q.
It is easy to verify that

$$\forall s,t \in S \ : \ s \subset t \to F(s) \subset F(t) \ .$$

Next we define the transition function: it assigns to a state s and an event $e \in F(s)$ a new state.

Definition 4.
The *transition function* T of a des satisfies:

$$T \in S \times E \nrightarrow S$$

such that

$$\forall s \in S \ : \ \forall e \in F(s) \ : \ \langle s,e \rangle \in \mathrm{dom}(T) \wedge$$
$$T(s,e) = s + \sum(m \in \mathrm{dom}(e) \ : \ R_m(e(m)) - g_{e,m}) \ .$$

The *transition relation* of a des is:

$$\{\langle s,t \rangle \in S \times S \mid \exists e \in F(s) \ : \ T(s,e) = t\} \ .$$

We use the symbol T also to denote the transition relation. It is easy to verify that the graph $\langle S,T \rangle$, where S is the state space and T the transition relation of a des, forms a basic system. If we speak of path, trace or process of a DES we mean the path, trace or process of the basic system induced by the des.
Note that we have 'true' parallellism in our model: processors may execute simultaneously. However, it is always possible to split an event into other events such that each of them triggers only one processor, and such that the successive execution of

these events, in any order, ends in the same state as the original compound event. This property is a consequence of the following theorem, which we present here without proof.

Theorem 5.
Let a des be given and let $s \in S$ and $e, e_1, e_2 \in F(s)$ such that $e = e_1 \cup e_2$ and $e_1 \cap e_2 = \emptyset$. Then $e_2 \in F(T(s,e_1))$ and $T(T(s,e_1),e_2) = T(s,e)$.

Note that $\forall s \in S : \forall e \in F(s) : \forall m \in \mathrm{dom}(e) : \{\langle m, e(m)\rangle\} \in F(s)$. Hence for some $e \in F(s)$ and some $m \in \mathrm{dom}(e)$: $e_1 := \{\langle m, e(m)\rangle\}$ and $e_2 := e \setminus e_1$ satisfy the conditions of Theorem 5. In this way we may split e into a sequence of single-processor events. Note that the characteristic of a single-processor event is that its domain is a singleton.
Let

$$\tilde{T} := \{\langle s,t\rangle \in S \times S \mid \exists e \in F(s) : e \text{ is a singleton } \wedge T(s,e) = t\} .$$

Then $\langle S, \tilde{T}\rangle$ forms also a basic system. It is easy to verify that the transitive closures of $\tilde{T}$ and T are equal and therefore we may consider the system $\langle S, \tilde{T}\rangle$ as a *simulation* of $\langle S, T\rangle$.

Now we introduce another framework to model also real-time aspects of discrete event systems, we call it RTDES. This framework is closely related to DES: a real-time des (rtdes) is a des with some more structure and some extra restrictions on events. In a des each trigger has a value and in an rtdes it has a time stamp in addition.

Definition 6.
A *real-rime discrete event system* (rtdes) is a des

$$\langle R, C, I, O\rangle$$

with the properties:

- there is a set-valued function V and an ordered set D such that: $\mathrm{dom}(V) = K$ and $\forall k \in K : C_k = V_k \times D$,

- $\forall m \in P : \forall c \in \Pi(C) :$
  $R_m(c \restriction I_m) \subset \{\langle\langle k, \langle x,t\rangle\rangle, n\rangle \mid k \in O_m \wedge x \in V_k \wedge t \in D \wedge$
  $n \in \mathbb{N}_0 \wedge (n > 0 \rightarrow t \geq \max\{d \in D \mid \exists l \in I_m : \exists v \in V_m : \langle l, \langle v, d\rangle\rangle \in c \restriction I_m\})\}$.

Hence, the time stamps of the produced triggers are at least as large as the time stamps of the consumed triggers.
The trigger set, state space and event set of an rtdes are the same as of the corresponding des.
The events that may be used in a transition are more restricted than in a des. This is because of the meaning of the time stamps. A time stamp of a trigger means that that trigger may not be consumed before that time. Hence each event has an earliest time

it may be executed. This is expressed by the function h.

