Executable specifications for information systems

3348
335859

Executable Specifications for Information Systems

by

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

88/05

March 1988

COMPUTING SCIENCE NOTES

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

Executable Specifications for Information Systems

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

Eindhoven University of Technology

ABSTRACT

In this paper we present a survey of a framework for modeling and specifying information systems. Our method [4, 5] is supported by a software tool for checking and executing specifications. An executable specification may be considered as a prototype for a target system. The specification language resembles the language of mathematics; it is related to the Z and VDM methods [1, 3]. However, specifications in Z and VDM are descriptive (and therefore not executable), whereas ours are constructive. All these methods share almost the same power and versatility.

In section 1 we give our viewpoint on the engineering of information systems. In section 2 we give an informal treatment of our framework and design method. In section 3 we survey the specification language. In section 4 our solution to the inventory control case study [6] is presented and finally in section 5 we give a full specification of that case study.

1. INTRODUCTION

Software engineering is a branch of systems engineering focussed on the automatic control of tasks in a system. A system is characterized formally by a state space and some transition mechanism that transfers the system from one state into another one. Here, we restrict ourselves to discrete dynamic systems, which means that we describe the behaviour of a system by a (possibly infinite) sequence of states.

In general the state spaces of a system can be considered as a cartesian product and therefore its states can be considered as vectors.
The transition mechanism often consists of one or more processors. The behaviour of a processor is described by a function that has two types of arguments: a trigger and a subvector of the system state. Its effect may consist of a change of the state subvector and a set of triggers for other processors or possibly itself. Processors may be implemented by persons, machines or computer systems. A state of a system may be determined by the presence of sets of physical or abstract objects such as products and agreements.
When considering a business system in more detail we distinguish a subsystem that executes the primary tasks of the business system, called the primary system, and a system that controls the primary system, called the control system.

Automated information systems play a role in control systems. One of the classical tasks of an (automated) information system is keeping track of the state of the primary system. A variable, called a database, contains information about the actual states possibly combined with past states of the primary system. The case study belongs to this class of information systems.

A more advanced task of an information system is the support of decision makers in control systems. These subsystems are called expert systems or decision support systems.

In section 2 we sketch a formal framework to model such systems. The need for such descriptions is demonstrated by the wide use of dataflow diagram techniques in methods as SADT [9], ISAC [7] and Yourdon [10]. These techniques lack good semantics and hence are not suitable for the precise specification of an (information) system, although the diagrams often help to understand more formal specifications. Another approach to systems description is data modeling. It can be used for describing the state space of a primary system or its image in an information system. Data modeling techniques are much more formal, but they are only suitable for modeling state spaces.

Our approach seems to be interesting because it integrates formal modeling of processors and state spaces, combined with diagramming techniques. In earlier work [4] we used a similar framework, supported by a logic language.

Main objects to specify are variables, sometimes having a complex structure, and functions. In fact a database scheme defines a type for a variable called database. What we need is a type system that allows us to define rather complex types for variables, and a mechanism to define functions.
A typed lambda calculus or functional language seems to be a natural choice. This is the basis of our language, called EXSPECT. A nice feature of it is that we are able to stick to the relational model but that it is also possible to work in non-first-normal-form. In the application we have chosen for the last option.

Our software tool consists of an editor, a type checker and an interpreter. With the type checker one can test a description for type consistency. With the interpreter one can simulate the behaviour of the described system. This last facility is essential for validation purposes: for non-experts it is difficult to understand a formal system description. On the other hand it is relatively easy for future users of a system to validate a prototype, generated from an executable specification.

An important difference between an executable specification and a real implementation is that the designer of the specification is only concerned with the functionality of a system and not with matters like performance, system load, reliability, concurrency, etc. Therefore a specification language may use more powerful constructs than an implementation language; so it is much faster to design a specification than an implementation.

The first step in the lifecycle of an information system is the description of the environment, i.e. the primary system and possibly parts of the control system of a business system. It is possible to model this on several levels of detail within our framework.
One usually proceeds with requirements engineering as the next step. Here the tasks of an information system are defined. Usually the functional and non-functional requirements are written down informally. We then have a preliminary specification. If it were used as a specification for implementation then with high probability the resulting implementation would be inadequate. We all know that system changes are very expensive. Therefore we advocate a third phase of the lifecycle that is devoted to a formal specification in the way sketched above. We call this conceptual modeling because this phase produces an abstract system that has the same functionality as the target system. With an executable specification we already have a primitive implementation of the target system.

2. FRAMEWORK AND DESIGN METHOD

Systems have three main aspects: state structure, data flow and control flow. Many methods are available for each individual aspect. As said before, state structure can be described by data models, data flow by data flow diagrams and control flow can be modeled for instance by Petri nets [8] or finite state machines.

Our framework integrates all three aspects. The difference between control flow and data flow in our framework is that the control flow directs the transport of parameters to processors, triggering the incorporated functions, while data flow means transport of parameters between a processor and a (stored) variable. For a formal treatment we refer to [4]. 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 and *discrete* means that each state on a process path has a successor. 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 8-tuple $<S, C, IC, OC, M, R, IS, OS>$.
Before we explain the meaning of these components we give two diagrams of a DES. Of course it is possible to combine the two diagrams into one.

