Modelling of discrete dynamic systems framework and examples

Modelling of discrete dynamic systems
framework and examples

by

K.M. van Hee
G.J. Houben
J.L.G. Dietz

87/17

december 1987

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science of 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

editor: F.A.J. van Neerven

Modelling of Discrete Dynamic Systems framework and examples

by

Kees M. van Hee, Geert-Jan Houben, Jan L.G. Dietz

1. Introduction

A major problem in software and systems engineering is the precise specification of the system to be analysed or designed. A formal model of the system to be build can be considered as a specification of the system, restricted to the aspects considered in the model.

A computer-model is an implementation of a formal model. It can be used to simulate the behaviour of a modelled system. In case this system is an information system a computer-model can be used as a prototype of the system. Users or potential users of an information system usually are unable to understand a formal model of the system. With a prototype of the system they can see if their requirements are translated correctly by the system designers.

The systems we are dealing with are discrete dynamic systems. Such a system is at each moment in one of a set of states. At some moments it performs a transition to another, not necessarily different state. The number of transitions in each finite time interval is finite. A transition is triggered by one or more actions. The system may produce by each transition one or more reactions. Actions are coming from the environment of the system or they may be created by the system itself and fed back to the system.

Many real systems, including information systems, are discrete dynamic systems.

In literature there are many approaches to model discrete dynamic systems. Finite state machines, Markov chains and Petri nets are well-known examples of generic models. In [Dietz and van Hee (86)] a framework, called SMARTIE is developed. It is an extension of finite state machines combined with a modeling language based on predicate logic.

In [Harel (86)] another generalization of finite state machines combined with a graphical modeling language is presented. A different and less formal approach is found in Jackson's System Development [Jackson (83)]. In [Jackson (83)] several interesting examples are presented. In [Sridhar and Hoare (85)] some of these examples are modelled using the

language of Hoare CSP. In that paper it is suggested that CSP could provide a formal basis for the JSD method.

We feel that our approach is a powerful alternative and we will demonstrate that by treating a set of examples including an example treated by Sridhar and Hoare.

Aspects of our model of a discrete dynamic system may be displayed graphically as a data flow diagram [Ward and Mellor (85)]. In fact our framework provides formal semantics for a data flow diagram technique.

We start with the description of our model of a discrete dynamic system (dds) in Section 2. Then we describe in Section 3 our modeling language. In Section 4 we present a diagramming technique and in Section 5 we give some examples. In Section 6 we conpare our diagramming technique with two other well-known techniques. Finally, in Section 7 we draw some conclusions.

2. Model of discrete dynamic systems

Each dds is determined by a seven-tuple, the components of which will not all be known. Some of the components will be fully specified during the design phase of a dds, while others will become known during the operational phase of the system.

In this section we define the components of a dds, its behaviour and we define the aggregate of a dds. It turns out that such an aggregate is a dds itself. Hence our model allows decomposition and integration of dds'ses.

The model we present here is an extension of the one given in [Dietz, van Hee (86)]. We use the following notations:

- $P(X)$ denotes the set of all finite subsets of a set X.
- Δ denotes the symmetric difference-operator, i.e. $a \Delta b = (a \cup b) \setminus (a \cap b)$.
- dom and rng are functions that assign domain and range to a function, respectively.
- $X \rightarrow Y$ denotes the set of all functions with domain contained in X and range contained in Y.
- $(\cup j \in J : A_j)$ denotes $\{x \mid \exists j \in J : x \in A_j\}$.
- $(\cap j \in J : A_j)$ denotes $\{x \mid \forall j \in J : x \in A_j\}$.

Furthermore, we use the usual notations of set theory and symbolic logic, including $\rightarrow$ for implication.

Often we write f_x instead of $f(x)$ for a function application. We frequently use the Δ-operator and the fact that this operator is commutative and associative. Therefore, we may define for some set of sets X, $\Delta X = X_1 \Delta X_2 \Delta \cdots \Delta X_n$ for some enumeration $X_1, \ldots, X_n$ of X. Similarly for some set-valued function X, $(\Delta i \in \mathrm{dom}(X) : X_i) = X_{i_1} \Delta X_{i_2} \Delta \cdots \Delta X_{i_n}$ for some enumeration $i_1, \ldots, i_n$ of $\mathrm{dom}(X)$.

Let $\mathbb{N}$ denote the set of natural numbers. Let $\mathbb{R}^+$ denote $\{x \mid x \in \mathbb{R} \wedge x \geq \varepsilon\}$ for some fixed $\varepsilon > 0$, where $\mathbb{R}$ denotes the reals. $\mathbb{R}^+$ will be used as the time domain of dds'ses.

Definition 2.1.
A discrete dynamic system is a seven-tuple

$$<S, M, A, R, T, I, E>$$

where

- S is a set-valued function,

  for $i, j \in \mathrm{dom}(S)$ we have $i \neq j \rightarrow S_i \cap S_j = \varnothing$,
  $\mathrm{dom}(S)$ is called the set of store indices,
  for $j \in \mathrm{dom}(S)$: S_j is called the state base of store j,
  $\overline{S_j} = P(S_j)$ is called the state space of store j.

- M is a function,

  $\mathrm{dom}(M)$ is called the set of processor indices,
  for $i \in \mathrm{dom}(M) : M_i = \langle MC_i, MR_i \rangle$
  where:
  M_i is called the motor of processor i,
  MC_i and MR_i are functions,
  MC_i is called the change function of processor i,
  MR_i is called the response function of processor i.

- A is a set-valued function,

  $\mathrm{dom}(A) = \mathrm{dom}(M)$,
  for $i, j \in \mathrm{dom}(M) : i \neq j \rightarrow A_i \cap A_j = \varnothing$,
  for $i \in \mathrm{dom}(M)$ and $j \in \mathrm{dom}(S) : A_i \cap S_j = \varnothing$,
  $i \in \mathrm{dom}(M) : A_i$ is called the action base of processor i,
  $\overline{A_i} = P(A_i)$ is called the action space of processor i.

- R is a set-valued function,

  $\mathrm{dom}(R) = \mathrm{dom}(M)$,
  for $i, j \in \mathrm{dom}(M) : i \neq j \rightarrow R_i \cap R_j = \varnothing$,
  $i \in \mathrm{dom}(M) : R_i$ is called the reaction base of processor i,
  $\overline{R_i} = P(R_i)$ is called the reaction space of processor i.

- T is a function, $\mathrm{dom}(T) = \mathrm{dom}(M) \times \mathrm{dom}(M)$

  for $i, j \in \mathrm{dom}(M) : T_{ij} \in \overline{R_i} \rightarrow P(A_j \times \mathbb{R}^+)$,
  T is called the transfer function.

- I is a set-valued function, $\mathrm{dom}(I) = \mathrm{dom}(M)$

  for $i \in \mathrm{dom}(M) : I_i \subset \mathrm{dom}(S)$,
  I is called the interaction function.

- For $i \in \mathrm{dom}(M)$:

  * $MS_i = (\cup j \in I_i : S_j)$ is called the state base of processor i,
  * $\overline{MS_i} = P(MS_i)$ the state space of processor i,
  * $MC_i \in \overline{MS_i} \times \overline{A_i} \rightarrow \overline{MS_i}$,
  * $MR_i \in \overline{MS_i} \times \overline{A_i} \rightarrow \overline{R_i}$,

  and
  $\forall s \in \overline{MS_i} : MC(s, \varnothing) = MR(s, \varnothing) = \varnothing$.

- $E = \langle EU, ES, EA \rangle$ where EU, ES and EA are functions:
  - $\mathrm{dom}(EU) = \mathrm{dom}(S)$ and
    for $j \in \mathrm{dom}(S) : EU_j \in P(S_j \times \mathbb{R}^+)$,
    EU_j is called the external update set of store j
  - $\mathrm{dom}(ES) = \mathrm{dom}(S)$ and
    for $j \in \mathrm{dom}(S) : ES_j \in \overline{S}_j$,
    ES_j is called the initial state of store j.
  - $\mathrm{dom}(EA) = \mathrm{dom}(M)$ and
    for $i \in \mathrm{dom}(M) : EA_i \in P(A_i \times \mathbb{R}^+)$,
    EA_j is called the external event set of processor i.

