A formal model for system specification

A Formal Model for System Specification

by

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

88/08

April 1988

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB Eindhoven
The Netherlands
All rights reserved
editor: F.A.J. van Neerven

A Formal Model for System Specification

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

Eindhoven University of Technology

ABSTRACT

In this paper we present a framework for modeling and specifying systems, in particular information systems. The framework consists of a formal model for distributed systems and a language for specifying the components of the model. The language consists of an imperative part for specifying state components and their transitions and a functional part where functions and datatypes are given. It may be used to describe non-first-normal-form data structures and to specify recursive queries. It also resembles (after adding some syntactic sugar) the conventional mathematical notations; it is related to the languages of the Z and VDM methods [BJ82,HA87a]. However, in Z and VDM specifications are descriptive, whereas ours are constructive. Therefore our specifications can be executed; this is a very attractive way for potential users (lacking background and training to read formal specifications) to validate a specification.

In section 1 we give an overview to the framework and its usage. In section 2 we describe the model formally and in section 3 we introduce the language. Finally in section 4 we present an example of a distributed database system specified by our method.

1. Introduction

Database design is the specification of a system that is able to store and retrieve information from its environment. We might model this system as an abstract machine. This machine has at each moment an element of a set (called database universe) as its state and can be activated by its environment. An activation consists of a piece of data; in reaction the machine performs a state transition and/or sends data to its environment.

Database design usually is seen as just the specification of the state space or database universe. Within the framework of a data model, one specifies a large set; then this set is restricted by adding constraints. Although this is an essential step, the (often more complex) transition mechanism has to be considered too.

At a more detailed level, a system can be described as a set of processors and data stores. A processor can be activated by the environment or by another processor (even self-activation is possible) and stores can be accessible by a single processor or shared by several processors. This seems to be the natural way to describe e.g. distributed database systems. For the specification of a system as a network of stores and processors, there exist methods like SADT [RO77], ISAC [LU79], and Yourdon [WA85]. These methods have no rigorous semantics and therefore are not suitable for formal specification. Another problem is that their data modeling capacities are weak.

The dynamics of a system is specified in a process model. There are two kinds of dynamics: data flow and control flow. The methods mentioned above stress data flow, whereas control flow is often modeled by Petri nets [PE81] or finite state machines [HA87]. In [HO88], these methods have been combined. Our definition of control flow is the transport of data from one processor to another, whereby the latter is triggered. Data flow means transport of data between a processor and a store.

Our framework integrates three fundamental aspects of system specification: data structuring, data flow and control flow. It can be applied to (monolithic or distributed) database systems, but also to other fields. We call systems that fit into our framework *Distributed Event Systems* (DES). The term *event* is used to describe the triggering of a state transition. A DES is always a closed system, so a target system and its environment together form a DES. Of course we will not specify all components of the environment. We shall treat this subject later in more detail.

A DES is completely determined by a seven-tuple $<S, IS, OS, C, TC, M, R>$.
Before we explain the meaning of these components we give two diagrams of an example of a DES. It is possible to combine the two diagrams into one.

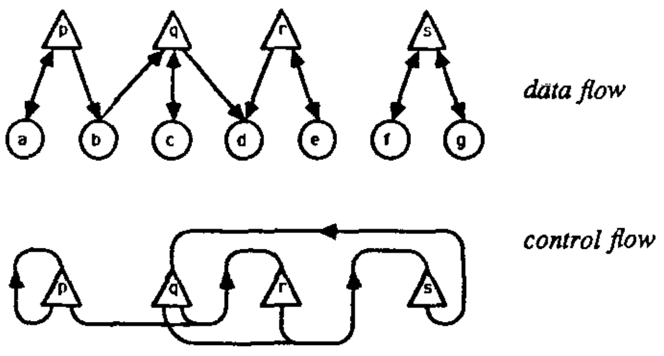

Fig. 1

The triangles p,q,r,s represent processors. Each processor k consists of two functions: M_k and R_k. The circles $a,...,g$ represent stores. They may have simple structures like a calendar date, or complex structures like a database. The connections between processors and stores mean that a processor may access the store. If there is an arrow in the direction of the store then it acts as output for the processor, if an arrow points to the processor it acts as input. Note that there is no direct data flow between two processors. However, it is possible to transmit data from one processor to another as indicated in the second diagram. Each processor has one input channel and it may have several output channels.
For each channel, the structure of the values that may pass through it is determined. Channels may join. Processors have a single input channel; they are triggered by the values arriving through that channel. Note that the type of a channel may allow very complex values. The values passing through a channel are called triggers.

In Fig. 1 we have already met three of the components of the seven-tuple:

- TC is a function that assigns to each processor a set of output channel names (processor names); in the picture for p: p,r, for q: r,s, etc.
- IS is a function that assigns to each processor a set of names of (stored) variables that are used as input variables for that processor; for p: a, for q: b,c, for r: e, etc.
- OS is a function similar to IS, it assigns to each processor a set of names of output variables; for p: a,b, for q: c,d, for r: d,e, etc.

Now we will explain S and C.

- S is a set-valued function, where dom(S) is the set of names of stored variables (stores). S_i is a set that is called the type of the variable with name i.
- C is a set-valued function, where dom(C) is the set of processor names. C_k is called the type of the triggers passing through the input channel of processor k.

Finally we turn to M and R.

- M is a function-valued function, where dom(M) is the set of processor names. For a processor k, M_k is a function with as arguments the store names of IS_k, their values, and a trigger value taken from C_k. The result of M_k is a partial function which assigns to some stores from OS_k a new value.
  M_k is called the *manipulator* of processor k because it may modify the stored variables.
- R is also a function-valued function, where dom(R) is the set of processor names. For a processor k, R_k is a function with the same arguments as M_k. Its result, however, is a partial function that assigns a value to some triggers of the set TC_k.
  R_k is called the *reactor* of processor k because it produces triggers.

We will describe the behavior of a DES in an informal way. For every input channel there is a multiset of triggers. At each moment a processor k having a non-empty multiset of triggers may commit a transition which consists of the following actions:

a. selection of a trigger from the available triggers,

b. simultaneous computation of M_k and R_k with as arguments the values of the input stores and the trigger value.

At the same moment, several processors may commit a transition, however no two processors sharing a stored variable that is an output variable, may commit at the same moment. It is required that each produced trigger value is taken into execution at some moment, so a system must be starvation-free.
We do not specify how processors select triggers from their multiset, nor how they control the exclusive updating of output variables. It is left to the implementation to choose a solution for these problems. It is easy to find a solution by committing transitions for processors sequentially, however it is often desired to exploit parallelism. Since for a DES the selection of triggers to be executed is not specified, it may be considered a non-deterministic system.

To define the behavior of a DES formally, we use a top-down approach, starting with a very simple system structure that evolves by stepwise refinement into a DES. Several concepts and properties are introduced during this evolution process; each one of them at its proper level, so their treatment is not obscured by too much detail. This is done in section 2.

Many systems may be modeled as a DES, for instance typical database systems as in [HE88], but also communication networks or integrated circuits. An important modeling issue is the separation of the (closed) DES into a target system and its environment. We may model the environment as one or more "black box" processors, possibly with stores, whose specification is unknown. An example is given in the following figure.