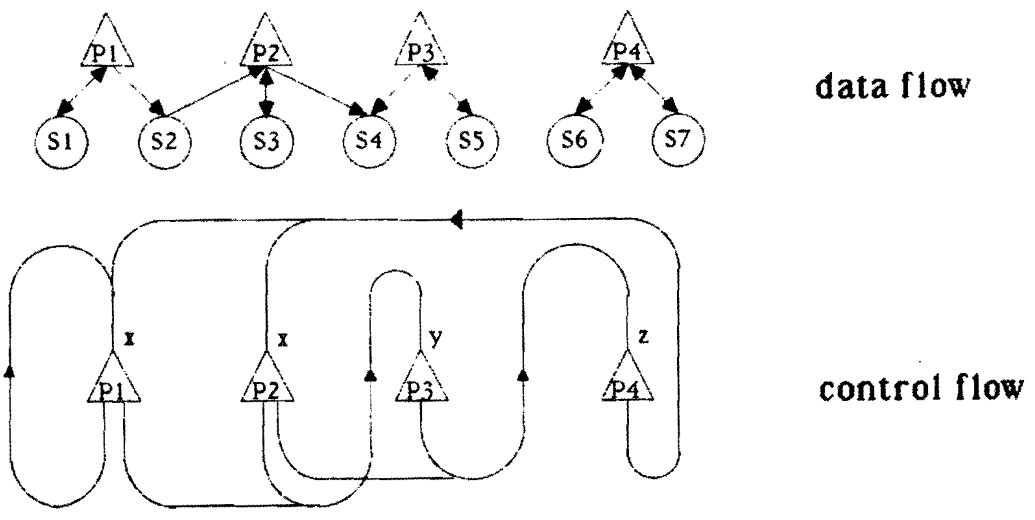

Fig. 1

The triangles $P_1 \cdots P_4$ represent processors. Each processor i consists of two functions: M_i and R_i. The circles $S_1 \cdots S_7$ represent (stored) variables. They may have simple structures like a calendar date, or complex like a database. The connections between processors and stores mean that a processor may acces the variable. If there is an arrow in the direction of the variable then it is an output variable for the processor, if an arrow points to the processor it is an input variable. 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 type of the values that may pass through it are determined. Channels may split and 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. It is easy do deal with cases where it is intuitively felt natural to have more than one trigger channel. Trigger channels may be considered as mailboxes. The values passing through a channel are called triggers.

In fig. 1 we have already met four of the components of the 8-tuple:

- IC is a function that assigns to each processor one input channel-index, in the picture x ,x ,y ,z for respectively P_1,P_2,P_3 and P_4.
- OC is a function that assigns to each processor a set of output channel- indexes for $P_1 : x , y$, for $P_2 : y , z$ etc.

An output channel is connected to all input channels with the same index.

- IS is a function that assigns to each processor a set of indexes of (stored) variables that are used as input variables for that processor; for $P_1 : S_1$ for $P_2 : S_2$ and S_3 for $P_3 : S_5$ etc.
- OS is a function similar to IS, it assigns to each processor a set of indexes of output variables; for $P_1 : S_1 , S_2$, for $P_2 : S_3 , S_4$, for $P_3 : S_4 , S_5$ etc.

Now we will explain S and V.

- S is a set-valued function, where $dom(S)$ is the set of indexes of stored variables and for such an index i , S_i is a set that represents the type of the variable with index i.
- C is a set-valued function, where $dom(C)$ is the set of channel indexes and for such an index j , C_j represents the type of the triggers passing through the channel.

Finally we return to M and R.

- M is a function-valued function, where $dom(M)$ is the set of processor indexes. For a processor k , M_k is a function with input variables with indexes in IS_k and IC_k and output variables with indexes in OS_k.
  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 indexes. For a processor k, R_k is a function with the same input variables as M_k however its result is a partial function that assigns a value to zero or more trigger variables with indexes in the set OC_k. R_k is called the *reactor* of processor k because it produces triggers.

The functions M_k and K_k are specified by means of a typed lambda calculus or functional language. This is treated in section 3.

We will describe the behaviour 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 input parameters the values of the input variables 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.
Note that we not specify how processors select triggers from their multiset, nor how they control the exclusive updating of output variables. It is left to the implementers 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 parallellism. Since for a DES the selection of triggers to be executed is not specified, it may be considered a non-deterministic system.

Many systems may be modeled as a DES, for instance many communication protocols between two systems can be modeled explicitely within the framework. Then we model in fact the communication at a higher level, while the implicit way of triggering is the lowest level of communication.
An important modeling issue, in connection with the case study, is the separation of a closed system into a target system and its environment. The target system is the information system we want to develop. Many information systems can be modeled at a high level as a reactive system, i.e. the environment offers a trigger and the system performs one transition and a trigger for its environment. In that case there is no internal triggering in the system. We may model the environment as one or more processors, possibly with stored variables. However, the specification of these processors is unknown, as are their stored variables.
The processors are considered as black boxes, their in- and output channels only are known (see fig. 2).