(end of definition)

A mechanical appreciation of a dds is as follows. A dds consists of a set of processors and a set of stores. Processors are mutually connected by transaction channels and processors and stores may be connected by interconnections. The motor of a processor transforms instantaneously a set of actions into updates for the connected stores (by means of the function MC) and simultaneously it produces some set of reactions. The transformations may depend on the states of the connected stores. The state of a store may change by an update from a connected processor or by some external update. Hence an environment may influence a dds by external updates on stores and by imposing actions on the processors. The occurrence of an action at a particular moment is called an event. More than one event at a time for one processor is allowed. The output of a processor is sent to the environment and a transfer function transforms some reactions into actions with a time delay. Such a pair is sent to a processor as a new event with a time stamp equal to the sum of the processing time and the delay.

The delays are elements of $\mathbb{R}^+$ and therefore the number of transitions in a finite time interval is always finite. The events produced by a dds for itself or another processor are called internal events. They are inserted into the event agenda of the receiving processor. Initially, this agenda consists of all the external events, later on it contains also internal events. The external updates are supposed to commute; in fact we assume that each update is specified by some value from the state space of a store. If this value is denoted by s_1 and the actual state of that store is s_2, then the effect of the update will be $s_1 \Delta s_2$.

Next we define the behaviour of a dds.

Definition 2.2.
Let $\langle S, M, A, R, T, I, E \rangle$ be a dds. The process of the dds is a five-tuple

$$\langle \tau, \alpha, \rho, \sigma, \phi \rangle$$

where:

- $\tau \in \mathbb{N} \rightarrow \mathbb{R}$ and for $n \in \mathbb{N}$: τ_n is the time stamp of the n-th activation,
- α, ρ, σ, ϕ are functions with

$$\mathrm{dom}(\alpha) = \mathrm{dom}(\rho) = \mathrm{dom}(\phi) = \mathrm{dom}(M) \text{ and } \mathrm{dom}(\sigma) = \mathrm{dom}(S) .$$

For $i \in \mathrm{dom}(M)$:

- $\alpha_i \in \mathbb{R} \rightarrow \overline{A_i}$, $\alpha_i(t)$ is the action set of processor i at time t,
- $\rho_i \in \mathbb{R} \rightarrow \overline{R_i}$, $\rho_i(t)$ is the reaction set of processor i at time t,
- $\phi_i \in \mathbb{R} \rightarrow P(A_i \times \mathbb{R}^+)$, $\phi_i(t)$ is the event agenda of processor i at time t.

For $j \in \mathrm{dom}(S)$:

- $\sigma_j \in \mathbb{R} \rightarrow \overline{S_j}$, $\sigma_j(t)$ is the state of store j at time t.

These functions are defined recursively:

- $\tau_0 = 0$, for $i \in \mathrm{dom}(M)$: $\alpha_i(0) = \varnothing$, $\rho_i(0) = \varnothing$, $\phi_i(0) = EA_i$
  for $j \in \mathrm{dom}(S)$: $\sigma_j(0) = ES_j$.

Let τ_n be defined and let the functions α, ρ, σ and ϕ be defined on the interval $[0,\tau_n]$,

- $\tau_{n+1} = \min \{t \mid \exists i \in \mathrm{dom}(M) : \exists a \in A_i : \langle a,t \rangle \in \phi_i(\tau_n)\}$

and for $j \in \mathrm{dom}(S)$ let δ be defined by

$$\delta_j(\tau_{n+1}) = \sigma_j(\tau_n) \Delta (\Delta \{x \mid \langle x,y \rangle \in EU_j \wedge \tau_n < y < \tau_{n+1}\})$$

and for $i \in \mathrm{dom}(M)$ let γ be defined by:

$$\gamma_i(\tau_{n+1}) = (\cup j \in I_i : \delta_j(\tau_{n+1}))$$

and let:

- $\alpha_i(\tau_{n+1}) = \{a \mid \langle a,\tau_{n+1} \rangle \in \phi_i(\tau_n)\}$,
  for $\tau_n < t < \tau_{n+1}$: $\alpha_i(t) = \varnothing$;
- $\rho_i(\tau_{n+1}) = MR_i(\gamma_i(\tau_{n+1}),\alpha_i(\tau_{n+1}))$,
  for $\tau_n < t < \tau_{n+1}$: $\rho_i(t) = \varnothing$;
- $\phi_i(\tau_{n+1}) = \{\langle a,t \rangle \mid (\langle a,t \rangle \in \phi_i(\tau_n) \wedge t > \tau_{n+1}) \vee$
  $\vee (\exists d \in \mathbb{R}^+ : \exists k \in \mathrm{dom}(M) : t = \tau_{n+1} + d \wedge \langle a,d \rangle \in T_{ki}(\rho_k(\tau_{n+1})))\}$
  for $\tau_n < t < \tau_{n+1}$: $\phi_i(t) = \phi_i(\tau_n)$.

For $j \in \mathrm{dom}(S)$:

- $\sigma_j(\tau_{n+1}) = \delta_j(\tau_{n+1}) \Delta (\Delta i \in \mathrm{dom}(M) : S_j \cap MC_i(\gamma_i(\tau_{n+1}),\alpha_i(\tau_{n+1}))) \Delta$
  $\Delta \Delta \{x \mid \langle x,\tau_{n+1} \rangle \in EU_j\}$
  for $\tau_n < t < \tau_{n+1}$: $\sigma_j(t) = \sigma_j(\tau_n) \Delta \Delta \{x \mid \langle x,y \rangle \in EU_j \wedge \tau_n < y \leq t\}$.

(end of definition)

Note that $\delta_j(\tau_{n+1})$ is the last state of store j before τ_{n+1} and that $\gamma_i(\tau_{n+1})$ is the last state of processor i before τ_{n+1}. The state of store j at τ_{n+1} includes also the external updates at time τ_{n+1}.

Finally, we define the aggregate of a dds. This is also a dds. However, it has only one processor and one store. So it may be called a simple dds. In a top-down design-process we start in fact with a simple dds and we decompose it into a dds with more stores and processors. At the top level we do not specify much components of the dds. However, the further we decompose the system, the more details we specify. If we finally have specified at the bottom level all details of the dds, then that is also the specification of the simple dds at the top level.

Definition 2.3.
Let

$$D_1 = \langle S, M, A, R, T, I, E \rangle \quad \text{and} \quad D_2 = \langle SS, MM, AA, RR, TT, II, EE \rangle$$

be dds'ses and let # dom(SS) = # dom(MM) = 1 .
We omit the subscript in applications of the functions SS , AA etc.
Then D_2 is called the aggregate dds of D_1 if :

- $SS = (\cup j \in \mathrm{dom}(S) : S_j)$
- $AA = (\cup i \in \mathrm{dom}(M) : A_i)$
- $RR = (\cup i \in \mathrm{dom}(M) : R_i)$
- $MM = \langle MMC, MMR \rangle$ where for
  $s \in P(SS)$, $a \in P(AA)$ and
  for $i \in \mathrm{dom}(M)$: $s_i = s \cap (\cup j \in I_i : S_j)$ and $a_i = a \cap A_i$:

  $MMC(s, a) = (\Delta i \in \mathrm{dom}(M) : MC_i(s_i, a_i))$
  $MMR(s, a) = (\cup i \in \mathrm{dom}(M) : MR_i(s_i, a_i))$
- TT :
  for $r \in P(RR)$ let $r_i = r \cap R_i$, then
  $TT(r) = (\cup \langle i, j \rangle \in \mathrm{dom}(M) \times \mathrm{dom}(M) : T_{ij}(r_i))$
  (we drop the index on TT because there is only one processor)
- II = dom(SS) (a singleton)
- $EE = \langle EEA, EEU, EES \rangle$
  where
  $EEA = (\cup i \in \mathrm{dom}(M) : EA_i)$
  $EEU = (\cup j \in \mathrm{dom}(S) : EU_j)$
  $EES = (\cup j \in \mathrm{dom}(S) : ES_j)$

(end of definition)

Let $\langle \tau, \alpha, \rho, \sigma, \phi \rangle$ and $\langle \tau\tau, \alpha\alpha, \rho\rho, \sigma\sigma, \phi\phi \rangle$ be the processes of D_1 and D_2 in Definition 2.3,

respectively. Using induction one may verify that:

- for $n \in \mathbb{N} : \tau(n) = \tau\tau(n)$,

for $t \in \mathbb{R}$:

- $\forall j \in \mathrm{dom}(S) : \sigma\sigma(t) \cap S_j = \sigma_j(t)$.
- $\forall i \in \mathrm{dom}(M) : \rho\rho(t) \cap R_i = \rho_i(t)$,
  $\alpha\alpha(t) \cap A_i = \alpha_i(t)$,
  $\phi_i(t) = \{<a,t> \mid <a,t> \in \phi\phi(t) \wedge a \in A_i\}$.