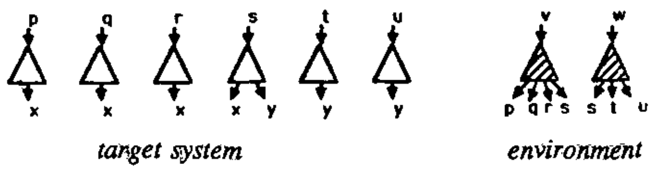

Fig. 2

Black box *x* may trigger processors *p,q,r,s*; black box *y* may trigger *s,t,u*. Each processor triggers its invoker; processor *d* may trigger both black boxes. It is also possible to model that a processor in the environment may access stored variables of the target system.

The language EXSPECT (for EXecutable SPECification Tool) is used to specify the components of a DES. It consists of a functional part and an imperative part. In the imperative part the possible triggers and store values are specified and the state transitions caused by the processors are described. The former is done by associating to each channel and store a certain data type. A data type represents a certain set of objects. The state transitions are specified with the help of functions, applied to the input store values and triggers.
In the functional part, the types and functions from the imperative part are functionally decomposed into simpler types and functions until a basic level is reached. In section 3 we shall present the language with some examples in greater detail.

Our framework is supported by software tools for editing, (type) checking and interpreting specifications. The type checker tests a description for type consistency. The interpreter simulates the behavior of a described system. This last facility is essential for validation purposes; for non-experts it is difficult to understand a formal specification, whereas a prototype is easily understood.
An important difference between an executable specification and a real implementation is that the specificator is only concerned with the functionality of the system and not with matters like performance, system load, reliability, etc. Therefore a specification language may use more powerful constructs than a programming language, sacrificing execution speed for the sake of clarity and ease of use.
The use of formal and executable specification methods has influences on the life cycle of a system. Its specification phase is lengthened somewhat (it remains much shorter than the implementation phase). In return the system validation by the potential users takes place at an early stage, modifications are less costly and finally the implementation phase is shortened. In the specification phase we may distinguish the following activities (not necessarily in the given order).

- Identify the stores and processors in the target system and identify the "black boxes" in the environment (data flow analysis).
- Identify the channel structure (control flow analysis).
- Define a data type for the stored values in each store (data modeling).
- Define constraints on the store types (database constraints).
- Define a type for the triggers passing through each channel.
- Define the manipulator and reactor functions for each processor.
- Verify that the constraints are kept invariant.

2. Theoretical framework

In this section we define a framework for the formal description of a class of systems. A framework consists of related mathematical objects like sets and functions, each describing some characteristics of the systems we consider. We say that a system fits into our framework if there is a one-to-one mapping between the objects in the system and the mathematical objects in the framework.

We develop our framework top-down, which means that we define a sequence of frameworks for systems starting from a very general one. Each successor framework of a framework defines a subclass of systems, hence in each successor framework more details are specified. After each specialization step, the specification task becomes better structured and less complicated.

In principle this process of specialization can be continued endlessly. Here our goal is to define the framework called distributed event systems (DES). Many systems occurring in the real world fit into this framework.

We start with some notations and conventions.

- The symbol $\mathbb{P}$ is used to denote the power set.
- For X a set of sets, $\cup X$ is the union of the elements of X.
- For a set A, $\mathbb{B}(A)$ denotes the set of multisets (bags) over A, isomorphic to $A \rightarrow \mathbb{N}$. For $x \in \mathbb{B}(A)$ and $a \in A$,

  $$a \in x \Leftrightarrow x(a) > 0.$$

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

  $$s \subset x \Leftrightarrow \forall a \in s : a \in x,$$

  $$x \setminus s = \lambda a \in A : \text{if } a \in s \wedge a \in x \text{ then } x(a)-1 \text{ else } x(a) \text{ fi},$$

  $$x \cup s = \lambda a \in A : \text{if } a \in s \text{ then } x(a)+1 \text{ else } x(a) \text{ fi}.$$

  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.
- For a set-valued function F, ΠF denotes the set of all functions f over dom(F), and $\Pi^* F$ the set of all partial functions f over dom(F), both with

  $$\forall x \in \text{dom}(f) : f(x) \in F(x).$$

- We denote function restriction by $\upharpoonright$.

In the reminder of this section we omit proofs. A full-scale discussion of our model with proofs added can be found in [HE88a]. We start with the most general concept.

Definition 1 (basic system (BS))

A *basic system* is a pair $<\Sigma, T>$, where
- Σ is a set, called the basic state space,
- T is a binary relation over Σ, i.e. $T \subset \Sigma \times \Sigma$, and T is called the transition relation.

Definition 2 (process)

Let $<\Sigma,T>$ be a BS. Then

$$\Sigma^{\infty} = \{p \in \mathbb{N} \to \Sigma \mid p_0 \in \Sigma \wedge \forall n \in \mathbb{N}: <p_n, p_{n+1}> \in T\}$$

is called the *process space* and an element of Σ^{∞} is called a *process path*.
Let $\mathbb{N}_m = \{n \in \mathbb{N} \mid n \leq m\}$. Then

$$\Sigma^{m} = \{p \in \mathbb{N}_m \to \Sigma \mid p_0 \in \Sigma \wedge \forall n \in \mathbb{N}_{m-1}: <p_n, p_{n+1}> \in T\}$$

and

$$\Sigma^{*} = \cup \{\Sigma^{m} \mid m \in \mathbb{N}\}.$$

An element of Σ^* is called a trace. One can easily prove that Σ^* is a prefix-closed trace structure.
We proceed now with the first specialization.

Definition 3 (event system (ES))

An *event system* is a four-tuple $<S,E,M,R>$, where

- S is a set, called the state space,
- E is a set of sets, called the (conflict-free) event set space, $\{\} \notin E$; the event type $\mathbb{E}$ is defined by $\mathbb{E} = \cup E$,
- M is a function, called the manipulator with $M \in S \times E \to S$,
- R is a function, called the reactor, with $R \in S \times E \to \mathbb{B}(\mathbb{E})$,

Lemma 1

An event system $<S,E,M,R>$ specifies a basic system $<\Sigma,T>$, where Σ and T are given by

- $\Sigma = S \times \mathbb{B}(\mathbb{E})$,
- $T = \{<<s,b>,<s',b'>> \mid \exists e \in E: e \subset b \wedge s' = M(s,e) \wedge b' = b \setminus e \cup R(s,e)\}$.

We give an operational view of an ES. Let it start in $s \in S$ with initial event bag b.
After some time a transition is performed, this means that a conflict-free event set $e \subset b$ is selected and that the new state is transformed into $M(s,e)$ and that some new events are created according to $R(s,e)$. These new events are added to the bag of events. The moment of transition must be considered as the moment at which a transformation is committed. The system may have worked on this transformation for some time and it may be working on some others too, that will be committed at a later stage.

A natural requirement for an event system is that each event in an actual event bag will be taken into execution at some moment. This means that we want to exclude the possibility of starvation. However, since we are not able to distinguish all individual events in a bag, we introduce the notion of observable starvation.

Definition 4 (observable starvation)

Let p be a process path of an ES and let p_n be denoted by $<s_n, b_n>$ for $n \in \mathbb{N}$.
Then p has *observable starvation* iff

$$\exists n \in \mathbb{N}: \exists x \in b_n: \forall m \in \mathbb{N}: m \geq n \Rightarrow$$
$$\forall e \in E: e \subset b_m \wedge s_{m+1} = M(s_m,e) \wedge b_{m+1} = b_m \setminus e \cup R(s_m,e) \Rightarrow x \notin e.$$

Note that we can have a non-decreasing number of x's in the bag on starvation-free paths if, for example, for $x \in \mathbb{E}$ it holds that

$$\forall s \in S : \forall e \in E : x \in e \Rightarrow x \in R(s,e).$$

We say that a system has deadlock if at some stage it still has events but no transition is possible.