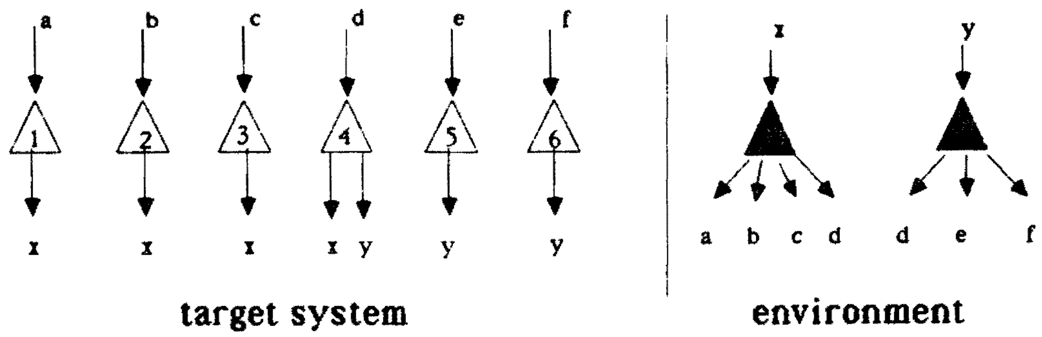

Fig. 2

Blackbox *x* may trigger processors 1 ,2 ,3 and 4, blackbox *y* 4 ,5 and 6, and every processor is producing a trigger for the invoker, however processor 4 may trigger both blackboxes.
It is also possible to model that a processor in the environment may access stored variables of the target system.
In the case study most system tasks are of the reactive type. We can model the several access control classes as different blackbox processors.

We conclude this section with some remarks on a design method based on our framework. We only consider the conceptual modeling phase. It is quite natural to proceed along the following steps:

1. Identify the processors and stored variables in the target system and identify the blackboxes in the environment (in fact this is a data flow analysis).
2. Identify the channel structure (control flow analysis).
3. Define types for the stored variables (data modeling).
4. Define constraints on the types of stored variables (database constraints).
5. Define types for the trigger variables.
6. Define the manipulator and reactor functions for each processor.
7 Verify that the processors keep the constraints invariant.

Of course, it is sometimes useful to change the order or to take on two steps simultaneously. However, if all 7 steps are accomplished the specification is complete.

3. LANGUAGE

In this section we describe a language for specifying systems according to the model defined in section 2. The description is given in an informal way, for a more concise treatment we refer to [5].
A DES is built from processors and stores. A processor repeatedly selects a trigger from the available triggers, updates the values of the stores it is connected to and sends new triggers to other processors. The language EXSPECT, of which a subset is treated here, is suited for specifying and executing such systems.

From now on we call stored variables stores and they may be considered as global variables. They are declared by giving their name and type. For example,

```
    store  s: str
```

declares a store of name s and type str(ing).
Processors are defined by giving their name, the type of the input trigger, and the actions they perform. For example,

```
    proc  p1[i:str] ::= s ← i,
                        q <= 'store updated'
```

This processor is named p1 and is triggered by a string. When p1 reacts upon a certain trigger it stores the value of this trigger, which is denoted by i, in the store s we have declared above. Furthermore it sends a trigger with value 'store updated' to processor q. In the present version of EXSPECT, input channels cannot be shared by processors and therefore we identify a processor and its input channel.
For each store updated there is a line containing a ← and for each trigger sent there is a line with a <=.

In general, processors also transform input values of stores and triggers into other values. This is where the functional aspect of EXSPECT comes in. At the right hand side of a ← or <= sign we may use any function of the input trigger of the processor and the stores in the system, which are connected to this processor as input stores. A function is defined in terms of other functions and so on till we reach the basic functions of the language. For example, we may define a function to calculate the length of a string

```
    strlen[x:str] ::= if  x = ''  then  0
                                  else  strlen(tail(x)) + 1
                      fi
```

with the help of already existing functions to add numbers (+) and to take all but the first character of a string (tail).
This new function in turn can be used for defining other functions.

Assignments to stores and triggers in the definition of a processor can also be done conditionally. A processor p2 which only updates store s when the length of the trigger is more than 10 is given by

```
    proc  p2[i:str] ::=  if  strlen(i) > 10  then  s ← i,
                                                   q <= 'store updated'
                                             else  q <= 'store not updated'
                         fi
```

Untouched stores, like s in the second alternative of the **if**, are left invariant.
Up to this point we have dealt only with simple types like "bool", "num" and "str"; respectively for boolean values (true/false), rational numbers and strings. For modeling more real-life situations we need more complex types like sets.
With the help of the type constructors ×, $ and → we can construct compound types of arbitrary complexity.
Cartesian products are constructed with the help of ×, for example pairs of numbers, or triples of bool,number,string:

```
num × num

bool × number × string
```

Sets are constructed with the help of $ and mappings (functions with finite domains, to be interpreted as sets of pairs) with the help of →. Examples are

```
$num

num → bool

$num × $bool
```

for a finite set of numbers, a mapping from num to bool and pairs of sets of numbers and booleans.
We are now able to define a processor p3 that updates a store t that holds a set of strings,

```
store t: $str;

proc  p3[i:str] ::=  if  strlen(i) > 10  then  t ← ins(i,t),
                                                q <= 'store updated'
                                           else  q <= 'store not updated'
                     fi
```