Hence the aggregate has the same outputs and therefore the same external behaviour as the original dds. We could consider two dds'ses with the same external behaviour equivalent.

In practice we only specify the first six components of a dds, since we cannot look into the future to determine EU and EA. However often we know or require some properties from these sets, for example that the time lag between two events or external updates is bounded from below by some known quantity.
Such information may be used to prove properties of the behaviour of the system, i.e. of the process of the system. On the other hand we sometimes require properties of the process of a system, and then these requirements may be translated into requirements for E and therefore for the environment of the system.

In our model we assume that state transitions are executed instantaneously. This assumption is made to facilitate modeling. In practice is often impossible to implement instantaneous transitions. There are several ways to guarantee that the time lag between two transitions is longer than the time need to realise the transition in the real system. One of these methods is demonstrated in the first example of section 5. We think that this kind of modifications of a model is a next phase in the design process : first we model an ideal system, afterwards we take care of the limitations of implementations such as bounds on store sizes and execution times for state transitions.
Finally we note that our framework assumes the existence of absolute time. However the processors we model do not have the possibility to inspect some absolute clock. The absolute time we assume is just for the definition of the process of a system and may be used to express and prove properties of the dynamics of systems.

3. Modeling language

The modeling language that we introduce in this section is one of the possible ways to describe the components of a dds, defined in the framework of Section 2. Although we feel that a large class of systems can be described in this way, we do not claim that this is true for every dds.

Our modeling language consists of two parts. The first part is a first order language L that is used to describe the state, action and reaction bases and spaces. The second part is a language *PRL* for production rules, that is used to describe the motor functions.

The first order language L is constructed in the usual way (cf. [Chang and Lee (73)]). It is extended by introducing two additional quantifiers for summation and cardinality.

The alphabet consists of:

- set of variables
- set of constants, called F_0
- set of n-ary function symbols, called F_n for $n \in \mathbb{N}$ and $n > 0$
- set of n-ary predicate symbols, called P_n for $n \in \mathbb{N}$, $P = (\cup n \in \mathbb{N} : P_n)$
- quantifiers: $\exists$, $\forall$, Σ, #
- logical operators: $\vee$, $\wedge$, $\neg$, $\rightarrow$, $\leftrightarrow$
- relational operators: $<$, $>$, $\leq$, $\geq$, $=$, $\neq$
- arithmetic operators: $+$, $-$, $*$, $\underline{\text{mod}}$, $\underline{\text{div}}$
- punctuation symbols: (,), :, ,, { , }, |

Terms are defined by:

- constants and variables are terms;
- if $t_1,\ldots,t_n$ are terms and $f \in F_n$, then $f(t_1,\ldots,t_n)$ is a term;
- if t_1 and t_2 are terms, then (t_1+t_2), $(t_1 * t_2)$, (t_1-t_2), $(t_1 \,\underline{\text{div}}\, t_2)$ and $(t_1 \,\underline{\text{mod}}\, t_2)$ are terms;
- if t is a term, x is a variable and Q a formula, then $(\Sigma x : Q : t)$ and $(\#x : Q)$ are terms.

Atoms are defined by:

- if $t_1,\ldots,t_n$ are terms and $p \in P_n$, then $p(t_1,\ldots,t_n)$ is an atom;
- if t_1 and t_2 are terms, then $(t_1 \leq t_2)$, $(t_1 \geq t_2)$, $(t_1 < t_2)$, $(t_1 > t_2)$, $(t_1 = t_2)$ and $(t_1 \neq t_2)$ are atoms.

Formulas are defined by:

- an atom is a formula;
- if Q and R are formulas, then $(Q \vee R)$, $(Q \wedge R)$, $(\neg Q)$, $(Q \rightarrow R)$ and $(Q \leftrightarrow R)$ are formulas;
- if Q is a formula and x is a variable, then $(\forall x : Q)$ and $(\exists x : Q)$ are formulas.

Note that, when no problems arise, parentheses are often omitted.
In formulas free and bounded variables are distinguished, in the usual way.

To give formulas a (formal) interpretation (cf. [Lloyd (84)]), we choose the set of integers as the domain of interpretation. This means that every constant is mapped to an integer and every variable is given an integer value, but this restriction to integers is only made for convenience and is not essential.

For describing the state base of a store or an action- and reaction base of a processor we choose a subset of the predicate symbols P. The bases are defined as the sets of all ground atoms with corresponding predicate symbols. All sets of predicate symbols for base-definitions should be mutually disjoint.
Remember that the state space of a processor is the union of the state bases of stores with which it is connected.
We assume that relational and arithmetic operators have their usual interpretation, as have the logical operators and quantifiers. For each processor i with action- and reaction bases defined with predicate symbol sets PA_i and PR_i respectively and a state base defined with predicate symbols PS_i, a subset PD_i of P is defined that is disjoint with PA_i, PR_i and PS_i. The predicate symbols in PD_i are used for shorthands in the description of the motor of i.
For each processor i a set of closed formulas D_i is defined. Formulas in D_i may contain predicate symbols of $PD_i \cup PA_i \cup PR_i \cup PS_i$. These formulas are considered to be axioms; they have the truth value true. These formulas serve as definitions for shorthands or as constraints on states and actions. The set D_i is called the axiom base of processor i.

We follow the closed-world assumption (cf. [Reiter (84)]), which implies that, given some state s and some action a all ground atoms in s and a have the truth value true, whereas all other ground atoms that can be formed by predicate symbols from the corresponding bases have the truth value false.

We require that a processor i never has to deal with a state or an action that is in contradiction with D_i. It is the responsibility of the designer of a system to prove that a dds fulfills this requirement. Usually, this is done by showing that, given a state and an action, that do not contradict D_i, the new state does not contradict D_i either.

The definitions in D_i are closed formulas of the kind:

$$\forall x_1 : \cdots : \forall x_n : p(x_1, \ldots, x_n) \leftrightarrow Q \ ,$$

where $p \in PD_i$ and Q is some formula involving at most $x_1, \ldots, x_n$ as free variables and predicate symbols from $PD_i \cup PA_i \cup PR_i \cup PS_i$.
Each predicate symbol of PD_i occurs exactly once in such a formula on the left-hand side.

It is again the designer's responsibility to guarantee that for each ground atom, with its predicate symbol in PD_i, a truth value can be determined w.r.t. some state s of processor i and some action a for processor i.

Given D_i, some state s of processor i and some action a for processor i, the motor M_i may change the state and therefore the formal interpretation of formulas. Such a change of state consists of additions and deletions of ground atoms with predicate symbols from PS_i.
A deletion means that the negation of that atom gets truth value true. This can never cause a contradiction with a definition in D_i, but it may create contradictions with constraints. If an axiom base, a state and an action are considered to be axioms of a theory, then a transition may change this theory into a new one.

Now we can define the language of production rules PRL, for describing the motor of a processor.
First we define formally the language's syntax; its semantics will be defined informally afterwards.

In PRL, formulas of L occur.
The non-terminals <formula> and <atom> refer to formulas and atoms of L resp.
Using EBNF-notation we define:

<rule>	::=	<condition><D -part><I -part><R -part>
<condition>	::=	$\models$ <formula>
<D -part>	::=	$\overset{D}{\Rightarrow}$ <atom set>
<I -part>	::=	$\overset{I}{\Rightarrow}$ <atom set>
<R -part>	::=	$\overset{R}{\Rightarrow}$ <atom set>
<atom set>	::=	[<enumerated set> | <conditional set> | <atom set> $\underline{\cup}$ <atom set >]
<enumerated set>	::=	$\underline{\{}$ <atom list> $\underline{\}}$
<atom list>	::=	<atom> { $\underline{,}$ <atom>}
<conditional set>	::=	$\underline{\{}$ <atom> $\underline{\mid}$ <formula> $\underline{\}}$

Note that '$\models$', '$\overset{D}{\Rightarrow}$', '$\overset{I}{\Rightarrow}$', '$\overset{R}{\Rightarrow}$' and the underscored symbols are terminals.