Definition 5 (deadlock)

An ES $<S,E,M,R>$ has *deadlock* iff there exists a non-empty event set $e \subset \mathbb{E}$ such that

$$\forall e' \subset e : e' \notin E.$$

Note that almost every ES can have paths with starvation. This is because the choice of $e \subset b$ in Lemma 1 is left free. We can introduce a *selection function* $\sigma \in S \times \mathbb{B}(\mathbb{E}) \rightarrow E$ to choose triggers from the event bag. Such a function determines uniquely a path for an ES, given an initial state s_0 and event bag b_0. It is possible to define a class of event systems for which a selection function σ exists, such that the paths determined by σ are starvation-free. Every deadlock-free ES can be simulated by an ES from this class.

It is natural for a lot of systems to require that individual events always may cause a transition. A transition triggered by a set of events can be equal in effect to a sequence of transitions caused by the individual events from that set. This property is called serializability; if the effect does not depend upon the ordering within the sequence it is called strong serializability.

Definition 6 (serializability)

Let $<S,E,M,R>$ be an ES. It is called *serializable* if

$$\forall e \in E : \forall e' \subset e : e' \in E$$

and for all $s \in S$ and $e \in E$ it holds that there exists an $x \in e$, such that

$$M(s,e) = M(M(s,\{x\}), e \backslash \{x\})$$
$$R(s,e) = R(s,\{x\}) \cup R(M(s,\{x\}), e \backslash \{x\}).$$

It is called *strongly serializable* if the above holds for all $x \in e$.

Now we proceed to another framework, the distributed event systems. This framework describes systems that are composed of three entity types, processors, stores, and channels. A processor consists of a manipulator that may change the contents of stores and a reactor that may create new events for processors. A store is in fact a state variable. It has a type and it may be changed by one or more processors. We don't allow multiple processors to use a store at the same time if at least one of them may change it. We express this requirement by the specification of conflict-free event sets.
The processors are connected by channels. Several processors may transmit events through the same channel simultaneously. Each processor has only one input channel. An event put into a channel will arrive at the processor for which this channel is the input channel. At each transition of the induced basic system every processor is triggered by at most one event and for each output channel at most one event is produced.

Definition 7 (distributed event system (DES))

A *distributed event system* is a seven-tuple $<S,IS,OS,C,TC,M,R>$, where

- S is a set valued function, where $\mathrm{dom}(S)$ is called the set of store names,
- M is a function-valued function, where $\mathrm{dom}(M)$ is called the set of processor names; for $k \in \mathrm{dom}(M)$, M_k is called the manipulator of processor k,
- R is a function-valued function, $\mathrm{dom}(R) = \mathrm{dom}(M)$; R_k is called the reactor of processor k,
- IS is a set-valued function, $\mathrm{dom}(IS) = \mathrm{dom}(M)$, $IS_k \subset \mathrm{dom}(S)$; IS_k is called the input store set of processor k,
- OS is a set-valued function, $\mathrm{dom}(OS) = \mathrm{dom}(M)$, $OS_k \subset \mathrm{dom}(S)$; OS_k is called the output store set of processor k,
- C is a set-valued function, $\mathrm{dom}(C) = \mathrm{dom}(M)$; C_k is called the input channel type of processor k,
- TC is a set-valued function, $\mathrm{dom}(TC) = \mathrm{dom}(M)$, $TC_k \subset \mathrm{dom}(M)$; TC_k is called the output channel set of processor k,

such that

$$\forall k \in \mathrm{dom}(M)\colon M_k \in \Pi(S \restriction IS_k) \times C_k \rightarrow \Pi^*(S \restriction OS_k),$$
$$\forall k \in \mathrm{dom}(M)\colon R_k \in \Pi(S \restriction IS_k) \times C_k \rightarrow \Pi^*(\lambda l \in TC_k : C_l).$$

Lemma 2

A distributed event system $<S,IS,OS,C,TC,M,R>$ specifies an event system $<\bar{S},\bar{E},\bar{M},\bar{R}>$, where $\bar{S},\bar{E},\bar{M}$, and $\bar{R}$ are given by

- $\bar{S} = \Pi S$,
- $\bar{E} = \{e \in \Pi^*(\lambda k \in \mathrm{dom}(M)\colon C_k) \mid \forall k,l \in \mathrm{dom}(e)\colon k \neq l \Rightarrow IS_k \cap OS_l = OS_k \cap OS_l = \{\}\}$,
- for all $s \in \bar{S}$ and $e \in \bar{E}$

$$\bar{M}(s,e) = (\underset{k \in \mathrm{dom}(e)}{\circ} \tilde{M}_{k,e})(s)$$

where for $k \in \mathrm{dom}(e)$,

$$\tilde{M}_{k,e}(s) = \lambda i \in \mathrm{dom}(S)\colon \text{if } i \in \mathrm{dom}(M_k(s \restriction IS_k, e_k)) \text{ then } M_k(s \restriction IS_k, e_k)_i \text{ else } s_i \text{ fi}$$

and

$$\bar{R}(s,e) = \lambda <k,x> \in \cup \bar{E}\colon (\#l \in \mathrm{dom}(M)\colon R_l(s \restriction IS_l, e_l)_k = x).$$

In the following, the specification of an event system $<\bar{S},\bar{E},\bar{M},\bar{R}>$ by a distributed event system $<S,IS,OS,C,TC,M,R>$ will always be as above.

Note that in a DES each processor gets single events as opposed to an ES where a manipulator and a reactor get event sets as their input. Operationally speaking we have split the manipulator and the reactor from the ES that we are specifying, into a number of processors. Each processor consists of a manipulator and a reactor, that both operate with single events as their input. We have also split the state into several stores each representing a part of the state. The state is partitioned to be able to specify that processors only need a projection of the total state in order to determine their manipulation and reaction.

We have to verify that the functions $\tilde{M}_{k,e}$, introduced in Lemma 2, commute in order to justify the use of the compound composition °.

Lemma 3

Let $<S,IS,OS,C,TC,M,R>$ be a DES that specifies the ES $<\overline{S},\overline{E},\overline{M},\overline{R}>$. Then:

$$\forall s\in\overline{S}\ \forall e\in\overline{E}\ \forall k,l\in \mathrm{dom}(M)\text{: } k\neq l \Rightarrow \tilde{M}_{k,e}(\tilde{M}_{l,e}(s)) = \tilde{M}_{l,e}(\tilde{M}_{k,e}(s)).$$

We can prove that the ES $<\overline{S},\overline{E},\overline{M},\overline{R}>$ that is specified by the DES $<S,IS,OS,C,TC,M,R>$ is strongly serializable.

3. The language

3.1 Introduction

The language EXSPECT (from EXecutable SPECification Tool) is designed for the formal specification of information systems. Since it is executable, EXSPECT can be used for prototyping. In this section we shall show some of its features; afterward we shall more formally present its syntax and semantics.

In the functional part of EXSPECT we have types, objects, and functions. Types correspond to sets of objects and functions have types (or sets of types) as domain. These types, objects, and functions can be defined with the help of a small set of basic types, objects, and functions. In the imperative part, types are used to define stores and channels, objects and functions are used to define the processors. We start with type definitions, then proceed to store and channel definitions.
A typographical note: in this section pieces of EXSPECT text and "placeholders" for it are in roman font; other objects are in italics.

