Two case studies in ExSpect

Citation for published version (APA):

Document status and date:
Published: 01/01/1990

Publisher Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.

Download date: 15. Oct. 2019
Two Case Studies in ExSpect

by

P.A.C. Verkoulen  M.M. de Brouwer

November, 1990
This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology. Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review. Copies of these notes are available from the author or the editor.
Two Case Studies in ExSpect

P.A.C. Verkoulen M.M. de Brouwer

Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513
NL-5600 MB Eindhoven
The Netherlands
E-mail: wsinpet@win.tue.nl

Abstract

The specification method ExSpect consists of a model, a language and a toolbox. It can be used to create executable specifications of distributed (information-) systems.

When describing a system, one is interested in the state space of the system, in the way a state can be transformed and in the interaction of the components of the system. In ExSpect, we have integrated the description of processes, their interaction, and the structuring of objects. This last issue is an advanced way of datamodeling, using a powerful type system.

In this paper, we describe two case studies that have been worked out in ExSpect. The first one is a model of a jobshop system. The second one is a larger specification, dealing with a transport firm.

Keywords: case studies, CASE-tools, formal specifications, prototyping, software engineering.

1 Summary of ExSpect

In this section, we give a short description of ExSpect, which consists of three parts: a model, a language and a computer tool. It resembles Coloured Petri Nets ([8]), but there are some differences. For an extensive treatment, we refer to the ExSpect literature [2, 4, 5, 6, 7, 9].

The model, called DES model\(^1\), consists of two kinds of components, viz. processors and channels, corresponding to transitions and places in Petri Net terminology. Each channel contains at each moment zero or more tokens which must have the type of that channel. Moreover, each token has a time-stamp, denoting the time before which

\(^1\) DES stands for Discrete Event System
it cannot be ‘used’. The state of a system is represented by the bag of all tokens present in the system.

Processors communicate with each other by sending tokens via channels. Each processor has one or more input channels and zero or more output channels. A processor can execute its body (fire) provided that each of its input channels contains at least one consumable token. An assignment of tokens to processors such that each processor can fire is called an event. Processors are eager to start, i.e. they will fire as soon as possible. The (simulated) event time at which an event may occur is the maximum of the time-stamps of the consumed tokens. The transition time of a system in a certain state is the minimum of the event times of all possible events. Being in a certain state, an event will be selected which has an event time which is equal to the transition time of the system. This event will then be executed. This causes a state transition: the consumed tokens are removed from the state and the generated tokens are added to it. This all happens instantaneously. The time-stamps of the produced tokens are equal to the sum of the transition time plus a specified delay. This delay is non-negative, so it is obvious that the transition times of successive events will be ascending.

Processors may execute their bodies simultaneously. When two processors are both able to fire, but they cannot fire together, one of them is enabled non-deterministically.

The ExSpect language consists of two parts, viz. the functional and the dynamic part. The dynamic part makes it possible to define processors, channels, stores and systems. A store is a special kind of channel which contains always exactly one token. Whenever this token is ‘consumed’ by any processor, it is replaced by another one without delay, either explicitly by the processor or implicitly by putting the old one back. A system is an aggregate of processors, channels, stores and subsystems. A system may contain subsystems, which allows for structuring a specification hierarchically, like in [10].

For the dynamic part, we have the following (graphical) conventions: an ExSpect system can be represented by a graph. This graph has two kinds of nodes, viz. nodes representing processors or systems and nodes representing channels or stores. Triangles represent processors and squares represent systems, whereas circles represent channels (empty circle) or stores (circle which contains a dot). Systems can be connected to each other via so-called pins: there are input- and output pins for channels and there are store pins. Graphically, they are denoted by a circle containing a $i$, $o$ or $s$, respectively. Two nodes of the same kind cannot be connected directly to each other.

The functional part is in fact a sugared typed $\lambda$-calculus. It serves to define types and functions. Types are defined by means of type expressions, using some basic types and some type constructors. Functions can be defined using constants and other functions (standard- or user-defined functions). Functions may be recursive. The type system allows for subtyping and polymorphism. When a type $T_1$ is subtype of a type $T_2$ (notation: $T_1 \subset T_2$), this means that the set of objects that have type $T_1$ is a subset of the set of objects having type $T_2$. 
The polymorphism-concept is used when a system, processor or function has to be defined which can be applied to a class of types. E.g., it is possible to define a polymorphic intersection-function which computes the intersection of two sets of the same type\(^2\); a possible definition for this intersection-function is the following one (which is recursive)\(^3\):

```plaintext
intersection [ x: $T, y: $T ] :=
    if x = {} then {}
    else if pick(x) in y
        then ins(pick(x), intersection(rest(x), y))
        else intersection(rest(x), y)
    fi
fi: $T;
```