Definition 7.
For an rtdes the function h assigns an *event time* to each event, such that for $e \in E$:

$$h(e) = \max\{d \in D \ : \exists m \in P \ : \ \exists k \in I_m \ : \ \exists x \in V_k \ : \\ \langle k, \langle x, d \rangle \rangle \in e(m)\} \ .$$

The chosen event must have the lowest possible event time. This expresses that processors are eager to start: as soon as they can get their triggers they execute. So for each state we may define a transition time: it is the time the system will leave the state.

Definition 8.
For an rtdes the function H assigns to each state a *transition time*, such that for $s \in S$:

$$H(s) = \min\{h(e) \ : \ e \in F(s)\} \ .$$

Now we are able to define the event function for an rtdes, it is called the real-time event function.

Definition 9.
For an rtdes the *real-time event function* FT satisfies:

- $FT \in S \rightarrow I\!P(E)$,
- $\forall s \in S \ : \ \forall e \in FT(s) \ : \ e \in FT(s) \leftrightarrow e \in F(s) \wedge h(e) = H(s)$.

This expresses that only events with lowest event time are allowable. Now we define the real-time transition relation:

Definition 10.
For an rtdes the *real-time transition relation* TR satisfies:

$$TR = \{\langle s,t \rangle \in S \times S \mid \exists e \in FT(s) \ : \ T(s,e) = t\} \ .$$

We can prove that transition times of successive states on a path are ascending.

To specify an rtdes, one often specifies the computation of values and time stamps of triggers separately. Then the values of produced triggers are independent of the time stamps of the consumed triggers. Furthermore, the time stamps are computed by a delay depending only on the values of the consumed triggers. This delay is added to the event time to obtain the time stamps of the produced triggers.

Hence, one specifies also a des, namely by leaving the time stamps aside. The rtdes realizes the des: each path of the rtdes is also a path of the des (note that the opposite does not hold).
Finally, we remark that a des or an rtdes may have starvation of triggers, i.e. some trigger is never consumed. It is the responsibility of the designer to avoid this.

3. SPECIFICATION LANGUAGE

3.1. Functional part

As mentioned in the introduction, our specification language (called EXSPECT) has a functional and a dynamic part. The functional part is used to specify functions and types. Functions and types together form a many-sorted algebra, i.e. a set of types (or sorts) and functions operating on these sorts. The functions include the constants (functions having no parameters, only a result).
Constructive specification of a many-sorted algebra means defining its types and functions by means of simpler types and functions; this process is continued until a basic level is reached that is generally understood. How to choose the intermediate levels is not clear; there exists a multitude of methods and heuristics for it. In this section we concentrate on the construction of types and/or functions from simpler ones.
For the functions, we introduce variables, expressions and the lambda quantor. This suffices (together with recursion) to define every function we need. For types we need type constructors, like e.g. the *array* constructor in procedural languages. Type constructors are accompanied by functions that go back and forth between the composite type and its constituents, like e.g. array indexing. These functions are of a *polymorphic* nature, i.e. they do not operate on individual types, but on classes of types.
From a set of types and type operators we can form *type expressions* that symbolize new (composite) types. We can attach names to type expressions, thus defining new types. The new type is a subtype of the type expression it is derived from and "inherits" all functions that could be applied to the original type expression.
Summarizing, the ingredients of the functional part are as follows.

1. Basic types and type constructors.
2. Basic functions (often polymorphic).
3. A way to construct new functions from expressions (with variables).
4. A way to construct new types from type expressions.

In the remainder of this section we shall indicate how these ingredients are realized.

We start with defining *objects*. These objects provide a semantics for types and functions. Each EXSPECT type expression denotes a set of objects, whereas functions map objects onto objects. Functions are not objects themselves, so EXSPECT is first-order. We have the following recursive definition for objects.

1. The booleans, rational numbers and character strings are objects.
2. Finite sets of objects are objects.
3. Ordered pairs of objects are objects.

The set of objects corresponding to type expression A is denoted by $s(A)$. The basic types are `void`, `bool`, `num` and `str`. These correspond to the empty set, the booleans,

the rationals and the strings, respectively. The basic type operators are *set* (denoted by a **$** prefix) and *cart* (denoted by a >< infix). If A and B are type expressions, then **$**$A$ denotes the set of finite sets of objects in $s(A)$ and A >< B denotes the set of pairs of objects, the first in $s(A)$, the second in $s(B)$.
The type operator *map* (denoted by a -> infix) is derived from set and cart. The type expression A -> B denotes the set of mappings from $s(A)$ to $s(B)$; a mapping is a finite set of pairs with different first components.
In type expressions containing these operators, *set* takes precedence over the other two and *cart* over *map*. Explicit precedence is indicated by "()" brackets.