EXSPECT is a typed language. Datatypes (types for short) correspond to sets of objects. A type can be constructed from the basic types "bool", "str", and "num" (corresponding respectively to the booleans, strings and rational numbers) and the type constructors $\times$, \$, and $\rightarrow$. If T and U are types, corresponding to sets $\theta(T)$ and $\theta(U)$ respectively, then \$T corresponds to the set of finite subsets of $\theta(T)$, $T\times U$ to the set of pairs $\ll t,u \gg$ with t in $\theta(T)$ and u in $\theta(U)$, and $T\rightarrow U$ to the set of mappings (finite functions) from $\theta(T)$ to $\theta(U)$. The order in which type constructors are applied must be indicated by brackets; when absent, \$ takes precedence over $\times$ and $\rightarrow$.
An EXSPECT program can contain type definitions, where names are given to type expressions. These names then can be used to construct new types. An example is formed by the following set of type definitions,

> **type** anr **from** num;
> **type** sender **from** str;
> **type** delivery **from** anr×sender.

Stored variables (or stores) and trigger channels can be declared with their type, for example

> **channel** dc: delivery;
> **channel** ac: anr;
> **store** ds: $delivery.

We can construct expressions, starting from a set of constants and a few basic functions. Expressions denote objects; for instance the expression

> ins(3, ins(div(1,2), {}))

denotes the set consisting of the two rational numbers ½ and 3. Many expressions denote the same object. We identify an object and the equivalence class of expressions denoting this object and choose a standard expression from each equivalence class to represent it. A process called evaluation transforms an expression to the standard expression in its class.
We can build expressions with constants (in the example above the numbers 1,2,3 and the empty set {}) and function applications: the function name followed by a list of arguments between brackets (in the above example we met the functions "ins", insertion of an element into a set and "div", division of two rationals). A third way is by defining mappings; the example

$[x{:}\, S \mid E_x]$

denotes the mapping with finite domain S (S must be an expression denoting a finite set) in which each element x is mapped to E_x (E_x is an expression which may contain x as a dummy parameter). Mappings can serve as arguments of standard functions like "rng", giving its range, like in

rng($[x{:}\, S \mid E_x]$).

Expressions can be given names and these names can be used to construct new expressions. In the evaluation of such a new expression, the defining expression is substituted for the name. More interesting still is giving names to parametrized expressions or functions. A function can contain parameters (dummy identifiers); the parameter and its type must be given with the name, for example

artnr [d:delivery] := π_1(d).

This defines the function "artnr" with as domain the type "delivery". The basic function π_1 takes the first constituent from a pair.
For each function (basic or defined) the possible types of the parameters and the type of the result must be known. This is the reason why the type "delivery" had to be attached to the dummy "d" in the above definition. For polymorphic functions like π_1 this looks hard, since pairs of all possible types can serve as parameter. We can still describe the type behavior of such functions by using *type variables*. Type variables and type expressions containing them do not correspond to sets, but to functions from substitutions to sets, a substitution being a mapping of type variables to type expressions not containing variables. The type behavior of π_1 is declared (using type variables T and S) as follows,

π_1[x:T×S]: T.

From this information, a piece of software called type checker can derive (not surprisingly) that given a delivery d, the type of artnr(d) is anr. We can also define our own polymorphic functions, for example

π_{11}[x: (T×S)×U] := $\pi_1(\pi_1(x))$.

The type checker detects and reports typing errors; for instance the following definitions contain typing errors,

f [x:T] := π_1(x);
p [x:$delivery] := artnr(x).

The type system thus helps in data structuring and error detection.

Function definitions may be recursive, i.e. the definition may contain applications of the same function. For example with help of the functions "cond", "lt", and "sub" (performing respectively if-then-else selection, less-than-test, and numeric subtraction) we can define the fractional part of a number as follows,

```
frc [x:num] := cond(lt(x,0),
                    sub(0,frc(sub(0,x))),
                    cond(lt(x,1),x,frc(sub(x,1)))) : num.
```

In recursive definitions, the type of the result must be added. This can be done in other definitions too; the type checker compares the result type of the expression with the added type.

Expressions like the one directly above may have a nice and simple syntax, they are not very nice-looking to the human eye. Therefore a certain "sugaring" is applied, representing many basic functions in infix notation. In the sugared version the above expression reads

```
if x<0
then –frc(–x)
else if x<1 then x else frc(x–1) fi fi.
```

We shall present our examples from now on in the sugared version. However, because of its lengthy and tedious nature, we do not present a syntax and semantics for the sugared version of EXSPECT.

The expressions we have treated so far denoted objects. The situation becomes more complicated when we allow stores and channels in them. For instance the expression

```
ins(dc, ds)
```

denotes a set of deliveries that depends on the values of ds and dc. We call these expressions *state-dependent*. These expressions cannot be used in function definitions, but are used for defining processors. A processor consists of heading and a body. In the processor heading we mention the processor name, the stores inspected (if any), the stores updated (if any), the input channel, and the output channels (if any).

The processor body consists of (conditional) assignments to output stores and channels. As an example we define a processor triggered by dc that inserts its trigger in store ds and, if the article number in dc was already in ds as article number of some delivery, emits it through channel ac.

```
proc p[tin dc, tout ac, sin ds, sout ds] := ds ← ins(dc,ds),
                                            if artnr(dc) ∈ rng([x: ds | artnr(x)])
                                            then ac ⇐ artnr(dc) fi
```

Note that the set ds in the expression rng([x: ds | artnr(x)]) has not been modified by the assignment directly above it; assignments to stores and channels are effectuated only after all the expressions in the processor body have been evaluated.

3.2 Syntax and semantics

We describe the syntax in a self-explanatory BNF-like manner. Symbols between square brackets "[]" are optional. Terminals are in boldface. The non-terminal "nat" denotes a natural number in decimal notation. The non-terminals "typvar", "var" and all non-terminals ending in "name" denote identifiers. The non-terminal "nqstr" denotes any string not containing quotes. An EXSPECT program is called script.

```
script      := net netname ; lines ;
lines       := line | line ; lines
line        := typedef | stdecl | chdecl | exdef | fundef | procdef
typedef     := type  typname from typex
typex       := typcon | typvar | typname | ( typex ) | $ typex | typex × typex | typex → typex
typcon      := bool | num | str
stdecl      := store stname : typex
chdecl      := channel chname : typex
exdef       := exname := expr [: typex]
expr        := con | var | stname | chname | exname | funappl | mapping
con         := nat | ′ nqstr ′ | quote | true | false | {}
funappl     := fname ( arglist )
arglist     := expr [, arglist]
mapping     := [ var : expr | expr ]
```

```
fundef       := fname [ parlist ] := expr [: typex]
parlist      := var : typex [, parlist]
procdef      := proc pname [ iodef ] := statlist
iodef        := tin chname [, tout chlist] [, sin stlist] [, sout stlist]
chlist       := chname [, chlist]
stlist       := stname [, stlist]
statlist     := statement [, statlist]
statement    := condstat | assignment
condstat     := if expr then statlist else statlist fi
assignment   := chname ⇐ expr | stname ← expr
```

The above syntax must obey the following context rules.