The basic ExSpect types are void, num, real, bool and str, corresponding to the empty type, the rational numbers, the real numbers, the booleans and the character strings, respectively. The type operators are \( \times \) (cartesian product), \( \$ \) (\( \$T \) is the type of all finite subsets of \( T \)) and finally \( \rightarrow \) (\( A \rightarrow B \) is the type containing all finite functions (mappings in ExSpect terminology) with domain \( A \) and co-domain \( B \)). It is also possible to define labeled tuples [8]. A tuple type expression has the form \([ l_1 : T_1, \ldots, l_n : T_n ]\), where the \( l_i \) are disjoint labels and the \( T_i \) are type expressions. The subtyping rules for tuples have been chosen analogously to those given by Cardelli in [2]. This definition allows for specialisation and inheritance.

The construction of new types is done using a type-from-clause:

```
type \( T_1 \) from \( T_2 \) with Condition
```

means that \( T_1 \) is subtype of \( T_2 \). The with-part is optional; it may be used to express a condition that has to be satisfied by an element of type \( T_2 \) in order to be an element of type \( T_1 \) too.

Because of the polymorphism of the type system, it is possible to make general-purpose libraries containing generic components which can be used to model concrete situations. For example, a toolbox is being developed for modeling logistic systems ([1]).

For now, the state space of a system is modeled using the ExSpect type system. This is powerful, but it lacks overview and support when specifying complex systems. Therefore, we are working on giving more support to modeling the state space. A first approximation of a solution is described in [9].

---

\(^2\)Actually, they only have to have a so-called common supertype (cf. [3, 8])

\(^3\)In the example, three basic functions are used: the ins-function inserts an element into a set, the pick-function gives an arbitrary but fixed element of a non-empty set and the rest-function is defined by the equation \( \text{rest}(x) \cup \{ \text{pick}(x) \} = x \), for each non-empty set \( x \).
2 Case 1: The Jobshop System

2.1 Introduction

We will now present a practical example of the use of ExSpect in specifying complex systems. We will model a factory which produces five different but similar products. It is a jobshop, i.e. no stocks of ready-made products are kept; the manufacturing process does not start until a client places an order.

We want to specify this system formally and use this specification for simulation purposes to analyze the performance of the factory and to check whether there is a bottleneck at one of the machines in the factory.

2.2 Informal description

The products are made in four phases, which can each be one of three types of processes, which we will call type A, B and C processes. For every product type the sequence of processes to manufacture it is prescribed. The sequences of processing phases for each job type are given in the following table:

<table>
<thead>
<tr>
<th>product-type</th>
<th>manufacturing-process</th>
</tr>
</thead>
<tbody>
<tr>
<td>type 1</td>
<td>A B C B</td>
</tr>
<tr>
<td>type 2</td>
<td>B C A C</td>
</tr>
<tr>
<td>type 3</td>
<td>C B C A</td>
</tr>
<tr>
<td>type 4</td>
<td>A C A B</td>
</tr>
<tr>
<td>type 5</td>
<td>B C A C</td>
</tr>
</tbody>
</table>

Figure 1: The processing phases of the jobs

The factory has six different machines available on the workfloor. Machines 4 through 6 can perform type A, B and C processes, respectively. Machines 1, 2 and 3 can switch between type A/B, A/C and B/C processes, but changing the machine mode takes some time.

The partially completed jobs are equally divided among the machines that are configured for the required processing type using a round robin scheduling mechanism. Every machine has its own FIFO queue to store incoming jobs while it is not available. So, whether or not a machine is a bottleneck of the manufacturing process can easily be observed by looking at the average queue lengths of all machines. If one of them is clearly much longer than the other ones, we have discovered a bottleneck. Also, we could measure the occupation rates of the machines to find out whether some machine is constantly at work while others are doing nothing. By measuring the time that elapses between placing an order and the final delivery of the completed product we can measure the performance of the factory, for every job type separately.

Another factor in the production process is that the machines break down from time
to time. When this happens the defect must be repaired, which takes some time. A machine is only discovered to be defective when it should be processing a job; i.e., no defects are discovered while the machine is idle. The time a machine is in repair is not taken as working time in the calculation of the occupation rate for that machine. The exact processing times for each machine/job combination and other technical details are beyond the scope of this text.

2.3 Specification

In this section we will be specifying the jobshop in terms of ExSpect systems, processors, stores and channels.

At the highest level of abstraction, the factory consists of six processing unit systems which include the modeling of the queues, the actual job processing and the modeling of failures. Moreover, there is a dispatcher system which determines where incoming jobs are to be sent for further processing. This dispatcher also takes care of the scheduling. When a unit has finished a job, this job will be sent back to the dispatcher.