We shall now define our set of basic functions. We indicate the signature of polymorphic functions by means of type variables, represented in this paper by the identifiers R, S and T. The semantics of a type expression with type variables is a function from assignments (functions of type variables to type expressions without variables) to sets of objects. A function of signature

$$TE_1 \times \ldots \times TE_n \rightarrow TE_0 \ ,$$

where the TE_i are type expressions, possibly containing type variables, accepts n objects in $s_v(TE_1), \ldots, s_v(TE_n)$, respectively and returns an object in $s_v(TE_0)$ for any possible assignment v. Here $s_v(TE)$ is the semantics of TE derived from assignment v. We indicate the basic constants and functions in EXSPECT by their signature and a short explanation in words. A complete algebraic specification is omitted because of its lengthy nature.

```
false, true  :  bool                    -- falsehood and truth
0,1,-1,...   :  num                     -- integers (decimal notation)
'...'        :  str                     -- strings
   -- between the above quotes any non-quote character is allowed
quote        :  str                     -- the character "'"
empty        :  $void                   -- empty set
cond         :  bool × T × T → T        -- if-then-else construction
eq           :  T × T → bool            -- equality test
lt           :  num × num → bool        -- less-than comparison
sub          :  num × num → num         -- subtraction
div          :  num × num → num         -- rational division
cat          :  str × str → str         -- string concatenation
head         :  str → str               -- first char of string
tail         :  str → str               -- string with head removed
ins          :  T × $T → $T             -- insertion in set
pick         :  $T → T                  -- "first" element of set
rest         :  $T → $T                 -- set with pick deleted
pi1          :  T >< S → T              -- projection first coordinate
pi2          :  T >< S → S              -- projection second coordinate
prod         :  T × S → T >< S          -- pair formation
```

The syntax of EXSPECT is denoted in BNF-like format. Terminals are in typewriter font. Optional parts are enclosed in "[]" brackets; parts that can be repeated without separator or with a comma "," separator are enclosed in "{ }" or "⟨⟩" brackets, respectively.
The type definitions are represented as follows.

```
tdef  :=  type id [from te]
te    :=  id | ( te ) | $ te | te >< te | te -> te
```

When a type A is defined from a type expression TE, A inherits all functions allowed for TE. Similarly, A -> B inherits all functions allowed for $(A >< B)$.
Function (constant) definitions are represented as follows.

```
def   :=  id [[⟨id : te⟩]] := expr : te
expr  :=  id [(⟨expr⟩)] | [ id : expr | expr ]
```

The definition

$$f[x : TE_1 , y : TE_2] := e : TE_0$$

defines a function of signature $TE_1 \times TE_2 \rightarrow TE_0$ given by e. The expression e may contain x and y as variables. The expression

$$g(a,b)$$

denotes the application of g onto the expressions a and b. The function g must be basic or defined and of the correct signature. The expression

$$[t : s \mid e]$$

denotes the mapping with domain s (s must be an expression denoting a set) defined by e. The expression e may contain t as a variable.

The set of basic functions can be extended with definitions for general-purpose functions, like addition, multiplication or deletion from a set. We shall give a few examples.

```
not [x:bool] :=            -- logical inversion
     cond (x, false, true) :  bool;
elt [x:T,y:$T] :=          -- test whether y contains x
     cond (eq(y,empty), false,
                        cond (eq(x,pick(y)), true,
                                             elt(x,rest(y)))) :  bool;
add [x:num,y:num] := sub (x, sub(0,y)) :  num;      -- addition
sum [x:T->num] :=          -- sum quantor
     cond (eq(x,empty), 0, add(pi2(pick(x)),sum(rest(x)))) :  num;
dom [x:T->S] :=            -- domain of a mapping
     cond (eq(x,empty), empty,
                        ins (pi1(pick(x)), dom(rest(x)))) :  $T;
set [x:T->bool] :=         -- domain restriction
     cond (eq(x,empty), empty,
                        cond (pi2(pick(x)), ins(pi1(pick(x)),set(rest(x))),
                                            set(rest(x)))) :  $T
apply [x:T->S,y:T] :=      -- application of mapping
     cond (eq(pi1(pick(x)),y), pi2(pick(x)),
                               apply(rest(x),y)) :  S;
inv [x:T->S,y:S] :=        -- inverse of mapping
     set ([t:dom(x)|eq(apply(x,t),y)]) :  $T;
sdiff [x:$T,y:$T] :=       -- set difference
     set ([t:x|not(elt(t,y)]) :  $T;
```