- A type expression ("typex") in a type definition ("typdef"), store or channel declaration ("stdecl", "chdecl"), and expression definition ("exdef") may contain no type variables ("typvar").
- A type expression in the right-hand side (after the ":=" symbol) of a function definition ("fundef") may not contain other type variables than those in the type expressions of its parameter list ("parlist").
- An expression ("expr") in an expression definition, argument list ("arglist"), mapping or function definition may contain no store or channel names ("stname", "chname").
- An expression in an assignment inside a processor definition ("procdef") may contain only the channel name mentioned after "**tin**" in the I/O definition ("iodef") of the same processor definition; it may contain only store names mentioned after "**sin**" in the same I/O definition.
- An expression in an expression definition or assignment may not contain any free variables. We denote the set of free variables of an expression E by $\mathbb{F}$. This is defined as follows.
  $\mathbb{F}(E) = \{\}$, if E is a constant ("con"), store or channel name, or expression name ("exname").
  $\mathbb{F}(E) = \{x\}$, if E is a variable ("var") of the form x.
  $\mathbb{F}(E) = \mathbb{F}(E_1) \cup \ldots \cup \mathbb{F}(E_n)$, if E is a function application ("funappl") of the form $f(E_1,\ldots,E_n)$.
  $\mathbb{F}(E) = \mathbb{F}(E_1) \cup (\mathbb{F}(E_2) \backslash \{x\})$, if E is a mapping of the form $[x{:}\,E_1 \mid E_2]$.
  Note that the above definition of $\mathbb{F}$ is recursive; since the length the expression diminishes in each recursion step, we can prove by induction that this recursion is finite. A similar argument holds for many recursive (syntax-driven) definitions to come. The possibility of an infinite recursion is explicitly mentioned in each case.
- An expression in a function definition may contain no free variables other than those in its parameter list.
- For each type name ("typname") in a type expression or expression name in an expression there must be one and only one type definition ("typdef") or expression definition where the same name is in the left-hand side (of the "from" or ":=" sign).
- Each function name ("fname") in an expression must be the name of a basic function or there must be a function definition with the same name in the left-hand side (of the ":=" sign). The names of the basic functions are cond, eq, lt (involving booleans), sub, div (involving numerals), head, tail, cat (involving strings), pick, ins, del (involving sets), and π_1, π_2, prod (involving pairs).
- Each channel or store name in an I/O definition must correspond to a channel or store declaration with the same name.

- Each channel or store name in the left-hand side (of the "$\leftarrow$" or "$\Leftarrow$" symbol) of an assignment inside a processor definition must be mentioned after "**tout**", respectively "**sout**" in the corresponding I/O definition.
- The identifiers **net**, **type**, **from**, **store**, **channel**, **proc**, **tin**, **tout**, **sin**, **sout**, **if**, **then**, **else**, **fi**, bool, num, str, quote, true, and false are reserved, as are the names of the basic functions.
- The list of names in the left-hand sides of type definitions, store or channel declarations, expression or function definitions and processor definitions and the netname may not contain duplicates.
- There is a 1-1 correspondence m between channel and processor names. This correspondence is given by adding to each processor the channel name after "**tin**" in the corresponding I/O definition.
- In a statement list ("statlist"), no assignments may occur to the same store or channel. Formally, the statement list S must satisfy $\mathcal{C}(S)$ = true, where $\mathcal{C}$ is defined as follows.

  $\mathcal{C}(S)$ = true, if S is an assignment.
  $\mathcal{C}(S) = \mathcal{C}(S_1)$, if S is of the form "**if** E **then** S_1 **fi**".
  $\mathcal{C}(S) = \mathcal{C}(S_1) \wedge \mathcal{C}(S_2)$, if S is of the form "**if** E **then** S_1 **else** S_2 **fi**".
  $\mathcal{C}(S) = \mathcal{C}(S_1) \wedge \mathcal{C}(S_2) \wedge (\mathbb{N}(S_1) \cap \mathbb{N}(S_2) = \{\})$, if S is of the form "$S_1$, S_2".

  Here the set of channels and stores $\mathbb{N}(S)$ of a statement list S is defined as follows.

  $\mathbb{N}(S) = \{a\}$, if S is an assignment of the form "$a \leftarrow E$" or "$a \Leftarrow E$".
  $\mathbb{N}(S) = \mathbb{N}(S_1)$, if S is of the form "**if** E **then** S_1 **fi**".
  $\mathbb{N}(S) = \mathbb{N}(S_1) \cup \mathbb{N}(S_2)$, if S is of the form "**if** E **then** S_1 **else** S_2 **fi**" or "S_1, S_2".

The rules for type correctness of an EXSPECT script are elaborated in [HE88b]. The EXSPECT type system allows for hierarchy: a type can be a subtype of another type. We have already seen polymorphy in action when introducing type variables. These two concepts together make the typing rules rather complicated; here we give some vague rules based on intuition.
An expression is correctly typed if its type can be computed. The type of constants is mostly obvious; only the empty set {} has the somewhat mysterious type "\$void", which acts as a supertype of all "set" types.
The type of a channel or store name is given in the corresponding declaration. The type of a variable is context-dependent.

An expression name has the same type as the expression at the right-hand side of the corresponding expression definition if this expression definition is not typed. If the expression definition is typed i.e. of the form d := E: T, then its type is T, provided T is (a sub- or supertype of) the type of E.

In a function application $f(E_1,...,E_n)$, the expressions $E_1,...,E_n$ must be correctly typed. If the function is defined, the corresponding function definition must look like

$$f[x_1{:}T_1, \ldots, x_n{:}T_n] := E$$

or

$$f[x_1{:}T_1, \ldots, x_n{:}Tn] := E: T.$$

The types $t_1,...,t_n$ of $E_1,...,E_n$ must "fit" $T_1,...,T_n$, which means that a simultaneous substitution of the type variables in $T_1,...,T_n$ giving types $S_1,...,S_n$ can be found such that $t_1,...,t_n$ are (subtypes of) $S_1,...,S_n$. The type of $f(E_1,...,E_n)$ then is found in the first case by adding the variables $x_1,...,x_n$ to the context with types $S_1,...,S_n$ and then computing the type of E.
In the second case, the type of $f(E_1,...,E_n)$ is found by performing the same substitution of type variables

to T. If there are more substitutions possible, the one giving the "strongest" type for $f(E_1,...,E_n)$ is selected.
In the second case it is also checked that the type of E (with the modified context) is (a sub- or supertype of) the type of $f(E_1,...,E_n)$.

If the function is basic, an analogous fitting procedure is applied. For instance $\pi_1(E)$ is correctly typed if the type of E is $T \times S$ for some types T and S. The type of $\pi_1(E)$ is then T.

In a mapping $[x{:}A \mid E]$, the type of A must be (subtype of) \$T for some T. We then add the variable x with type T to the context and compute the type of E, giving, say, S. The type of the mapping then is $T \rightarrow S$.

In a conditional statement **if** E **then** S_1 [**else** S_2] **fi**, the type of the expression E must be bool.

In an assignment $v \leftarrow E$ or $v \Leftarrow E$, the type of the expression E must be (subtype of) the type of the channel or store name v.

3.3 Semantics of the functional part

The functional part of EXSPECT consists of type, expression and function definitions. We can describe its semantics in terms of data objects as defined below. We restrict ourselves here to the semantics of type expressions without type variables and expressions without variables in a semi-formal way. For a formal and complete treaty of the semantics of EXSPECT, we refer to [HE88b].

Definition 8

The set DO of data objects satisfies the following properties.

1. Truth, falsehood, the rational numbers, and strings are elements of DO.
2. If $x_1,...,x_n$ are elements of DO, then the set $\{x_1,...,x_n\}$ is an element of DO.
3. If x_1 and x_2 are elements of DO, then the pair $\ll x_1,x_2 \gg$ is an element of DO.