Here "ins" is a basic function that inserts an element in a set.
New types can also be introduced by giving them a name. We can introduce a type addr(ess), which holds street, house, town and postal code (all considered to be strings) by

```
type  addr from str × str × str × str
```

A store (with the name "index") to hold names and addresses can be declared by

```
type  name from str;

store  index: name → addr
```

We have used a mapping, since for each name there is never more than one address.
A processor p4 that adds a name, which is not yet present, and address to "index" may be written as

```
proc  p4[i:name × addr] ::=  if  π1(i) ∉ dom(index)  then  index ← ins(i,index)
                             fi
```

The function π1 projects upon the first element of a pair. This process can also be written as

```
proc  p5[i:name, j:addr]::=
   if  i ∉ dom(index)  then  index ← [x:ins(i,dom(index)) | if  x=i  then  j  else index · x  fi ]
   fi
```

The constructor [x: D| E(x)] defines a mapping with domain D and range E(D). So the example assigns to (the store) index a new mapping that has the same domain as the old index, but with the new name added. The values of the mapping are the old ones (index · x means apply index to x) and the new address.

Yet a third way to represent the above processor is

```
proc p6[i:name, j:addr]::=
  if i ∉ dom(index) then index ← fupd(index,[x:{i} | j])
  fi
```

Here {i} is the set with i as only element, [x:{i} | j] is therefore the mapping consisting of only one pair ≪ i,j ≫ and "fupd" (defined formally in section 4.4) is a general function that accepts two mappings as parameters and returns the "overwriting" of the first mapping by the second one, it is defined formally in section 4.4.

In the above we have given enough information about the language to understand the case in the next section.
It is possible to construct libraries of functions. These libraries will assist in developing a description in a modular way.
Apart from libraries of functions one can also make toolboxes of parametrized processors or networks of processors. These can be used to assemble a system from existing parts. The part of the language that deals with these modular networks is not treated in this paper, since the case of the next chapter is essentially a flat one.

4. THE INVENTORY CONTROL SYSTEM IN EXSPECT

4.1 Control Flow and Data Flow

The first step in designing an EXSPECT prototype for an information system consists of designing the control and data flow of the various processors of the system. First we must draw a boundary between the system and its environment.

In the inventory control system of the case study [6], the environment consists of a number of users who can perform a selection out of several tasks. A dialogue guides the user to the task he wishes to perform and prompts him for the right parameters for this task. Before even this dialogue starts, the user must login to the system, whereby his acess control class becomes known.

We have chosen to exclude the dialogue and access control part from our system because it is not typical for this case. Therefore our environment consists of a number of *user agents*, who can trigger any of the task processors. A few *background* task processors are not triggered by any user agent but by some of the *foreground* processors. Our control flow scheme thus becomes as follows.

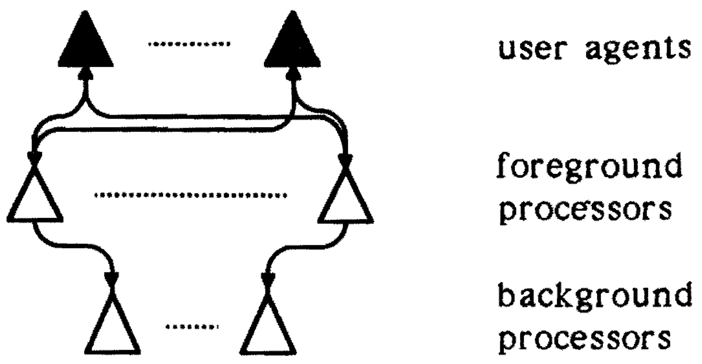

Fig. 3

For the data flow, we model our database as a single stored variable. The user agents have no access to it; some of the processors ("queries") only consult the database, others ("updates") also modify it. The data flow scheme thus becomes as follows.

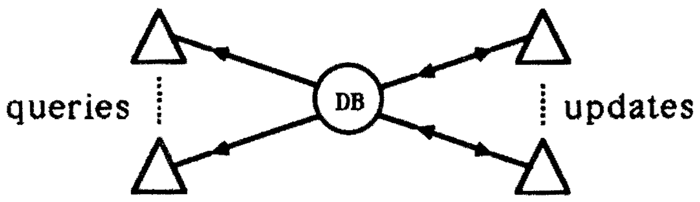

Fig. 4

For maximal clarity (since our goal is a prototype) we have reduced the number of user agents to one. This single-user system can be converted into a multi-user one by extending the trigger of each foreground processor with the user agent index of the caller and adding code to send the response to the caller.

4.2 Datatypes and Stores

The second step in designing the prototype is to design a structure for the stored database variable. This step is (for a strongly data-oriented case like this) more important than the preceding one. As mentioned in the introduction, we can choose to do so in various ways, ranging from many "flat" parts to few "structured" ones. To demonstrate the data structuring capabilities of EXSPECT we have chosen for this last option. To understand the following discussion one must study the case description [6].