The above notation for expressions and type expressions, though simple in structure, is awkward for the human eye. The original syntax is sugared to allow a more readable notation. In the list below we give some hints how this sugaring is done; the other constructions are in the same vein and can be easily understood.

`cond(`a`,`b`,`c`)`	**if** a **then** b **else** c **fi**
`eq(`a`,`b`)`	$a = b$
`prod(`a`,`b`)`	$\langle\langle a,b \rangle\rangle$
`ins(`a_1`,...,ins(`a_n`,empty)...)`	$\{a_1,\ldots,a_n\}$
`empty`	$\{\}$
`apply(`f`,`a`)`	$f.a$

3.2. Dynamic part

The complete language obeys the following syntax definition, where **"expr"** is defined in the previous section.

```
sysdef   :=  sys id ppars ::= { (tdef | def | procdef | chdef) : } end
ppars    :=  [ [in (id : te)] [out (id : te)] [val (id : te)] ]
procdef  :=  proc id [ (ppars) ] := (stat)
chdef    :=  channel id [te] [:= (expr)]
stat     :=  id <- expr [: expr] | if expr then (stat) [else (stat)] fi
```

The program parameters (ppars) specify the external interface of the system. They consist of values (val) that must be specified when the system is started and input (in) and output (out) channels for interactive communication.
The channel definitions (chdef) define the internal channels of the system. By default, channels are initially empty, but an initialization can be given by providing a list of expressions after the ":=" sign in the definition. Each expression in this list represents an initial trigger value.
The processor definition consists of a processor name, followed by a list of input channels. The activation of the processor requires a trigger in each input channel. These triggers are consumed, while the processor executes a series of (conditional) assignments. An assignment of the form "*a* <- *e*" evaluates expression *e* and produces the result for channel *a* (without delay). A delay can be specified by adding a numeric expression after the ":" sign. All expressions within a "procdef" may contain the names of the input channels of the processor; while evaluating them, the consumed triggers are substituted for these names.
A formal semantics definition in terms of Chapter 2 is omitted here. The above syntax suffices for the specification of small systems. For larger systems it is possible to include systems in each other, even with recursion.

4. AN EXAMPLE

In this section we illustrate the concepts of the previous chapters by specifying a simple inventory control system. In short, this system accepts orders from customers and schedules deliveries to these customers. Also it will generate replenishment orders for goods which are out of stock and it will handle the incoming deliveries from the supplier.
The environment of the system consists of the customers and the supplier. Customers have two interaction channels to the system: one for ordering products (`cin`) and one for receiving deliveries (`cout`). In the same way there are two channels for the supplier: one for the reception of replenishment orders (`rout`) and one for the delevery of products (`rin`).
We use three types for these channels,

```
type custid;
type prodid;
type qty from num;
```

Here `custid` is the customer identification, `prodid` the name of a product and `qty` denotes the quantity which is ordered or delivered. Therefore the channels are declared as follows,

```
channel cin:    (custid><prodid)->qty;
channel cout:   (custid><prodid)->qty;
channel rout:   prodid><qty;
channel rin:    prodid><qty;
```

We see that a customer order has as key the identification of customer and product. Since we assume only one supplier per article we don't need a supplier identification for replenishment orders.
For simplicity we assign each of the external channels to a dedicated processor. Processor `pcin` handles the incoming customer orders of `cin`; `pcout` schedules customer deliveries on `cout`; `prout` orders replenishments on `rout`; `prin` handles the replenishments of `rin`.
The status of the inventory is stored in three storelike channels,

```
channel custorders:  (custid><prodid)->qty;
channel stock:       prodid->qty;
channel replorders:  prodid->qty;
```

Channel `custorders` holds the aggregated customer orders: we don't keep track of the time at which an order has been made; `stock` is the physical stock; it only contains entries for products for which the stock quantity is positive. Finally `replorders` holds the aggregated quantity which has been ordered for each product. This quantity is always non-zero.