The semantics of a type expression without variables is given by a partial function θ from such type expressions to subsets of DO. The domain of θ is the set of correct types. We have that θ(bool) equals the set formed by truth and falsehood; θ(num) the set of rational numbers, and θ(str) the set of strings. For types A and B in dom(θ), $\theta(\$A)$ is the set of finite subsets of $\theta(A)$, $\theta(A\times B)$ is the set of pairs $\ll a,b \gg$ where $a \in \theta(A)$ and $b \in \theta(B)$, and $\theta(A \rightarrow B)$ is that subset of $\theta(\$(A\times B))$ such that for each f in this subset and any two elements $\ll a_1,b_1 \gg$ and $\ll a_2,b_2 \gg$ of f it must hold that $a_1=a_2$ implies $b_1=b_2$.
For a type name a, $\theta(a)$ is some subset of $\theta(p(a))$, where p(a) is the type expression on the right-hand side of the unique type definition with type name a, provided $\theta(p(a))$ exists. If a recursion occurs, i.e. in this substitution at some stage the type name a reappears, $\theta(a)$ does not exist.

The semantics of an expression E without free variables and channel or store names is described by a partial function δ from such expressions to data objects. The semantics of constants is simple and intuitively obvious. For instance, δ('help') equals the string "help". For an expression name e, $\delta(e) = \delta(d(e))$, where d(e) is the expression on the right-hand side of the unique expression definition with expression name e. If a recursion occurs $\delta(e)$ does not exist.
If E is a function application $f(E_1,...,E_n)$ and f is basic, $\delta(E)$ can easily be given. For example, $\delta(\pi_1(E_1))=a$ if the data object $\delta(E_1)$ is a pair of the form $\ll a,b \gg$; if $\delta(E_1)$ does not exist or is not a pair, $\delta(\pi_1(E_1))$ does not exist. If f is not basic, there exists a unique function definition of the form (omitting types)

$f[x_1, \ldots, x_n] := E'$.

Denoting the substitution in E′ of $a_1,...,a_n$ for $x_1,...,x_n$ respectively by $E'[x_1 \Rightarrow a_1,...,x_n \Rightarrow a_n]$, we set $\delta(E) = \delta(E'[x_1 \Rightarrow E_1,...,x_n \Rightarrow E_n])$, provided the latter exists. Here (finite) recursion is possible; infinite recursion implies that $\delta(E)$ does not exist.
If E is a mapping of the form $[x: A \mid E']$, then $\delta(A)$ must be a finite set, say $\{a_1,...,a_n\}$. Then $\delta(E)$ is the set of pairs $\{\ll a_1,e_1 \gg,...,\ll a_n,e_n \gg\}$, where for each i in $\{1,...,n\}$, $e_i = \delta(E'[x \Rightarrow a_i])$.

3.4 Semantics of the imperative part

The imperative part of EXSPECT consists of store and channel declarations and processor definitions. We shall define their semantics in terms of Discrete Event Systems. Informally, the semantics is such that the store definitions and channel definitions specify the S- and C-components of a DES $<S,IS,OS,C,TC,M,R>$. The I/O list of the processor definitions specify the IS-, OS-, and TC-components, whereas the M- and R-components are specified by their statement lists. We will now give the formal semantics of such a network, i.e. we specify how a DES $<S,IS,OS,C,TC,M,R>$ is specified by such a network definition. We assume that the script is correct, i.e. that the types and expressions below lie in the domains of θ and δ respectively.

For every store definition "**store** s: T", it holds that

$s \in \mathrm{dom}(S)$,

$S(s) = \theta(T)$.

For every channel definition "**channel** c: T", it holds that

$m(c) \in \mathrm{dom}(C)$,

$C(c) = \theta(T)$.

Because of the context conditions, $\mathrm{dom}(C) = \mathrm{dom}(M)$.
For every processor definition "**proc** p[io] := k", it holds that

$p \in \mathrm{dom}(M)$.

Consider the above processor definition. Let *is* be the (possibly empty) set of storenames occurring in the storelist ("stlist") after **sin** in *io*. Let *os* and *ot* be defined analogously for **sout** and **tout** respectively. Then

$$IS(p) = is,$$
$$OS(p) = os,$$
$$TC(p) = ot.$$

This leaves us to define $M(p)$ and $R(p)$ for every p in $\mathrm{dom}(M)$. Suppose $p \in \mathrm{dom}(M)$ given. Let s be a function assigning to each store name $r \in IS(p)$ a data object in $S(r)$ and let c be a data object in $C(p)$. Then for an expression E containing store names $r_1,...,r_n$ in $IS(p)$ and channel name q with $m(q) = p$ we define

$$\Phi(s,c,E) = \delta(E[r_1 \Rightarrow s(r_1),...,r_n \Rightarrow s(r_n), q \Rightarrow c]).$$

Let k be a statement list in which, for each expression E, $\Phi(s,c,E)$ is defined. Then we define $\mathbb{M}(s,c,k)$ and $\mathbb{R}(s,c,k)$ as follows.

- If k is an assignment of the form "r $\leftarrow$ E", then $\mathbb{M}(s,c,\mathrm{k}) = \{\ll \mathrm{r},\Phi(s,c,\mathrm{E})\gg\}$; $\mathbb{R}(s,c,\mathrm{k}) = \{\}$.
- If k is an assignment of the form "q $\Leftarrow$ E", then $\mathbb{M}(s,c,\mathrm{k}) = \{\}$; $\mathbb{R}(s,c,\mathrm{k}) = \{\ll \mathrm{q},\Phi(s,c,\mathrm{E})\gg\}$.
- If k has the form "**if** E **then** k_1 **fi**" and $\Phi(s,c,\mathrm{E}) = \delta(\mathrm{true})$, then $\mathbb{M}(s,c,\mathrm{k}) = \mathbb{M}(s,c,\mathrm{k}_1)$; $\mathbb{R}(s,c,\mathrm{k}) = \mathbb{R}(s,c,\mathrm{k}_1)$.
- If k has the form "**if** E **then** k_1 **fi**" and $\Phi(s,c,\mathrm{E}) = \delta(\mathrm{false})$, then $\mathbb{M}(s,c,\mathrm{k}) = \mathbb{R}(s,c,\mathrm{k}) = \{\}$.
- If k has the form "**if** E **then** k_1 **else** k_2 **fi**" and $\Phi(s,c,\mathrm{E}) = \delta(\mathrm{true})$, then $\mathbb{M}(s,c,\mathrm{k}) = \mathbb{M}(s,c,\mathrm{k}_1)$; $\mathbb{R}(s,c,\mathrm{k}) = \mathbb{R}(s,c,\mathrm{k}_1)$.
- If k has the form "**if** E **then** k_1 **else** k_2 **fi**" and $\Phi(s,c,\mathrm{E}) = \delta(\mathrm{false})$, then $\mathbb{M}(s,c,\mathrm{k}) = \mathbb{M}(s,c,\mathrm{k}_2)$; $\mathbb{R}(s,c,\mathrm{k}) = \mathbb{R}(s,c,\mathrm{k}_2)$.
- If k has the form "k_1, k_2", then $\mathbb{M}(s,c,\mathrm{k}) = \mathbb{M}(s,c,\mathrm{k}_1) \cup \mathbb{M}(s,c,\mathrm{k}_2)$; $\mathbb{R}(s,c,\mathrm{k}) = \mathbb{R}(s,c,\mathrm{k}_1) \cup \mathbb{R}(s,c,\mathrm{k}_2)$.

Note that the context conditions for statement lists guarantee that the two sets of pairs $\mathbb{M}(s,c,\mathrm{k})$ and $\mathbb{R}(s,c,\mathrm{k})$ are mappings and can be applied to store or channel names in their respective domains.