An example:

$$\models \; ((p(1,3) \wedge q(0)) \vee r(x,y,z))$$
$$\overset{D}{\Rightarrow} \; \{q(8), r(7)\} \cup \{p(2*x,y) \mid t(y) \wedge y \geq x\}$$
$$\overset{I}{\Rightarrow}$$
$$\overset{R}{\Rightarrow} \; \{m(x,z) \mid t(x)\}$$

There may occur free variables in a rule, i.e. in the formula of the condition or in an atom set. However, the free variables in an atom set have to occur also in the condition. Note that a variable that occurs as a free variable before and after the bar in a conditional set is bounded.
With each state and action of a processor we associate an active domain. This is the set of all constants that occur in the axiom base, the state or the action. The active domain and the set of all variables occurring in the description of a motor or store are both finite. A binding of a set of variables is a function with this set of variables as domain and the active domain as range.

The semantics of a rule are as follows.
For each binding of the free variables of the formula in the condition of a rule, it is checked whether this formula is true, with respect to the formulas in the axiom base, the (current) state and the action. If it is true, then the atom sets of the D-, I- and R-parts are computed, where for free variables in these atom sets the values defined by the binding are substituted. The quantifications over bounded variables in closed formulas and conditional sets are computed also with respect to the active domain, so these quantifications are computable. The computation of a conditional set is as usual.
Denote for rule n and binding b of the free variables of the condition of the rule, the sets of ground atoms computed in the D-, I- and R-part by $D_{n,b}$, $I_{n,b}$ and $R_{n,b}$, respectively.
Then we define:

$$C = (\Delta n : (\cup b : D_{n,b})) \; \Delta \; (\Delta n : (\cup b : I_{n,b})) \, ,$$

$$R = (\cup n : (\cup b : R_{n,b})) \, .$$

For a transition of a motor is now defined for state s and action a:

$$MC(s,a) = s \; \Delta \; C$$

$$MR(s,a) = R$$

so the state is changed by taking the symmetric difference of the old state and for all rules, the union, over all bindings of the free variables in the conditions in the condition of the rule, of the computed sets of the D- and I-parts, whereas the reaction is just the union over all computed sets of R-parts for all rules and bindings.
Note that the distinction between D- and I-parts is only made for convenience.

4. The diagramming technique

A tuple $<S, M, A, R, T, I, E>$ defined, as was explained in Section 2, a network of processors and stores, connected to each other by means of transaction channels and interconnections. Such a network is called a dds. In designing a dds, it appears to be helpful to use the diagramming technique, explained hereafter. This technique also allows for composition and decomposition of dds'ses.

Figure 4.1 shows the symbols used by the diagramming technique.

Figure 4.2 shows an example of a diagram of a dds at the lowest level of aggregation, which means that every processor is represented by a thin-lined box, that feedback channels are drawn as transaction channels, starting from and ending on a box, and that all stores are drawn separate from the processors.

A dds-diagram actually exhibits three aspect-diagrams. One of them is the element-contents, i.e. the distinct processors and stores.

The second aspect-diagram shows the communication-structure, i.e. the interconnections between processors and stores. The third aspect-diagram shows the interaction-structure, i.e. the transaction channels between processors (including the feedback channels).

A dds is connected to its environment by means of incoming and outgoing transaction channels, and by means of interconnections with processors in the environment.

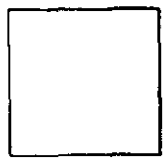
The existence of incoming transaction channels can be deduced from the component EA, whereas the existence of (inspection and change) interconnections with processors in the environment can be derived from the component EU.

The existence of outgoing transaction channels cannot be deduced from the dds definition, but may be known by chance e.g. because the dds considered is a subset of a known larger dds.

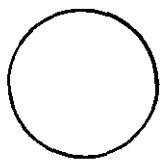
Both the incoming and outgoing transaction channels and the interconnections mentioned are not part however of the dds. Therefore, they are drawn with dashed lines (see Fig. 4.2).

In practice, one usually will describe a dds at several levels of aggregation. In that case, it is convenient to divide the (aggregate) store of a dds into two parts: the common store and the private store. The common store is the part, that has interconnections with processors in the environment. The other part is called the private store.

DIAGRAM SYMBOLS



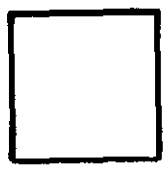
processor



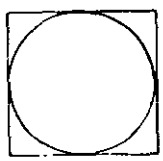
store

interconnection

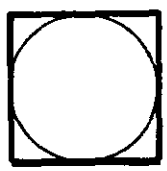
transaction channel



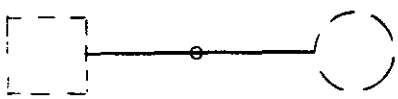
processor with feedback transaction channel



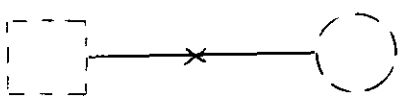
processor with private store



processor with feedback and private store

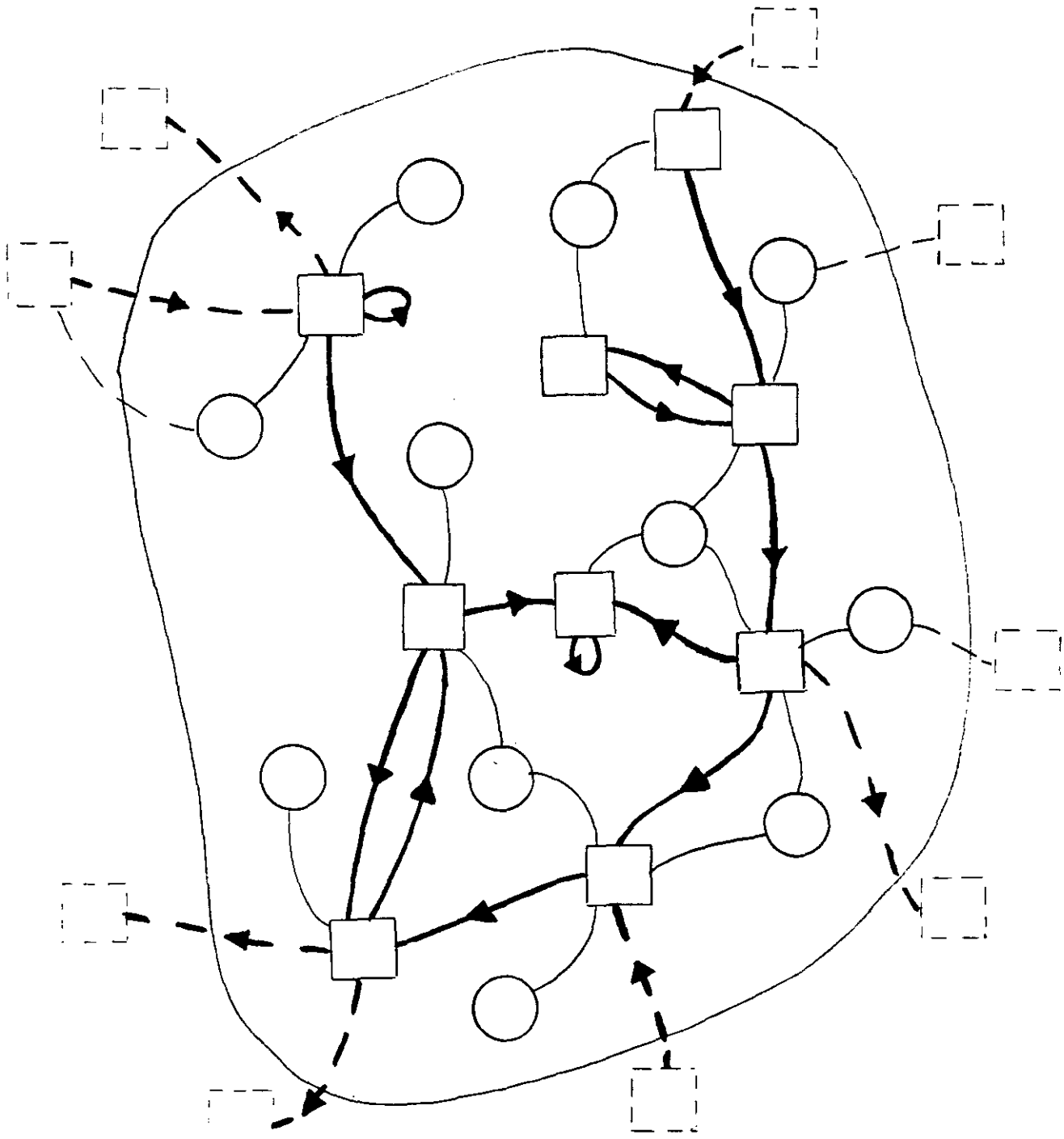


inspection-only interconnection



inspection-and-change interconnection

FIG. 4.1



interaction structure
communication structure

FIG. 4.2

In most cases, while constructing a hierarchy of (sub-) dds'ses, one is able to define the common store of a dds, whereas the processor and also the private store can only be identified (often, one will not even be sure that there is a private store).