We divide the database into five parts, called respectively the stocked item type file (sitf), stock item file (sif), supplier file (supf), purchase order file (pof) and the calendar (cal). Since we have no need for the database as a whole, we model these parts as separate stores.

The more or less compound "attributes" of the above stores are often described by defining a special "derived" datatype for them; these datatypes also serve as a vehicle for triggering processors. Inside the user agent formatting and checking information could be attached to them.

The stock item type file (sitf) is as specified in the case; it consists of attributes stock item type code (sitc) and description (sitd). Since the sitc attribute must be unique, we model sitf as a store of type "mapping of sitc to sitd", where sitd and sitc are both types derived from "string". We write this formally as follows.

```
type sitc from str;
type sitd from str;
store sitf : sitc → sitd
```

The value for this variable as given in the case description would be represented as the following set of pairs.

```
{   ≪ 'E','Office Equipment (capital expense)' ≫ ,
    ≪ 'S','Stationary supplies' ≫ ,
    ...
    ≪ 'K','Kitchen supplies' ≫
}
```

The stock item file (sif) consists of the "flat" attribute structure as specified in the case: stock item code (sic), stock item type code (sitc), stock item description (sid), replenishment level (type qty: quantity). To this is added a "history" component, containing the recorded stock levels (date and qty) together with the withdrawals (qty and issue) and replenishments (qty and purchase order responsible for it) at that moment. The date forms a key to a recorded stock level. Our "sif" store thus combines the stock item, stock on hand, replenishment and withdrawal files in the case description.

We could have added all stock items of a certain type as an attribute to the same stock item type in the "sitf" store. This would however make the retrieval of a stock item on its code quite cumbersome. Remodeling the stock item code as a pair ≪ sitc, n ≫ removes this disadvantage. At the same time it would be nice to deduce the item type directly from its code without accessing the database. We have however stuck to the description as given and therefore chosen to model "sif" as a separate store as follows.

```
type date from num;
type sic, sid, qty from num;
type sidat from sic × sid × qty;
type ponr from num;                -- purchase order nr
type wdr from qty × str;
type repl from qty × ponr;
type history from date → (qty × $wdr × $repl);
store sif : sic → (sidat × history)
```

A possible value for the "sif" variable would be as follows.

```
{     ≪ 5632, ≪ ≪ 'E', 'Compaq Plus Computer', 0 ≫ , {} ≫ ≫ ,
      ≪ 2389, ≪ ≪ 'F', 'Paracetamol', 144 ≫ ,
         { ≪ 870901,{ ≪ 1,'headache' ≫ , ≪ 2,'lost' ≫ }, {} ≫ ,
          ≪ 871111,{ ≪ 24,'lost' ≫ },{ ≪ 288, 74324 ≫ } ≫ } ≫  ≫
}
```

The supplier file (supf) with key supplier number (supnr) has as attributes the name, address and phone number of the supplier plus the set of item types he sells. This store thus combines the supplier and supplier of stock item type files. We model it as follows.

```
type supnr from num;
type phone, addr, name from str;
type supdat from name × addr × phone;
store : supnr ⟶ (supdat × $ sitc)
```

The purchase order file (pof) with key purchase order number (ponr) has as attributes the order date and the supplier plus the set of ordered items. This set is modeled as a mapping (pol) from "sic" to price (per unit) and quantity. This store thus combines the purchase order and purchase order line files. It is described as follows.

```
type podat from date × supnr;
type pol from sic ⟶ (price × qty);
store pof : ponr ⟶ (podat × pol)
```

The calendar (cal) consists of a single date variable.

```
store cal : date
```

After defining the stores and auxiliary types, it is helpful to define auxiliary functions based upon these stores. For instance, the order date of an order x is represented much more nicely by the expression "date (x)" then by " $\pi 1$ ($\pi 1$ (pof · x)". It does not matter that there exists already a type "date", because the parser knows when to expect a type or an expression. There could even be more functions named "date", provided their parameter types do not conflict.

One of the more complicated auxiliary functions computes the stock level of item x at date y. Since we only record stock level changes, this involves searching the history of x to find the last recorded change before or at date y. If item x has no history at or before date y (e.g. at date y it had just been decided to keep the item in stock and orders had been placed but no supply had arrived yet), the function must return 0. In full the definition reads

```
qty [x:sic, y:date] :=
 if $[t: dates(x) | t ≤ y] = {} then 0
                                 else π1 ( hist(x) · max ( $ [t: dates(x)| t ≤ y]) )
 fi
```

The expression " $[t: dates(x)| t ≤ y]" denotes the set of stocklevel change dates for the item x that lie before or at the date y. Taking the maximum of this set gives the last recorded change date before or at y. Applying the history to this date and taking the first part yields the stock level recorded at that date.

4.3 Constraints

The next step in the design process is the formulation of constraints. These are computable boolean expressions depending on the store contents. The designer of the prototype must show that every processor changing the store contents leaves the constraints invariant, i.e. assuming that they are true in the old state, they must be true in the new state too.