Suppose we have a processor p with definition "**proc** p[io] := k". For this statement list k $\mathbb{M}(s,c,\mathrm{k})$ and $\mathbb{R}(s,c,\mathrm{k})$ are defined because of the context conditions. Now for x in OS_p and y in TC_p,

$$M_p(s,c)(x) = \mathbb{M}(s,c,\mathrm{k})(x) \quad \text{iff} \quad x \in \mathrm{dom}(\mathbb{M}(s,c,\mathrm{k})),$$
$$R_p(s,c)(y) = \mathbb{R}(s,c,\mathrm{k})(y) \quad \text{iff} \quad y \in \mathrm{dom}(\mathbb{R}(s,c,\mathrm{k})).$$

4. An example

In this section a solution is given to the following problem.
Consider a number of factories each able to assemble products (called parts) from a number of subparts. Each subpart in turn can be obtained from a number of factories, where it is assembled from other subparts and so on. The number of subparts of a part may be zero; in that case the end of the recursion is reached. Assembling a part takes some time.
One can ask a factory f about the minimal time needed to assemble a certain part p. In general such a question will raise new questions about the time needed for assembling the subparts of p, and so on. Once all the subpart assembly times are known, f will answer the question.

The general idea of the solution is as follows.
Each factory is split into a processor which receives questions, a processor which receives answers, and five stores $s_1,...,s_5$ of type $t_1,...,t_5$ to which both processors are connected.
A question consists of a client (the sender of the question) and a number of parts. When a question is received, an answer is generated for all parts in the question for which all information is present. New questions are generated for all subparts for which not enough information is present and for which no outgoing question is pending (in s_4).
When an answer is received, a new answer is generated for all parts (in s_3) for which all information is now present.
Answers and questions are broadcasted: each answer- or question-processor receives everything and selects those things which are addressed to him. This broadcasting is performed by sending answers and questions to a special processor which transmits copies to each answer- or question-processor.

Each factory has the same description, the only difference being its identification.
In the next sections we give the types of the stores and channels, some local constraints, the functions which are used to describe the answer- and question-processors and finally the network describing one factory.

4.1 Types

Each factory has five stores of type $t_1,...,t_5$.
The stores of type t_1 and t_2 contain all product information about the parts the factory (ME) is able to assemble. This information is not modified by the questions and answers received.
Type t_1 describes the composition of each part a factory is able to assemble.
Data about the subparts of a part are needed to calculate the minimal assembly time of a certain part. Each subpart can be delivered by a number of factories, from which the one with the shortest time will be chosen. Type t_2 says how many suppliers for each subpart should have replied to a question about the subpart before this shortest time is calculated.
Questions about parts assembled by ME will in general not be answered immediately. In that case the asker of the question (the client) will be stored together with the parts for which no answer could be given yet in a store of type t_3.
When data about a subpart are needed it might be the case that those data are already requested by ME due to an earlier question. All subparts requested by ME are stored in a store of type t_4.
Finally a store of type t_5 will hold all data about subparts. In the process of receiving answers this store will be filled.

Connected to these types are a few functions to calculate some basic things.

- subpartsnotkown: all subparts (of a set p of parts) for which not enough information is present in a store s_5 of type t_5; that is the number of entries in s_5 for these subparts is not yet greater or equal to the number given in s_2.
- totaltime: the minimal total time needed for assembling a part p.
- partsknown: all parts (of a set p of parts) for which all subparts are totally described in s_5.

In the definition of the above functions other functions have been used, like appl[x:T→S, y:T]: S, the application of a mapping to an argument (denoted in infix notation as ·), \$[x:T→bool]: \$T, which constructs a set out of a mapping, and π_1[x:\$(T×S)]: \$T, which works on sets of pairs. All functions are eventually defined in terms of the basic functions given in 3.2.

> **type** part **from** str;
> **type** time **from** num;
> **type** fac **from** str;
> **type** client **from** fac;
>
> ME := 'fac0';
>
> **type** t_1 **from** part→(\$part × time); -- composition of each part which can be assembled by ME
> **type** t_2 **from** part→num; -- minimal number of suppliers for each subpart
> **type** t_3 **from** client→\$part; -- parts asked from ME
> **type** t_4 **from** \$part; -- subparts asked by ME
> **type** t_5 **from** part→(fac→time); -- subpart data
>
> subpartsnotknown [p:\$part, s_1:t_1, s_2:t_2, s_5:t_5]
> $:= \cup(\pi_1(s_1 \cdot p)) \setminus \$[x:\mathrm{dom}(s_5) \mid \mathrm{size}(\mathrm{dom}(s_5 \cdot x)) \geq s_2 \cdot x]$;
>
> totaltime [p:part, s_1:t_1, s_5:t_5]
> $:= \max_0[x:\pi_1(s_1 \cdot p) \mid \min(s_5 \cdot x)] + \pi_2(s_1 \cdot p)$;
>
> partsknown [p:\$part, s_1:t_1, s_2:t_2, s_5:t_5]
> $:= \$[y:p \mid \pi_1(s_1 \cdot y) \subset \$[x:\mathrm{dom}(s_5) \mid \mathrm{size}(\mathrm{dom}(s_5 \cdot x)) \geq s_2 \cdot x]]$;
>
> partsnotknown [p:\$part, s_1:t_1, s_2:t_2, s_5:t_5]
> $:= p \setminus \mathrm{partsknown}(p, s_1, s_2, s_5)$;

There are two trigger channels involved: one for questions and one for answers.
A question consists of the identification of the client and the parts for which information is wanted. A factory will only consider those parts in a question which it is able to assemble.
An answer consists of a number of addressees (clients who have asked something in the past) and information about parts as assembled by the factory who supplies the answer. We have chosen for an answer type resembling t_5 to facilitate the update of stores of type t_5.

There are two functions to check whether a question or an answer is empty (that is containing no parts).
- questionempty: the question does not contain any parts.
- answerempty: for each client in the answer there are no parts present.

> **type** question **from** client × \$part;
> **type** answer **from** client→(part→(fac × time));

questionempty [q:question]
    $:= \pi_2(q) = \{\}$;

answerempty [a:answer]
    $:= \forall$ [x:rng(a) | dom(x) = { }];

4.2 Constraints

In principle a lot of constraints can be formulated. For example, when one wants to have an answer to any valid question, it should hold that there are at least as many producers of a part as suggested by the stores of type t_2 at every factory. These kind of constraints will not be considered here.
Locally one can formulate four constraints $c_1,...,c_4$ which are essentially subset requirements. A constraint that says that there are no questions left in s_3 which could have been answered is not given here, since its formulation is almost the same as the formulation of the functions in 4.3 and 4.4.

c_1 [s_1:t_1, s_2:t_2]   -- every subpart in s_1 is mentioned in s_2 and vice versa
    $:= \mathrm{dom}(s_2) = \cup(\pi_1(\mathrm{rng}(s_1)))$;

c_2 [s_1:t_1, s_3:t_3]   -- only questions will be answered about parts known by ME
    $:= \cup(\mathrm{rng}(s_3)) \subset \mathrm{dom}(s_1)$;

c_3 [s_1:t_1, s_4:t_4]   -- only questions asked about subparts needed by ME
    $:= s_4 \subset \cup(\pi_1(\mathrm{rng}(s_1)))$;

c_4 [s_1:t_1, s_5:t_5]   -- only data present about subparts needed by ME
    $:= \mathrm{dom}(s_5) \subset \cup(\pi_1(\mathrm{rng}(s_1)))$;