Furthermore, a system has to be developed to model the customer demand. This system, called the generator, simulates a Poisson process to create random new jobs and sends them to the dispatcher (cf. figure 4). Finally, there is a manager system to allow the user to change the modes of the various machines.

2.3.1 The processing unit

A processing unit consists of three parts (which will be worked out in the next subsections): a queue, the actual machine, and a failure generator. Jobs enter the unit and arrive at the queue. When the machine is idle, the job is sent to the machine, otherwise it is stored inside the queue system. The machine does the actual processing on the incoming job, which takes a fixed amount of time. The machines are the only places in the entire factory where something is actually done with jobs other than just moving them around. While processing a job, a machine may get a failure signal from its fail system. Failures that occur during the processing of a certain job will suspend the processing of the job until the machine is repaired. When the processing is finished the resulting product is sent back to the dispatcher to see if it needs further processing. At the top level, the specification is as follows (cf. figure 2):

```
sys Unit[ in entry:job, out toDispatch:job,
          val unitid:num, timetable:str -> (job -> real), ir:real ]
:=
  channel mEntry:job,
  channel mSignal:$void,
  channel mFail:real,
  Queue(in entry, mSignal, out mEntry, val unitid),
  Machine(in mEntry, mFail, out mSignal, toDispatch,
           val unitid, timetable),
  Fail(out mFail, val ir);
```
2.3.2 The dispatcher system

The dispatcher unit consists of two stores and one processor. One store contains the modes of the machines while the other one contains the current schedule. A schedule consists of a table mapping the three processing types \((A, B, C)\) onto machine numbers. When a job enters the dispatcher, the processor knows if it is ready to go to the exit or what processing it needs next. If it needs further processing, the processor determines the machine scheduled for the required processing type, sends the job to the correct unit and changes the schedule for that processing type using the information in the store containing the modes of the machines. At the top level of its specification, the dispatcher looks like (cf. figure 3):

```plaintext
sys Dispatcher[ in incoming:job, out toM1:job, toM2:job,
  toM3:job, toM4:job, toM5:job,
  toM6:job, toExit:job ]
:=
store schedule : str -> num init { <<'A',4>>, <<'B',5>>, <<'C',6>> },
store modes : num -> str init initmodes,
Dispatch(in incoming, out toM1, toM2, toM3, toM4, toM5, toM6, toExit, store modes, schedule);
```

2.3.3 The generator system

The generator system is a very simple system which consists of one channel and one processor which triggers itself using the channel. The trigger on the channel contains a seed value for the built-in random function. When the processor fires, it picks a
random job type to send to the dispatcher. A new seed value is computed and put on the internal channel with a delay picked by an exponentially distributed random generator.

2.3.4 The manager system

The manager system creates a special reconfiguring job at the user's request. Such a job is automatically routed to the correct unit by the dispatcher. It is treated just like any other job. This means that it may take some time for the change to take place, because it has to 'stand in line' at the machine's queue. When a reconfiguring job passes through the dispatcher, the schedule is adapted immediately.
2.3.5 The queue system

The queue system, which is a part of the processing unit, consists of two processors and two stores. The first store contains a truthvalue and records whether the connected machine is idle or not. The other store contains the jobs waiting for processing. The first processor receives jobs from the dispatcher. When a job enters the queue system, this processor checks whether the machine is idle, and if so, it sends the job directly to the machine, simultaneously setting the idle store to false. If the machine is not idle, it stores the job in a FIFO datastructure in the other store.

The second processor handles the signals coming from the machine. If such a signal arrives, this means that the machine has become available for a new job. When jobs are waiting in the FIFO datastructure, the processor immediately takes out the first one and sends it to the machine; if the queue is empty, this processor sets the idle store to true. The top level of the queue specification looks as follows (cf. figure 6):

```
sys Queue[ in entry:job, t:$void, out exit:job, val id:num ] :=
    store buffer : num -> job init {},
    store idle : bool init true,
    AcceptJob(in entry, out exit, store buffer, idle),
    GetNextJob(in t, out exit, store buffer, idle);
```

In ExSpect, processors are (together with stores and channels) the basic building bricks of any system specification. They are described using a part of the ExSpect specification language called statements which resemble a programming language. For example, the processors AcceptJob and GetNextJob of the queue system are specified by:

```
proc AcceptJob[ in a:job, out x:job,
    store queue:num -> job, idle:bool ] :=
    if idle
        then idle <- false, x <- a
    else queue <- FIFO(a,queue)
fi;
```
which means as much as: 'depending on whether the store idle contains true or false, change the store to false and send the job to the machine or store the job in a waiting list'. The FIFO function makes sure the new job is stored at the end of the waiting list, so the oldest job is always removed first by GetNextJob, which is specified by:

\[
\text{proc GetNextJob[ in } t:\$void, \text{ out } x:\text{job}, \text{ idle:bool } ] \\
\quad := \\
\quad \text{if } \#\text{queue}=0 \\
\quad \quad \text{then } \text{idle } \leftarrow \text{true} \\
\quad \quad \text{else } x \leftarrow \text{top(queue)}, \text{ queue } \leftarrow \text{dequeue(queue)} \\
\quad \text{fi;}
\]

which means: 'depending on whether there are job waiting in the store or not, set the store idle to true or send the first job in the waiting list (top(queue)) to the machine, removing it from the waiting list (dequeue(queue))'.