We are now able to define the four processors. Processor `pcin` gets a customer order and updates the 'store' `custorders`.

```
proc pcin[pcin, custorders] :=
  custorders <- fupd (custorders, pi1(cin), pi2(cin));
```

The function `fupd` updates its first argument, which is a mapping, in the place denoted

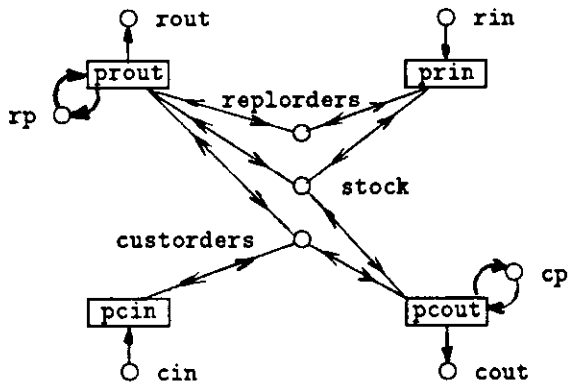

Figure 1: Inventory control system.

by the second argument by adding the third argument. Furthermore it will remove all elements which are mapped upon 0 (this is not really necessary for custorders).

```
fupd[f:A->num, x:A, y:num] :=
  clear([z:  ins(x,dom(f))
         |   if z elt dom(f) then f.z else 0 fi
             +
             if z = x then y else 0 fi]);
```

Here `clear` is a function which removes the kernel of a mapping: it restricts `g` to its domain with all points which are mapped to 0 removed,

```
clear[g:A->num] := restrict(g,dom(g)\inv(g,0));
```

The next processor we consider is `prout`, which generates orders to replenish the stock. This processor periodically triggers itself to check whether there are any products out of stock. It uses a channel `rp` of type `selftrigger`,

```
selftrigger:  type;
channel rp:  selftrigger;
```

If there are any products out of stock `prout` will order one such product. The other products which are out of stock will be handled later. When we are using the real time option it is possible to give the selftrigger `rp` a short delay when there are any other products out of stock and a long delay otherwise.

```
proc prout [rp, custorders, replorders, stock] :=
  rp <- rp,
  stock <- stock,
  if repl != {} then
    rout       <- <<pick(repl), -virtstock(pick(repl))>>,
    replorders <- fupd(replorders, pick(repl), -virtstock(pick(repl)))
  else
    rout       <- rout,
    replorders <- replorders
  fi
```

Here we use a function which gives us the virtual stock of a product, that is the quantity in stock plus the replenishment quantity already ordered from the supplier min the quantity ordered by the customers.

```
virtstock[x:prodid] :=
  if x elt dom(stock) the stock.x else 0 fi
  +
  if x elt dom(replorders) then replorders.x else 0 fi
  -
  sum[y:  $[z:  dom(custorders) | pi2(z) = x] | custorders.y]
```

Replenishments are ordered in such a way that the virtual stock of a product will be zero. `repl` is the set consisting of all products which should be replenished, `pick(repl)` is the product for which a replenishment order will be made.

```
repl := $[p:  pi2(custorders)) | virtstock(p) < 0];
```

The processor `prin` handles the incoming deliveries from the supplier. It updates the 'stores' `stock` and `replorders`. Note that the quantity in `replorders` may become negative if the supplier delivers more than has been ordered.

```
proc prin [replorders, stock] :=
  replorders <- fupd(replorders, pi1(rin), -pi2(rin)),
  stock      <- fupd(stock, pi1(rin), pi2(rin))
```

Finally we describe the processor `pcout` which handles the deliveries to the customers. It works more or less the same as `prout` by using a selftrigger channel

```
channel cp:  selftrigger;
```

and handling one delivery at a time. This could lead to starvation of certain customer orders and might be mended by numbering all incoming customer orders.

```
proc pcout [cp, custorders, stock] :=
  cp <- cp,
  if cust != {} then
    cout       <- <<pick(cust,amount(pick(cust))>>,
    custorders <- fupd(custorders, pick(cust), -amount(pick(cust))),
    stock      <- fupd(stock, pi2(pick(cust)), -amount(pick(cust)))
  else
    custorders <- custorders,
    stock      <- stock
  fi
```

Here `cust` is the set of all customer orders which may be (partially) delivered; `pick(cust)` is the order which will be handled and `amount` denotes the quantity which can be delivered for such an order.

```
cust := $[p:  dom(custorders) | pi2(p) elt dom(stock))];
amount[p:  custid><prodid] := min(stock.pi2(p), custorders.p]
```

In `cust` we have used the fact that `stock` only holds products which are really in stock (quantity positive).