In our design for the inventory control case, a lot of constraints as formulated are immediately guaranteed by the store definition. For instance uniqueness constraints are met by defining stores as mappings. A lot of referential constraints are met by combining files into a single non-first-normal-form store. There are some referential constraints left, for instance

"each stocked item has an existing type"

which is represented as

$\forall$ [x: dom(sif)| type(x) $\in$ dom(sitf)].

In studying the above formula, one sees how closely EXSPECT text resembles conventional mathematical notations.

There are some more interesting constraints, not mentioned in the case description; for instance,

"for each replenishment of a certain item, there must exist an order line for the same item; the replenishment date must not exceed the order date; the sum of all quantities replenished for the same order may not exceed the quantity ordered."

The EXSPECT formulation of the above constraint (using some earlier defined notions) becomes

$\forall$ [x: dom(sif)| $\forall$ [y:dates(x)| $\forall$ [z: repls(x,y)|

po(z) $\in$ dom(pof) **and** x $\in$ items(po(z)) **and** date(po(z)) $\leq$ y

and reporqty(x,po(z)) $\leq$ orqty(po(z)),x)]]]

The latter expression may be harder to understand (and to formulate) than the former, its meaning is uniquely determined. If legal texts were written in EXSPECT, a lot of lawyers would lose their jobs.

4.4 Processors

The following step in designing a prototype is to specify the diverse processors in the system. Having specified the structure of the stores, we must adapt the functionality of the system. The automatic generation of purchase orders is impossible, since data is lacking (suppliers per item, price per supplier-of-item). Instead, this processor produces a list of items that have to be ordered.

Now is the moment to go back to step 1; we identify the 25 processors (23 foreground and 2 background) as given in the case description, and specify their control flow in greater detail; also the data flow can be specified in more detail, having distinguished 5 stores.

Our task then becomes to determine the trigger types of each processor and determine its definition. For the user agent, we define the trigger "report" of type string.

As an example we treat action 4 of [6] (the addition of new items). It requires as input a list of items, each item consisting of item code, type code, item description and reorder level. Since the item codes are all different, we model the input type as

$$\text{sic} \rightarrow \text{sidat}$$

To keep our constraints invariant, the item codes must be new and the type codes (included in sidat) must exist already. If these input requirements are met, the items are added to the "sif" store together with an empty history. To achieve this we call the input x and define the mapping

$$f := [y: \text{dom}(x) \mid \ll x \cdot y, \{\} \gg]$$

So f is a mapping derived from x; each y in its domain is mapped to the pair formed by the value of x in y and the empty set. This f is thus the transformation of the input to "sif"-compatible format; the "sif" store is updated with this f. If the input requirements are not met, a message is sent to the user. In full the specification of action 4 is as follows.

```
AddItems [x: sic → sidat] ::=
 if  dom(x) ∩ dom(sif) = {}
 then if  π1 (rg(x)) ⊂ dom (sitf)
         then  sif ← fupd (sif, [t: dom(x)| ≪ x · t, {} ≫ ]),
               report <= 'ok'
         else  report <= 'undefined key in sitf'  fi
 else  report <= 'key conflict in sif'  fi
```

The generic function fupd used here accepts two mappings f and g of type A → B and returns the mapping h with as domain the union of the domains of f and g; an element x in the domain of h is mapped to f·x if x was in the domain of f, otherwise to g·x. Formally

```
fupd [f: A → B, g: A → B] :=
     [x: dom(f) ∪ dom(g) | if x ∈ dom(f) then f·x else g·x fi ]
```

In this way, we have modeled each of the 25 actions of [6]. A few concluding remarks have to be made. Action 13 (adding dates) is altered to setting a new system date. Also we have "sinned" by letting background processors perform checks and report to the user agent.