Therefore, the most practical concise representation of a dds is the one depicted in Figure 4.3.

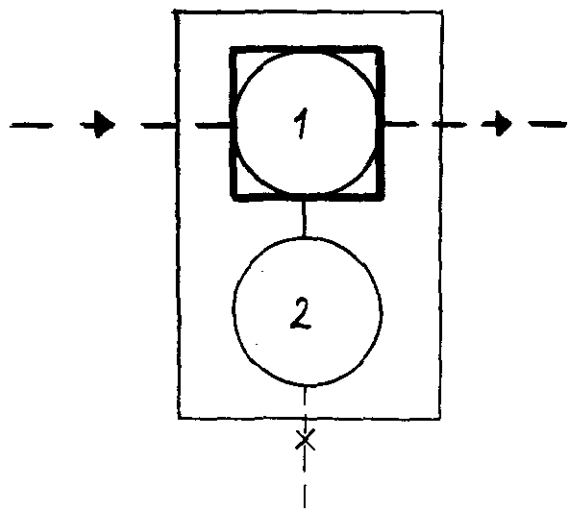


FIG. 4.3

In this diagram, the circle with number 1 represents the private store of the dds, whereas the circle with number 2 represents its common store.

In Section 5, examples of dds diagrams will be given.

5. Examples

In this section we present three examples. The first one shows how to deal with queues. The second one treats a banking system similar to the example treated in [Sridhar and Hoare (85)]. The third one is an inventory control system.

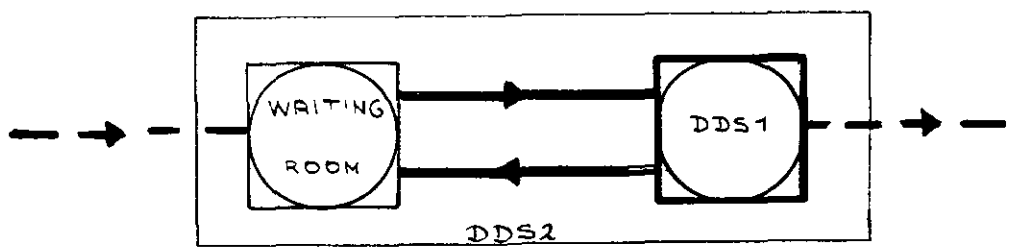
5.1. Queuing system

A processor will be activated instantaneously when a non-empty set of actions arrives. In practice many systems can process only one action at a time and moreover each processing takes time, so that the system is unable to handle new arriving actions during the busy period.

In principle it is possible to modify a given dds with instantaneously reacting processors into a system that simulates processing time and fifo-queueing order. To do this one needs to modify the processors such that they give themselves feedback events telling that the processing is finished and one has to keep an administration of the waiting actions.

Instead of modifying processors we add to the dds another dds, which is a simple dds and which behaves as a waiting room for the original dds. A nice feature is that the waiting room we describe here can be defined independent of the characteristics of the original dds. Hence, the waiting room may be considered as a standard dds construction.

Let a dds be given. Assume that it is a simple dds. Remember that every dds can be aggregated into a simple dds. Let us call this dds1. The composition of dds1 and the dds, called waiting room, is also a dds. This dds is called dds2.



From dds2 we only specify the processor and the store of the waiting room and we specify the action base of dds1. Further, we specify some properties of the transfer function T of dds2. We assume that the motor of dds1 produces upon arrival of an action a reaction that contains a ground atom of the form ready(d) where ready is a predicate symbol not used elsewhere in dds1 and where d is an integer that indicates the processing time of the just arrived action. This quantity d may depend on the state of dds1 and the received action as well. The transfer function interprets d as a delay. Further, we assume that the transfer function transforms such a reaction into a pair $<ok, d>$ where ok is a ground atom (with nullary predicate symbol) of the action base of the waiting room and d is the delay.

For simplicity we assume that the action base of dds1 consists of ground atoms with only one unary predicate symbol : p.

Now we are able to specify the waiting room. We call the state base of the store of the waiting room S and the action- and reaction bases A and R respectively. For S we use a binary predicate symbol ps, for A a unary predicate symbol pa and a nullary predicate symbol ok and for R a unary predicate symbol pr.
Variables are x, y, z, k, l, m and constants are the integers.
Further we define an axiom base:

- $\forall y : \forall m : \text{rank}(y,m) \leftrightarrow (pa(y) \wedge (\#z : pa(z) \wedge z \leq y) = m$,
  hence $\text{rank}(y,m)$ means that in the action set there are m ground atoms with a constant smaller or equal to y.
- $\forall k : \max(k) \leftrightarrow \exists y : ps(y,k) \wedge \neg\exists y : ps(y,k+1)$
  this means that in the state the maximal constant occurring in a ground atom is k, i.e. the highest scheduling number is k.
- $\max(0) \leftrightarrow \forall y : \forall k : \neg ps(y,k)$
  this means that if there is no ground atom in the state, then max(0) holds, and the queue is empty.

Now we formulate the rules:
Note that we omit a $D-$, $I-$ or $R-$ part of a rule if the corresponding set is empty.

1. $\underset{I}{\models}\ pa(x) \wedge \max(k) \wedge \text{rank}(x,m)$
   $\Rightarrow\ \{ps(x,k+m)\}$

2. $\underset{R}{\models}\ pa(x) \wedge \max(0) \wedge \text{rank}(x,1)$
   $\Rightarrow\ \{pr(x)\}$

3. $\underset{D}{\models}\ ok \wedge ps(x,k) \wedge \neg\exists y : ps(y,k-1)$
   $\Rightarrow\ \{ps(x,k)\}$

4. $\underset{R}{\models}\ ok \wedge ps(x,l) \wedge \exists y : ps(y,l-1) \wedge \neg\exists y : ps(y,l-2)$
   $\Rightarrow\ \{pr(x)\}$

Finally, we specify that the transfer function transforms

$$pr(x) \quad \text{into} \quad \langle p(x), \varepsilon \rangle .$$

Note that Rule 1 specifies that upon receiving an action this action is stored with a scheduling number equal to the maximal number in the store plus the rank of the action atom in the set of action atoms in the action. Rule 2 specifies that if the queue was empty, then the action can be

sent to dds1 directly. Rules 3 and 4 specify what has to be done upon receiving an *ok* signal of dds1. Rule 3 specifies that the action to which the *ok* reflects should be deleted and Rule 4 sends the next action if there is one.

5.2. Banking

A very simple account current system is described, adopted from [Sridhar and Hoare (85)]. The structure of this system may be depicted like it is done in Fig. 5.2.

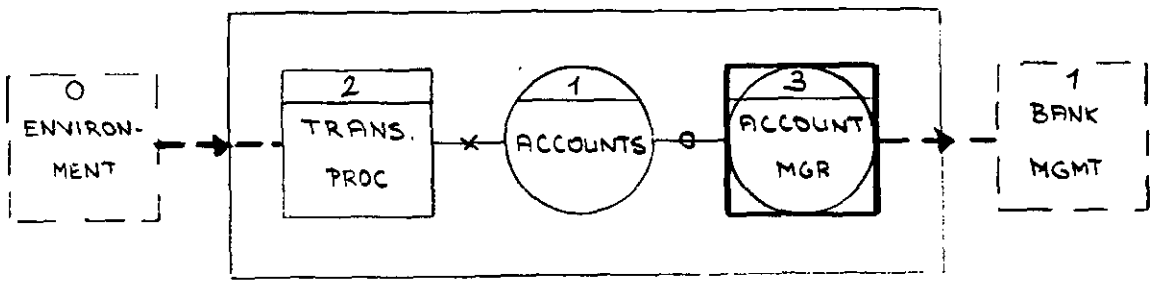


FIG. 5.2.1

Processor 2 receives banking transactions, originating from the account holders, and makes appropriate changes to the balances in Store 1.
Processor 3 periodically reads the contents of Store 1 and produces a balance report, which is sent to the bank management.
Store 1 contains ground atoms of type balance (i,x) where i is the account number and x the value of the balance.
Processor 2 can be specified as follows:

$A = \{\text{invest}(\cdot,\cdot), \text{payin}(\cdot,\cdot), \text{withdraw}(\cdot,\cdot), \text{terminate}(\cdot)\}$

$R = \varnothing$

M:

$\models_I \text{invest}(i,x) \wedge \text{no–account}(i)$
$\Rightarrow \{\text{balance}(i,x)\}$

$\models_D \text{terminate}(i) \wedge \text{balance}(i,x) \wedge x = 0$
$\Rightarrow \{\text{balance}(i,x)\}$

$\models$ balance(i,x) $\wedge$ $\neg$invest(i,y) $\wedge$ $\neg$terminate(i)

$\overset{D}{\Rightarrow}$ {balance(i,x)}

$\overset{I}{\Rightarrow}$ {balance(i,z) | $z = x + (\Sigma w$: payin(i,w) : $w) - (\Sigma w$: withdraw(i,w) : $w)$}

D = {no–account(i) $\leftrightarrow$ $\neg(\exists x$: balance(i,x)) $\wedge$
$\neg$ (terminate(i) $\wedge$ ($\exists x$: balance(i ,x) $\wedge$ $x > 0$))}

Processor 3 can be specified as follows:

A = {makereport}

R = {report(i,x),doreport(t)}

M:

$\models$ makereport $\wedge$ balance(i,x)

$\overset{R}{\Rightarrow}$ {report(i,x)}

% note that his rule is executed for all accounts, if a makereport action is received %

$\models$ makereport

$\overset{R}{\Rightarrow}$ {doreport(t)}

% note that this rule is executed only once when a makereport action is received %

T({doreport(t)}) = {<makereport,t>}

% via the feedback transaction channel a next makereport action is transferred, that will be received t time units later %

5.3. Inventory control

In this example, some of the operations of a warehouse company are considered, namely those dealing with inventory control. Globally spoken, this encompasses the delivery of goods on order to customers and the timely refreshing of a stock of goods.
The warehouse management provides general outlines for performing inventory control, and sets global parameters, which may be changed during operation. These parameters include:

- the set of customers, that are allowed to order goods,
- the set of products, that are deliverable,
- delivery and replenishment control parameters,

- time data, e.g. the today's date.

The general structure of the inventory control system is outlined as an (aggregate) dds in Figure 5.3.1.

Processor 1 (INVENTORY CONTROL) interacts with the environment of the warehouse by receiving orders from customers and replenishments from suppliers, and by issuing deliveries to customers and replenishment orders to suppliers. It communicates with Processor 2 (WAREHOUSE MANAGEMENT) via Store 2, containing the global control parameters. From the point of view of Processor 1 this communication is passive, i.e. it has an inspection-only connection with the store.

The dds of Fig. 5.3.1 may be decomposed into the one, depicted in Fig. 5.3.2. Store 2 is subdivided into five stores, and Processor 1 is decomposed into four processors and three stores through which these processors communicate. The structure shown in Figure 5.3.2 is a particular choice, many other decompositions would be equally acceptable. Its operation can be described as follows.

Orders, originating from customers are processed by Processor 1.1 and, if accepted, added to Store 1.6. Processor 1.2 periodically checks whether orders, kept in Store 1.6, can be delivered and, if so, produces deliveries, removes the corresponding customer orders from Store 1.6, and makes appropriate changes to the stock (Store 1.5). Processor 1.2 is also activated every time a replenishment is received by Processor 1.3. This processor makes, when activated, appropriate changes to Store 1.5 and removes the corresponding replenishment orders from Store 1.7.

Processor 1.4 periodically checks the stock and produces replenishment orders if necessary.

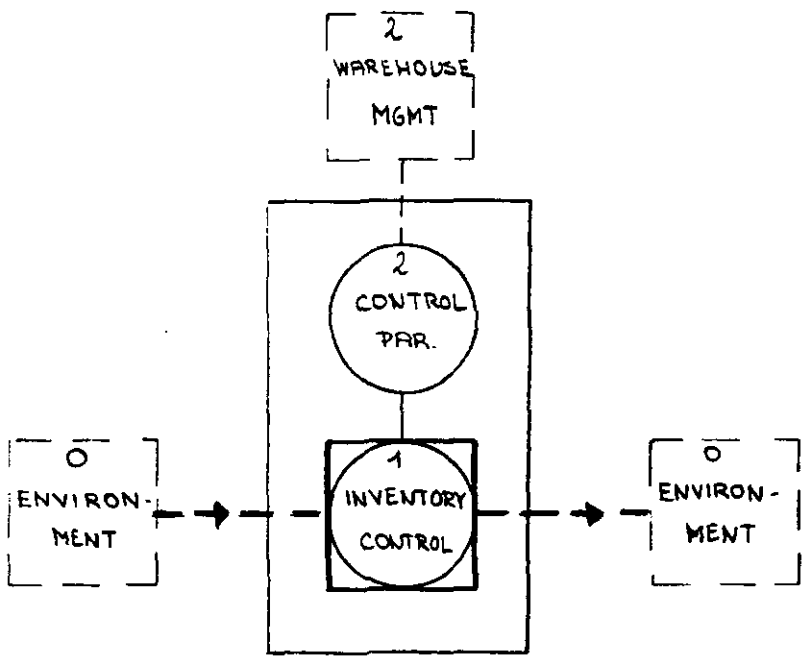


FIG. 5.3.1

The store state bases are defined by the next table

store id	store name	predicate symbol	arity
1.5	STOCK	qoh	2
1.6	CUSORDERS	cusorder	1
		newcusorder	1
		oldcusorder	1
		ordercust	2
		orderprod	2
		orderquant	2
		orderdate	2
1.7	REPORDERS	reporder	1
		newreporder	1
		oldreporder	1
		repprod	2
		repquant	2
2.1	PRODUCTS	product	1
2.2	CUSTOMERS	customer	1
2.3	TIME DATA	today	1
2.4	REPL. PARAM.	minqoh	2
		normq	2
		repinterval	2
2.5	DELIVER PARAM.	delinterval	2

As an example of the specification of a processor, Processor 1.4 (REPLENISHMENT-ORDERING) is described below. It is firstly decomposed into a Processor 1.4.2, which does the actual ordering and a queueing system (Processor 1.4.1), which ensures that Processor 1.4.2 gets replenish actions one at a time.

P1.4.2: REPLENISHMENT-PROCESSING

$A = \{\text{replenish}(\cdot)\}$

$R = \{\text{suporder}(\cdot,\cdot,\cdot), \text{doreplenish}(\cdot,\cdot)\}$

$I = \{1.5, 1.7, 2.4\}$

M:

$$\models \text{replenish}(p) \wedge \text{qoh}(p,s) \wedge \text{minqoh}(p,q) \wedge (s + \Sigma x : \text{reporder}(x) \wedge \text{repprod}(x,p) \wedge \text{repquant}(x,y) : y) < q \wedge \text{normq}(p,n) \wedge \text{newreporder}(r)$$