LITERATURE

Bachman, C.W.: Data structure diagrams.
ACM SIGBDP, Data Base 1, no. 2 (1969).

Bergstra, J.A., J.W. Klop: The algebra of recursively defined processes and the algebra of regular processes.
Proceedings 11th ICALP, Antwerpen 1984, Springer LNCS 172 (1984).

Chen, P.P.: The entity-relationship-model: toward a unified view of data.
ACM TODS, vol. 1, no. 1 (1976).

Concepcion, A.I., B.P. Zeigler: DEVS formalism: a framework for hierarchical model development.
IEEE Transactions on Software Engineering, vol. 14, no. 2 (1988).

Genrich, H.J., K. Lautenbach: System Modelling with High-Level Petri Nets
TCS, vol. 13, North-Holland (1981).

Goguen, J.A., J.W. Thatcher, E.G. Wagner: An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types.
In: Yeh, R.T. (ed.), Current Trends in Programming Methodology, vol. IV, Data Structuring, Prentice Hall (1977).

Guttag, J.V., J.J. Horning: The Algebraic Specification of Abstract Data Types.
Acta Informatica, vol. 10 (1978).

Harel, D.: Statcharts, a visual approach to complex systems.
Sci. Comp. Progr. 8-3 (1987).

Hayes, I.: Specification case studies.
Prentice Hall (1987).

Hoare, C.A.R.: Communicating sequential processes.
Prentice Hall (1985).

Jones, C.B.: Systematic software development using VDM.
Prentice Hall (1986).

Lundeberg, M., G. Goldkuhl, A. Nilsson: A systematic approach to information systems development.
Information Systems, vol. 4 (1979).

Mazurkiewicz, A.: Traces, histories, graphs: instances of a process monoid.
Math. Found. Comp. Sci., Springer LNCS 176 (1984).

Milner, R.: A calculus of communicating systems.
Springer LNCS 92 (1980).

Petri, C.: Introduction to general net theory. Advanced course of general net theory of processes and systems.
Springer LNCS (1980).

Rem, M.: Partially ordered computations, with applications to VLSI design. Proc. Found. of Comp. Sci. IV_2, MC Tract 159 (1983).

Ross, D.T., M.E. Dickover, C. McGowan: Software design using SADT. Auerbach Publishers Portfolio 35-05-03 (1977).

Shridhar, K.T., C.A.R. Hoare: JSD expressed in CSP. Technical Monograph PRG-51, Oxford University Computing Laboratory, England (1985).

Ullman, J.D.: Principles of database systems (2nd edition). Computer Science Press (1986).

Ward, D.T., S.J. Mellor: Structured development for real-time systems. Yourdon press (1985).

In this series appeared :

No.	Author(s)	Title
85/01	R.H. Mak	The formal specification and derivation of CMOS-circuits
85/02	W.M.C.J. van Overveld	On arithmetic operations with M-out-of-N-codes
85/03	W.J.M. Lemmens	Use of a computer for evaluation of flow films
85/04	T. Verhoeff H.M.J.L. Schols	Delay insensitive directed trace structures satisfy the foam rubber wrapper postulate
86/01	R. Koymans	Specifying message passing and real-time systems
86/02	G.A. Bussing K.M. van Hee M. Voorhoeve	ELISA, A language for formal specifications of information systems
86/03	Rob Hoogerwoord	Some reflections on the implementation of trace structures
86/04	G.J. Houben J. Paredaens K.M. van Hee	The partition of an information system in several parallel systems
86/05	Jan L.G. Dietz Kees M. van Hee	A framework for the conceptual modeling of discrete dynamic systems
86/06	Tom Verhoeff	Nondeterminism and divergence created by concealment in CSP
86/07	R. Gerth L. Shira	On proving communication closedness of distributed layers
86/08	R. Koymans R.K. Shyamasundar W.P. de Roever R. Gerth S. Arun Kumar	Compositional semantics for real-time distributed computing (Inf.&Control 1987)
86/09	C. Huizing R. Gerth W.P. de Roever	Full abstraction of a real-time denotational semantics for an OCCAM-like language
86/10	J. Hooman	A compositional proof theory for real-time distributed message passing
86/11	W.P. de Roever	Questions to Robin Milner - A responder's commentary (IFIP86)
86/12	A. Boucher R. Gerth	A timed failures model for extended communicating processes