5. SPECIFICATION TEXT

```
     -- types
type date from num;                            -- yymmdd
type qty from num;                             -- natural numbers only
type phone from str;                           -- leading zero's are significant
type addr from str;                            -- street + house + town + code
type price from num;                           -- multiples of .01
type sitc from str;                            -- stock item type code
type sitd from str;                            -- sit descr
type sic from num;                             -- stock item code
type sid from str;                             -- si descr
type sidat from sitc >< sid >< qty;            -- type, descr, replenishlevel
type supnr from num;                           -- supplier number
type supdat from str >< addr >< phone;         -- name, address, phone
type pol from sic -> (price><qty);             -- purchase order lines:
                                               -- si code, unitprice, qty

type ponr from num;                            -- purchase order nr
type podat from date >< supnr;                 -- po data
type wdr from qty >< str;                      -- qty and issue of withdrawal
type repl from qty >< ponr;                    -- qty and purchase order
type history from date -> (qty)><$wdr><$repl);
                                               -- stock history of item:
                                               -- moment of change, new qty,
                                               -- set of withdr. and repl.


    -- stores
store sitf : sitc -> sitd;                     -- stock item type file
store sif : sic -> (sidat><history);           -- stocked item file:
store supf : supnr -> (supdat><$sitc);         -- supplier file
store pof : ponr -> (podat><pol);              -- purchase order file
store cal : date;                              -- current date

     -- auxiliary functions (data)
data [x:sic] := π1(sif·x);                     -- data of x
type [x:sic] := π1(data(x));                   -- type of stocked item x
rlev [x:sic] := π3(data(x));                   -- replenishment level of x
hist [x:sic] := π2(sif·x);                     -- stock level history of x
dates [x:sic] := dom(hist(x))                  -- change dates for stocklevel of x
qty [x:sic, y:date] := if $[t: dates(x)| t ≤ y] = {} then 0 else
     π1 ( hist(x) · max ($[t: dates(x)| t ≤ y]) ) fi;
                                               -- qty of x in stock at date y
curqty [x:sic] := qty (x, cal);                -- qty of x now in stock
wdrs [x:sic, y: date] := if y ∈ dates(x) then π2(hist(x)·y) else {} fi;
                                               -- withdrawals of x at date y
repls [x:sic] := U(π3(rg(hist(x))));           -- set of replenishments of x
repls [x:sic, y:date] := if y ∈ dates(x) then π3(hist(x)·y) else {} fi;
po [x:repl] := π2(x);                          -- purchase order responsible for x
reporset [x:sic, y:ponr] := $[t: repls(x)| po(t)=y];
                                               -- set of repl of x due to order y
reporqty [x:sic, y:ponr] := Σ[t: reporset(x,y)| π1(t)];
                                               -- qty replenished due to y
orlines [x:ponr] := π2(pof·x);                 -- order lines in x
items [x:ponr] := dom(orlines(x));             -- items ordered in x
orders [x:sic] := $[t: dom(pof)| x ∈ items(t)];
                                               -- orders for item x
```

```
sup [x:ponr] := π2(π1(pof·x));                -- supplier of order x
date [x:ponr] := π1(π1(pof·x));               -- date of order x
orqty [x:ponr, y: sic] := π2(orlines(x)·y);
                                              -- qty ordered in x of item y
data [x:supnr] := π1(supf·x);                 -- supplier data of x
types [x:supnr] := π2(supf·x);                -- types supplied by x

    -- constraints
--   ∀[x: dom(sif)| type(x) ∈ dom(sitf)];
--   ∀[x: dom(pof)| items(x) ⊂ dom(sif)];
--   ∀[x: dom(supf)| types(x) ⊂ dom(sitf)];
--   ∀[x: dom(pof)| sup(x) ∈ dom(supf)];
--   ∀[x: dom(sif)| ∀[y: dates(x)| ∀[z: repls(x,y)|
--        po(z) ∈ dom(pof) and x ∈ items(po(z)) and
--        date(po(z)) ≤ y and reporqty (x, po(z)) ≤ orqty (po(z), x)]]];
     -- each replenishment of a stock item has a purchase order responsible
     -- for it; in this purchase order, a line must point to the item in
     -- question; the replenishment date cannot exceed the order date and
     -- the replenished quantities due to this order cannot
     -- exceed the number of items ordered.
--   ∀[x: dom(sif)| ∀[y: dates(x)| y ≤ cal]];
--   ∀[x: dom(pof)| date(x) ≤ cal];

     -- system environment
proc report [x:str];     -- trigger to user

    -- auxilliary functions (general)
convstr [x:T] :: str;
fupd [x:T->S, y:T->S] :=
     [u: dom(x) ∪ dom(y)| if u ∈ dom(y) then y·u else x·u fi];
disjdom [x:T->S, y:T->U] := dom(x) ∩ dom(y) = ();
contdom [x:T->S, y:T->U] := dom(x) ⊂ dom(y);

     -- report messages
c1 := 'key conflict in sitf';
c2 := 'key conflict in sif';
c3 := 'key conflict in supf';
c4 := 'key conflict in pof';
u1 := 'undefined key in sitf';
u2 := 'undefined key in sif';
u3 := 'undefined key in supf';
u4 := 'undefined key in pof';
sh := 'the stock level of the specified item is too low';
id := 'illegal date'
ok := 'ok';
s1 := 'the following items have to be ordered: ';

    -- processors
proc AddItemTypes [x: sitc -> sitd] ::=             -- action 1
     if disjdom (x, sitf)
     then sitf <- fupd (sitf, x), report <= ok
     else report <= c1 fi;
proc UpdItemTypes [x: sitc -> sitd] ::=             -- action 2