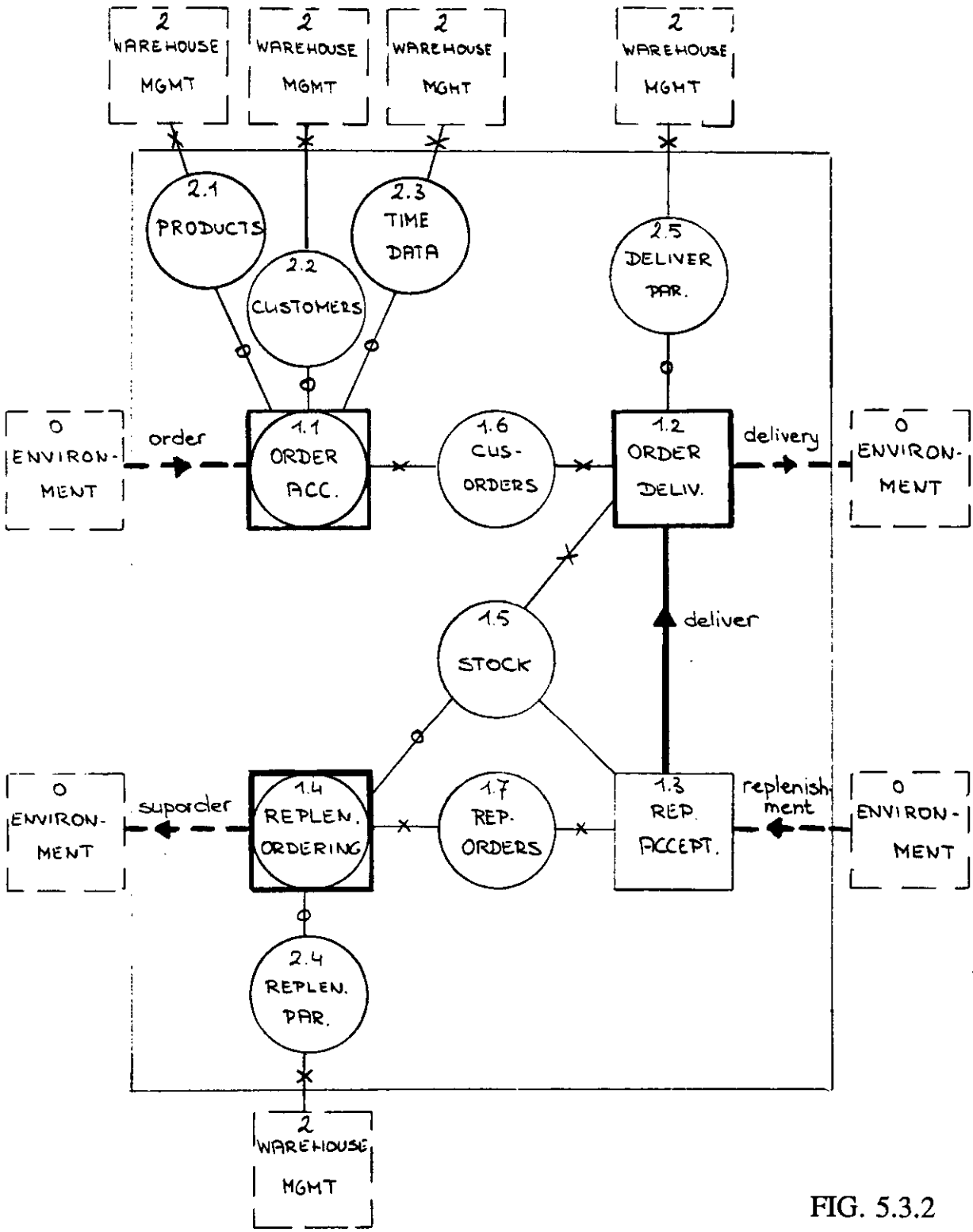

2 WAREHOUSE MGMT
2 WAREHOUSE MGMT
2 WAREHOUSE MGHT
2 WAREHOUSE MGHT
2.1 PRODUCTS
2.2 CUSTOMERS
2.3 TIME DATA
2.5 DELIVER PAR.
0 ENVIRON-MENT
order
1.1 ORDER ACC.
1.6 CUS-ORDERS
1.2 ORDER DELIV.
delivery
0 ENVIRON-MENT
1.5 STOCK
deliver
0 ENVIRON-MENT
suporder
1.4 REPLEN. ORDERING
1.7 REP. ORDERS
1.3 REP. ACCEPT.
replenish-ment
0 ENVIRON-MENT
2.4 REPLEN. PAR.
2 WAREHOUSE MGMT

FIG. 5.3.2

% if there is a replenish action for product p and the quantity-on-hand of p augmented with the quantity currently being replenished is less than the minimum quantity-on-hand for p, and if r is a free reporder number, then %

$\overset{D}{\Rightarrow}$ {newreporder(r)}

$\overset{I}{\Rightarrow}$ {reporder(r), repprod(r,p), repquant(r,n)}

$\overset{R}{\Rightarrow}$ {suporder(r,p,n)}

% r becomes the number of a reporder with product p and quantity n, and a corresponding reaction (suporder) is produced %

$\models$ replenish(p) $\wedge$ repinterval(p,t)

$\overset{R}{\Rightarrow}$ {doreplenish(p,t)}

% every time a replenish action is received, a doreplenish reaction is produced which will cause a next replenish action t time units later %

As an example of the specification of a transfer function, the transaction channel from P1.4.2 to P1.4.1 is described:

$$T(\{\text{doreplenish}(p,t)\}) = \{<\text{do}(p),t>\} ,$$

assuming that 'do' is the appropriate predicate symbol in the action base of P.1.4.1.

6. Comparison with other diagramming techniques

6.1. Comparison of the dds-diagram and the A-graph

The ISAC methodology [Lundeberg, Goldkuhl and Nilsson (79)] uses a diagramming technique, called the A-graph to show the input, the output and the subsystem structure of systems. At any aggregation level the subsystems are denoted by nodes. As an example for comparison, the dds-diagram of Figure 5.3.2 is "translated" into the A-graph of Figure 6.1..

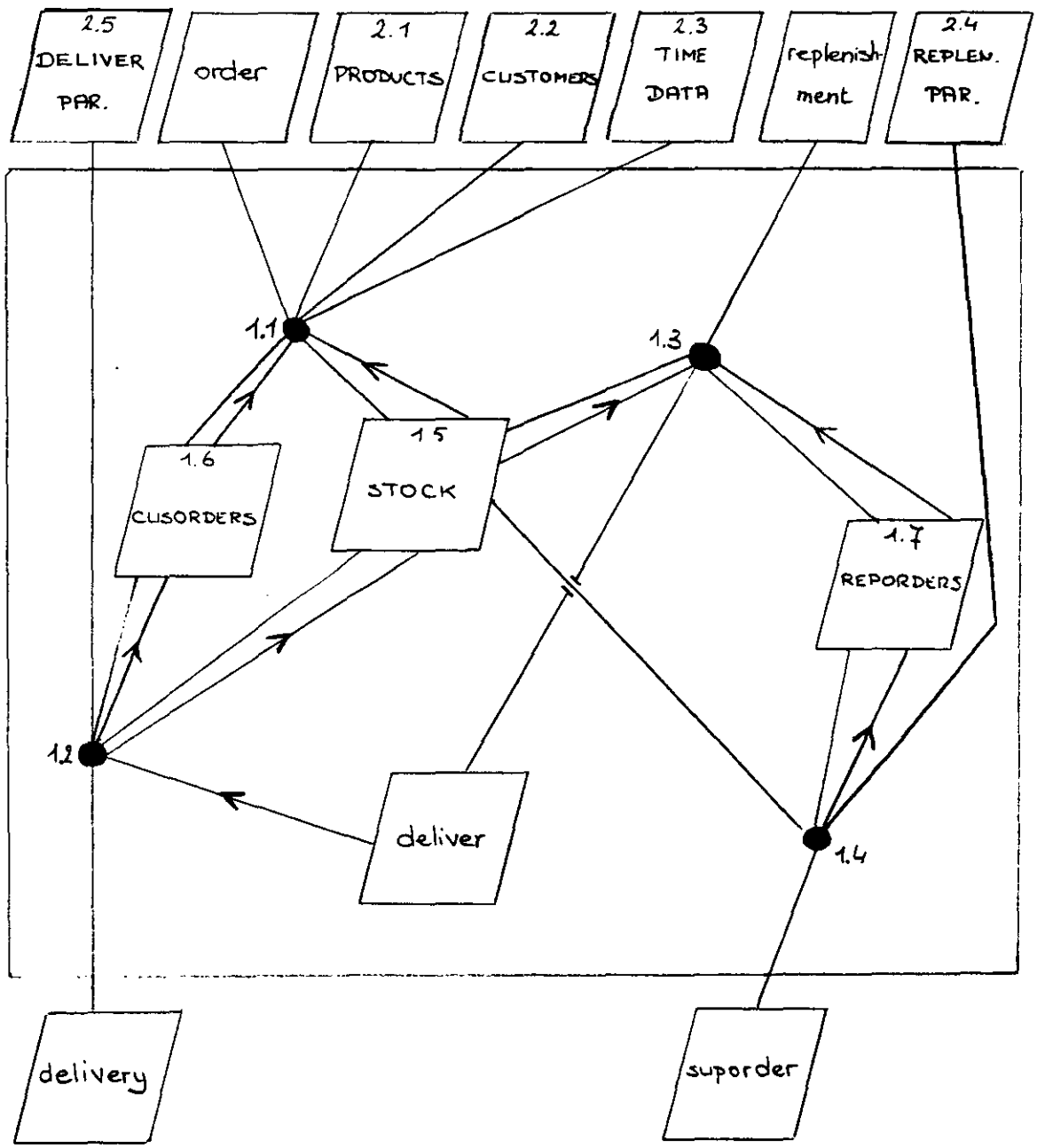


FIG. 6.1

The most important differences between the two techniques are, in our view, the following ones:

- The generic model behind the A-graph has two basic concepts: activities and flows. The activity concept fits very well in our concept of a processor, whereas the flow concept concords with the concept of a transaction channel. There is however no formal foundation of the generic model.
- The notion of communication is missing in the A-graph. This give rise to a modelling problem with regard to common permanent sets, like the sets 2.1 through 2.5 in Figure 6.1.. They may not be considered input flows, as "order" and "replenishment" are. An alternative would be to put these sets within the system boundary. However, in that case, the maintenance of these sets would become the responsibility of the system, and that would lead to additional input flows and additional activities, which are alien to the primary purpose of the system.
- The process of a system is ill-defined. For example, it is not clear how activity 1.4 is triggered, whereas the missing of this aspect becomes misleading for activity 1.2 !