86/13	R. Gerth W.P. de Roever	Proving monitors revisited: a first step towards verifying object oriented systems (Fund. Informatica IX-4)
86/14	R. Koymans	Specifying passing systems requires extending temporal logic
87/01	R. Gerth	On the existence of sound and complete axiomatizations of the monitor concept
87/02	Simon J. Klaver Chris F.M. Verberne	Federatieve Databases
87/03	G.J. Houben J.Paredaens	A formal approach to distributed information systems
87/04	T.Verhoeff	Delay-insensitive codes - An overview
87/05	R.Kuiper	Enforcing non-determinism via linear time temporal logic specification.
87/06	R.Koymans	Temporele logica specificatie van message passing en real-time systemen (in Dutch).
87/07	R.Koymans	Specifying message passing and real-time systems with real-time temporal logic.
87/08	H.M.J.L. Schols	The maximum number of states after projection.
87/09	J. Kalisvaart L.R.A. Kessener W.J.M. Lemmens M.L.P. van Lierop F.J. Peters H.M.M. van de Wetering	Language extensions to study structures for raster graphics.
87/10	T.Verhoeff	Three families of maximally nondeterministic automata.
87/11	P.Lemmens	Eldorado ins and outs. Specifications of a data base management toolkit according to the functional model.
87/12	K.M. van Hee and A.Lapinski	OR and AI approaches to decision support systems.
87/13	J.C.S.P. van der Woude	Playing with patterns, searching for strings.
87/14	J. Hooman	A compositional proof system for an occam-like real-time language

87/15	C. Huizing R. Gerth W.P. de Roever	A compositional semantics for statecharts
87/16	H.M.M. ten Eikelder J.C.F. Wilmont	Normal forms for a class of formulas
87/17	K.M. van Hee G.-J.Houben J.L.G. Dietz	Modelling of discrete dynamic systems framework and examples
87/18	C.W.A.M. van Overveld	An integer algorithm for rendering curved surfaces
87/19	A.J.Seebregts	Optimalisering van file allocatie in gedistribueerde database systemen
87/20	G.J. Houben J. Paredaens	The R^2 -Algebra: An extension of an algebra for nested relations
87/21	R. Gerth M. Codish Y. Lichtenstein E. Shapiro	Fully abstract denotational semantics for concurrent PROLOG
88/01	T. Verhoeff	A Parallel Program That Generates the Möbius Sequence
88/02	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specification for Information Systems
88/03	T. Verhoeff	Settling a Question about Pythagorean Triples
88/04	G.J. Houben J.Paredaens D.Tahon	The Nested Relational Algebra: A Tool to handle Structured Information
88/05	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specifications for Information Systems
88/06	H.M.J.L. Schols	Notes on Delay-Insensitive Communication
88/07	C. Huizing R. Gerth W.P. de Roever	Modelling Statecharts behaviour in a fully abstract way
88/08	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	A Formal model for System Specification
88/09	A.T.M. Aerts K.M. van Hee	A Tutorial for Data Modelling

88/10	J.C. Ebergen	A Formal Approach to Designing Delay Insensitive Circuits
88/11	G.J. Houben J.Paredaens	A graphical interface formalism: specifying nested relational databases
88/12	A.E. Eiben	Abstract theory of planning
88/13	A. Bijlsma	A unified approach to sequences, bags, and trees
88/14	H.M.M. ten Eikelder R.H. Mak	Language theory of a lambda-calculus with recursive types
88/15	R. Bos C. Hemerik	An introduction to the category theoretic solution of recursive domain equations
88/16	C.Hemerik J.P.Katoen	Bottom-up tree acceptors
88/17	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable specifications for discrete event systems