     if contdom (x, sitf)
     then sitf <- fupd (sitf, x), report <= ok
     else report <= u1 fi;
proc SeeItemTypes ::=                               -- action 3
     report <= convstr (sitf);
```

```
proc AddItems [x: sic -> sidat] ::=                     -- action 4
     if disjdom (x, sif)
     then if π1(rg(x) ∊ dom(sitf)
          then sif <- fupd (sif, [t:dom(x)| «x·t, ()»]), report <= ok
          else report <= u1 fi
     else report <= c2 fi;
proc UpdItem [x: sic, y: sidat] ::=                     -- action 5
     if x ∊ dom(sif)
     then if π1(y) ∊ dom(sitf)
          then sif <- fupd (sif, [t:{x}| «y, hist(t)»]), report <= ok
          else report <= u1 fi
     else report <= u2 fi;
proc SeeItems ::=                                       -- action 6
     report <= convstr ([t: dom(sif)| data(t)]);
proc AddSuppls [x: supnr -> supdat] ::=                 -- action 7
     if disjdom (x, supf)
     then supf <- fupd (supf, [t:dom(x)| «x·t, ()»]), report <= ok
     else report <= c3 fi;
proc UpdSuppl [x: supnr, y: supdat] ::=                 -- action 8
     if x ∊ dom(supf)
     then supf <- fupd (supf, [t:{x}| «y, types(t)»]), report <= ok
     else report <= u3 fi;
proc SeeSuppls ::=                                      -- action 9
     report <= convstr ([t: dom(supf)| data(t)]);
proc AddSupplTypes [x: supnr, y: $sitc] ::=             -- action 10
     if x ∊ dom(supf)
     then if y ⊂ dom(sitf)
          then supf <- fupd (supf, [t:{x}| «data(t), y ∪ types(t)»]),
               report <= ok
          else report <= u1 fi
     else report <= u3 fi;
proc SeeTypeSuppls [x: sitc] ::=                        -- action 11
     report <= convstr ([t: $[s: dom(supf)| x ∊ types(s)] | data(t)]);
proc SeeSupplTypes [x: supnr] ::=                       -- action 12
     report <= convstr ([t: types(x) | sitf·t]);
proc SetDate [x: date] ::=                              -- action 13
     if x > cal then cal <- x, report <= ok else report <= id fi;
proc UpdStock [x: sic, y: $wdr, z: $repl] :=            -- action 14
     if x ∊ dom(sif)
     then if Σ(π1(y)) ≤ curqty(x) + Σ(π1(z))
          then sif <- fupd (sif, [t: {x}|    «data(t),
               fupd (hist(t), [s: {cal}| «curqty(x) - Σ(π1(y)) + Σ(π1(z)),
                                         wdrs(t,s) ∪ y, repls(t,s) ∪ z»])
                                              » ]),
               ShowShortage <= {x}
          else report <= shortage
     else report <= u2 fi;
proc RecordMan [x: sic -> qty] ::=                      -- action 15
     if contdom (x, sif)
     then sif <- fupd (sif, [t: dom(x)| «data(t),
          fupd (hist(t), [s:{cal}| «x·t, wdrs(t,s), repls(t,s)»])»]),
          ShowShortage <= dom(x)
     else report <= u2 fi;
proc SeeSohProfile [x: sic, y: date, z: date] ::=       -- action 16
     report <=
     if x ∊ dom(sif)
     then convstr ([s: $[t: dates(x))| y ≤ t ≤ z]| qty(x,s)])
     else u2 fi;
```

```
proc SeeSohType [x: date, y: sitc] ::=                     -- action 17
     report <= convstr ([t: selcode(y)| qty(t,x)])
     where selcode [x: sitc] := $[t: dom(sif)| type(t) = x] erehw;
proc AddPurchOrder [x: ponr, y: supnr, z: pol] ::=         -- action 18
     if x ∈ dom(pof)
     then report <= c4
     else if y ∈ dom(supf)
          then if contdom (z, sif)
               then pof <- fupd(pof, [t: {x}| ⟪⟪cal,y⟫, z⟫]), report <= ok
               else report <= u2
          else report <= u3 fi fi fi;
proc ShowShortage [x: $sic] :=                             -- action 19
     report <= if shortitems(x) = {} then ok
               else s1 & convstr (shortitems(x)) fi
     where shortitems [x: $sic] := $[t: x| vs(t) ≤ rlev(t)]
     andwh vs [x:sic] :=             -- virtual stock
          curqty (x) + Σ[t: orders(x)| orqty(t,x) - reporqty(x,t)] erehw;
proc DatePurch [x: date] ::=                               -- action 20
     report <= convstr ([t: $[s: dom(pof)| date(s) = x]| pof.t]);
proc NumPurch [x: ponr] ::=                                -- action 21
     report <= if x ∈ dom(pof)
               then convstr (pof·x)
               else u4 fi;
proc WdrStock [x: sic, y: $wdr] ::=                        -- action 22
     UpdStock <= ⟪x, y, {}⟫;
proc SeeWdrProfile [x: sic, y: date, z: date] ::=          -- action 23
     report <= if x ∈ dom(sif)
               then convstr ([s: $[t: dates(x)| y ≤ t ≤ z] | withdrs(x,s)])
               else u2 fi;
proc ReplStock [x: sic, y: $repl] ::=                      -- action 24
     UpdStock <= ⟪x, {}, y⟫;
proc SeeReplProfile [x: sic, y: date, z: date] ::=         -- action 25
     report <= if x ∈ dom(sif)
               then convstr ([s: $[t: dates(x)| y ≤ t ≤ z] | repls(x,s)])
               else u2 fi;
```

References

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

[2] Dietz, J.L.G. and K.M. van Hee
A framwork for modeling of discrete dynamic systems
Proceedings TAIS conference Sophia Antipolis 1987
North-Holland, 1988

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

[4] van Hee, K.M., G.J. Houben, L.J. Somers and M. Voorhoeve
A formal model for system specification
to appear

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

[6] IFIP Working Group 8.1 Conference (Computerized Assistance during the system life cycle)
Example A: Business analysis and system design specifications for an inventory control and purchasing system

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

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

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

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