6.2. Comparison of the dds-diagram and the DFD

The DFD (Data Flow Diagram) is a graphical presentation technique, that is especially used by the "structured" analysis and design approaches (Yourdon, DeMarco, Ward etc.). One of the first publications is [Gane and Sarson (77)]. For the ease of discussion, the dds-diagram of Figure 5.3.2. is also "translated" into a DFD (Figure 6.2.). The most important differences between the two techniques are, in our view, the following ones:

- The generic model behind the DFD has three basic concepts: data files, data flows and data processors. The original focus on data, i.e. on the syntactic aspect of information, appears to be a major drawback of the technique, when it has to be used for the modelling of non-data-systems. As can be seen from Figure 6.2., every "relationship" between processors and files and between processors among each other becomes a data flow. Also, the "reading" of data in a file is a flow.
  When one accepts these drawbacks, the basic concepts of file and processor may be considered to concord with store and processor respectively. The concept of flow however "translates" to transaction channel as well as to interconnection. There is however no formal foundation of the generic model.
- Although interconnections can be modelled in the DFD, the notion of communication as defined in our framework is missing. Among others, this gives rise to the same modelling problem of the maintenance of the files 2.1 through 2.5 as were discussed in section 6.1..

- The process of a system is also ill-defined. The same remarks as were made in section 6.1. hold for the DFD.

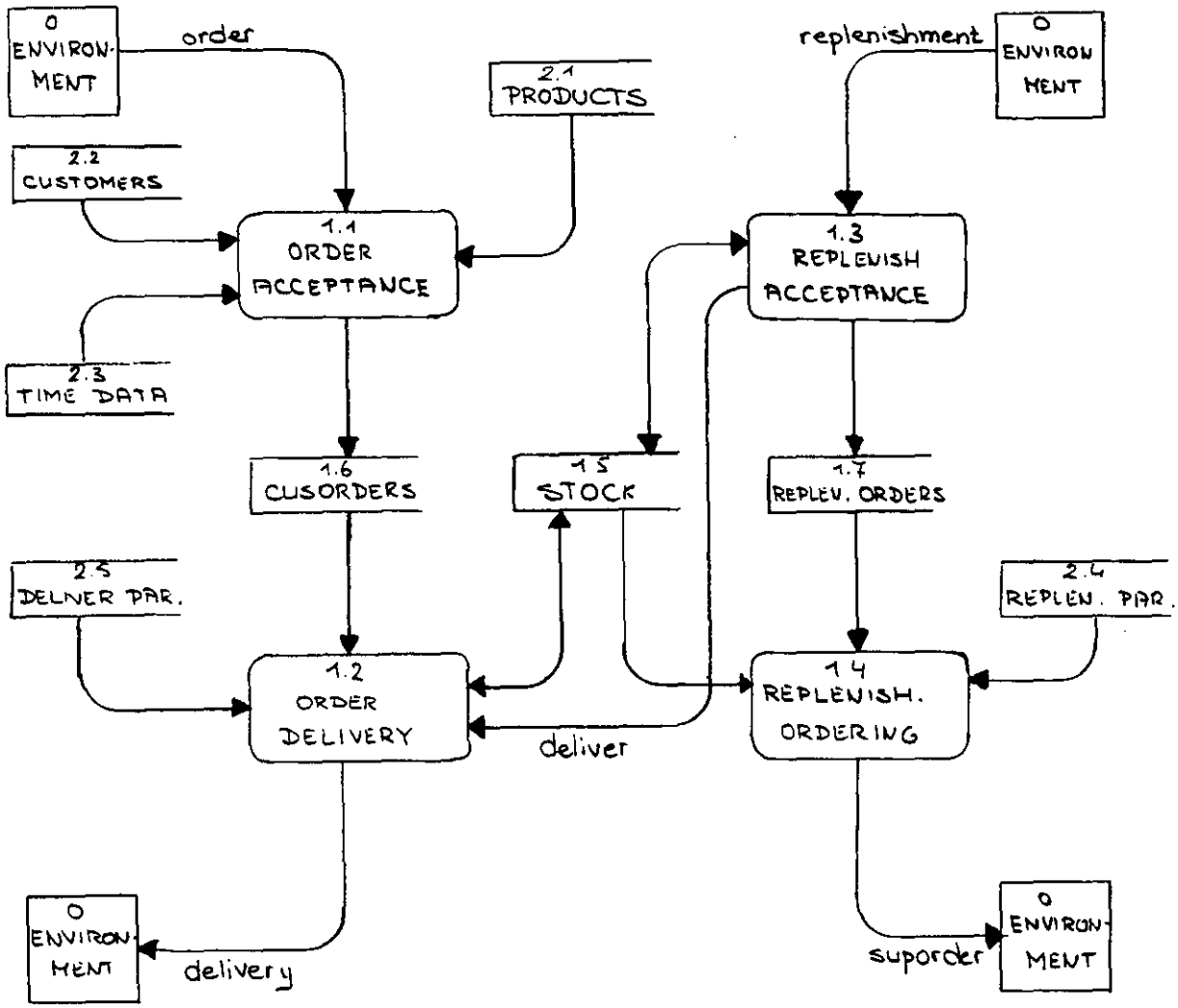


FIG. 6.2

7. Conclusions and future research

We have developed a framework for the formal description of systems of a large class, including information systems. In the framework data modelling and process specification are completely integrated. Hierarchical decomposition is possible. A diagram technique, which is in fact a data flow diagramming technique, is proposed. The language for system description we proposed in this paper is powerful, but may be replaced by many others. For instance, the data modelling can be replaced by the relational model, the entity relationship model, a binary model or by a functional data model. The process modelling may be replaced by any third generation programming language or by a functional language.

The framework is also useful for the development of simulation models of physical systems. The framework may be used for formulating and proving temporal properties of a dds. This issue requires some further research.

Another research topic is the extension of the framework to allow for the creation and starvation of copies of a dds. This extension will allow for the application of object-oriented programming techniques.

A software design environment based on our framework would consist of an editor, a consistency checker and an interpreter. With such a tool a prototype of a system may be derived directly from a high level system specification.

References

[Chang and Lee (73)]
Chang C.L., R.C.T. Lee
Symbolic Logic and Mechanical Theorem Proving
Academic Press, 1973.

[Dietz and van Hee (86)]
Dietz J.G., K.M. van Hee
A Framework for the Conceptual Modeling of Discrete Dynamic Systems
Proc. of Temporal Aspects in Information Systems, 1987.

[Gane and Sarson (77)]
Gane Ch., T. Sarson
Structured Systems Analysis: Tools & Techniques
Improved System Technologies Inc., 1977.

[Harel (86)]
Harel D.
Statecharts: A Visual Approach to Complex Systems
CS 86-02, The Weizmann Institute of Science, 1986.

[Jackson (83)]
Jackson M.
System Development
Prentice Hall, 1983.

[Lloyd (84)]
Lloyd J.W.
Foundations of Logic Programming
Springer Verlag, 1984.

[Lundeberg, Goldkuhl and Nilsson (79)]
Lundeberg M., G. Goldkuhl, A. Nilsson
A Systematic Approach to Information Systems Development
Information Systems, Vol. 4, 1979.

[Reiter (84)] Reiter R.
Towards a Logical Reconstruction of Relational Databases
Brodie M.L., J. Mylopoulos, J.W. Schmidt (eds.)
On Conceptual Modeling
Springer Verlag, 1984.

[Shridhar and Hoare (85)] Sridhar K.T., C.A.R. Hoare
Oxford University Computing Laboratory notes, 1985.

[Ward and Mellor (85)] Ward P.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 responders 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 a 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 A. Lapinski	OR and AI approaches to decision support systems
87/13	J. 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	G. 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