4.3 Question received

Upon receipt of a question the stores s_3 and s_4 will be updated and an answer and a new question are generated. In case the question is not meant for ME no update, answer, or new question has to be generated. There is no need to check this explicitly: it suffices to construct a new question and an answer, and when these are essentially empty they are simply not transmitted.
From the question (q) only those parts should be selected that are present in s_1: $\pi_2(q)\cap\mathrm{dom}(s_1)$.
The answer (qac) is a mapping, but since it contains only one element it is more conveniently written as a set of one pair.

qs_3 [q:question, s_1:t_1, s_2:t_2, s_3:t_3, s_5:t_5]
    $:=$ [x:dom(s_3)$\cup\{\pi_1(q)\}$ | **if** $x\in\mathrm{dom}(s_3)$ **then** $s_3\cdot x$ **else** {} **fi**
        $\cup$
        if $x=\pi_1(q)$ **then** partsnotknown($\pi_2(q)\cap\mathrm{dom}(s_1)$,$s_1$,$s_2$,$s_5$) **else** {} **fi**];

qs_4 [q:question, s_1:t_1, s_2:t_2, s_4:t_4, s_5:t_5]
    $:= s_4 \cup$ subpartsnotknown($\pi_2(q)\cap\mathrm{dom}(s_1)$,$s_1$,$s_2$,$s_5$);

qqc [q:question, s_1:t_1, s_2:t_2, s_4:t_4, s_5:t_5]
    $:=$ ME $\times$ (subpartsnotknown($\pi_2(q)\cap\mathrm{dom}(s_1)$,$s_1$,$s_2$,$s_5$) $\setminus s_4$);

```
qac [q:question, $s_1$:$t_1$, $s_2$:$t_2$, $s_5$:$t_5$]
    := {$\pi_1$(q) × [x:partsknown($\pi_2$(q)∩dom($s_1$),$s_1$,$s_2$,$s_5$) | ME × totaltime(x,$s_1$,$s_5$)]};
```

4.4 Answer received

Upon receipt of an answer the stores s_3, s_4, and s_5 will be updated and a new answer is generated.
Only those subparts of the incoming answer (a) have to be selected that are meant for ME: dom(a·ME).
To construct the answer it is necessary to use the new state of store s_5: $as_5(a,s_5)$. This new state is also necessary in the update of the stores s_3 and s_4, which contain the incoming and outgoing questions which are still pending.
A possibly empty answer can be detected with the help of a function defined in 4.1.

```
$as_3$ [a:answer, $s_1$:$t_1$, $s_2$:$t_2$, $s_3$:$t_3$, $s_5$:$t_5$]
    := [x:dom($s_3$) | partsnotknown($s_3$·x,$s_1$,$s_2$,$as_5$(a,$s_5$))];
$as_4$ [a:answer, $s_2$:$t_2$, $s_4$:$t_4$, $s_5$:$t_5$]
    := $s_4$ \ $[x:dom($as_5$(a,$s_5$)) | size(dom($as_5$(a,$s_5$)·x)) ≥ $s_2$·x];
$as_5$ [a:answer, $s_5$:$t_5$]
    := if ME ∈ dom(a)
         then [x:dom($s_5$)∪dom(a·ME) | [y  :  if x∈dom($s_5$) then dom($s_5$·x) else {} fi
                                               ∪
                                               if x∈dom(a·ME) then {$\pi_1$(a·ME·x)} else {} fi
                                            |  if x∈dom(a·ME)
                                               then if y=$\pi_1$(a·ME·x) then $\pi_2$(a·ME·x) else $s_5$·x·y fi
                                               else $s_5$·x·y fi]]
         else $s_5$ fi;
aac [a:answer, $s_1$:$t_1$, $s_2$:$t_2$, $s_3$:$t_3$, $s_5$:$t_5$]
    := [y:dom($s_3$) | [x:partsknown($s_3$·y,$s_1$,$s_2$,$as_5$(a,$s_5$)) | ME × totaltime(x,$s_1$,$as_5$(a,$s_5$))]];
```

4.5 The network

In this section we give the definition of the network of stores and processors describing one factory.
Since all necessary functions for updating the stores and constructing reactions upon questions and answers are defined in the preceding sections, the definition of the network itself will be very short. We only have to give the declarations of the stores s_1,...,s_5, the question and answer channels, and the two processors involved. These stores and channels have been used in a parametrized way in the preceding sections.
Each processor has only one input trigger, the question processor (qproc) a question, and the answer processor (aproc) an answer.
Note that we did not care about updates that are superfluous: one can easily add a test for each store update to see whether the new value differs from the old one or not.

```
store $s_1$: $t_1$;
store $s_2$: $t_2$;
store $s_3$: $t_3$;
store $s_4$: $t_4$;
store $s_5$: $t_5$;
channel q: question;
channel a: answer;
proc qproc [tin q, tout a,q, sin $s_1,s_2,s_3,s_4,s_5$, sout $s_3,s_4$]
    :=  $s_3 \leftarrow qs_3(q,s_1,s_2,s_3,s_5)$,
        $s_4 \leftarrow qs_4(q,s_1,s_2,s_4,s_5)$,
        if $\neg$questionempty($qqc(q,s_1,s_2,s_4,s_5)$) then q $\Leftarrow$ $qqc(q,s_1,s_2,s_4,s_5)$ fi,
        if $\neg$answerempty($qac(q,s_1,s_2,s_5)$) then a $\Leftarrow$ $qac(q,s_1,s_2,s_5)$ fi;
proc aproc [tin a, tout a, sin $s_1,s_2,s_3,s_4,s_5$, sout $s_3,s_4,s_5$]
    :=  $s_3 \leftarrow as_3(a,s_1,s_2,s_3,s_5)$,
        $s_4 \leftarrow as_4(a,s_2,s_4,s_5)$,
        $s_5 \leftarrow as_5(a,s_5)$,
        if $\neg$answerempty($aac(a,s_1,s_2,s_3,s_5)$) then a $\Leftarrow$ $aac(a,s_1,s_2,s_3,s_5)$ fi.
```

References

[BJ82] Bjorner, D., and C.B. Jones, Formal specification and software development, Prentice-Hall, 1982.

[HA87] Harel, D., Statecharts, a visual approach to complex systems, Sci. Comp. Prog. **8-3** (1987).

[HA87a] Hayes, I. (ed.), Specification case studies, Prentice-Hall, 1987.

[HE88] van Hee, K.M., G.J. Houben, L.J. Somers, and M. Voorhoeve, Executable specifications for information systems, submitted to IFIP Working Group 8.1 Conference (Computerized Assistance during the system life cycle) 1988.

[HE88a] van Hee, K.M., G.J. Houben, L.J. Somers, and M. Voorhoeve, Distributed event systems, to appear.

[HE88b] van Hee, K.M., L.J. Somers, and M. Voorhoeve, The language EXSPECT, to appear.

[HO88] Houben, G.J., J.L.G. Dietz, and K.M. van Hee, Discrete event systems: models and applications, P. Varaiya, A.B. Kurzhanski (*eds*), Lecture Notes in Control and Information Systems 103, Springer Verlag, 1988.

[LU79] Lundberg, M., G. Goldkuhl, and A. Nilsson, A systematic approach to information systems development, Information systems **4** (1979).

[PE81] Peterson, J.L., Petri net theory and the modeling of systems, Prentice-Hall, 1981.

[RO77] Ross, D.T., M.E. Dickover, and C. McGowan, Software design using SADT, Auerbach Publishers Portfolio **35-05-03** (1977).

[WA85] Ward, P.T., and S.J. Mellor, Structured development for real-time systems, Yourdon Press, 1985.

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 distri-buted 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 nondeter-ministic 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