2.3.6 The fail system

The fail system works similar to the generator system (cf. figure 7). The only difference is the StartUp processor. StartUp takes the token in start and puts it with a random delay into internal. Without this construction, all fail systems would simultaneously generate a failure-signal when the factory is started. This would cause all machines to malfunction immediately.
2.3.7 The machine system

The machine system is a very complex system, consisting of three processors (Start, Finish and Error), two stores (table and down) and one channel (internal). Jobs entering the machine arrive at processor Start. Start looks in the store table for the processing time for the current job and puts it on the internal channel with a corresponding delay. While the processing time is ticking away, processor Error accumulates incoming failure signals in the store down. When the processing time for the current job has elapsed, Finish can pick it off the channel, change the job and send it with the delay found in down back to the dispatcher. Also, a signal is given to the queue with the same delay and down is reset to zero. When the job was really a reconfiguring job, processor Finish does not send anything back to the dispatcher, but instead puts the correct processing time-table for the new mode in the store used by Start. The top level of the machine specification looks like:

```plaintext
sys Machine[ in entry:job, st:real, out sig:$void, exit:job, 
   val id:num, timetable:str -> (job -> real) ] :=
channel internal:job,
store down: real init 0.0,
store table: job -> real init timetable.(initmodes.id),
Start(in entry, out internal, store table),
Finish(in internal, out exit, sig, store down, table),
Error(in st, store down);
```

2.3.8 The complete system

We have now specified all the components of the jobshop. How these components have been joined in order to obtain the jobshop system can be seen in figure 9. For analysis purposes we have added some components. In section 2.4, something will be mentioned about these components, but they will not be worked out completely, as this falls outside the scope of this paper.
2.4 Analysis facilities

In this section we will describe the extensions of the specification from the previous section which enable us to monitor the factory while the simulation is running. An extra entry is made to the manager system, to allow the user to bring extra jobs into the system. By initializing the new channel with triggers having specific timestamps, scenarios can be tried on the factory.

Also, stores have been added for the queue lengths, average production time and machine occupation rates. The queue lengths are maintained by the queue systems and the occupation rates are maintained by the machine systems, but they need some extra internal administration. The average production time is determined by a new system called exit, which looks at the time-of-entry tag on the finished products that are received from the dispatcher. To make this possible, the type job has been extended with a field containing the time-of-entry for jobs. In the implementation, jobs are merely 2-digit numbers, of which the first digit indicates what kind of job we are dealing with and the second digit indicates how many processing phases this job has already passed through. Machine mode switching jobs are represented by negative numbers. Several other systems are added for the periodical reporting purposes.

2.5 Simulation

With the simulator module of ExSpect, we can simulate the behaviour of the jobshop we have specified above. We have added some analysis facilities to the specification, e.g. a system which reports the current average walk-through time of the jobs every 200 time units. These reports have been linked to the so-called bargraph application, which gives a graphical representation of these results: the graph displays the average
Figure 9: The Main System
walk-through time (vertically) belonging to each time period (horizontally). Figure 10 shows the normal scenario, i.e. what happens when no machine switches and when no user jobs are entered through the Manager system.

Observe that the graph is fairly steady, except for a little bump around period 20. It is likely that this is due to unfortunate malfunctioning of machine 6, which causes long queues and therefore larger walk-through times.

Another experiment is to switch machine 2 into mode A at time point 3000 (period 15). It is to be expected that machine 6 will get more work than it can handle then, since all jobs requiring type C processing are then routed to machine 6. At time point 4000 (period 20), we have switched machine 2 back to mode C again. As we can see from figure 11, it still takes some time before machine 6 has recovered from all the jobs that were waiting in its queue.

The large walk-through times reported after period 20 are due to all those jobs that had to stand in line at machine 6. They had to wait for all their predecessors to be processed, so they were in the factory for quite a while. During the simulation run, queue lengths of up to 26 could be observed at machine 6. Of course, the effect of switching machine 3 (type B) into type C would be far less dramatic, as type C processing is required more often than type B processing (cf. figure 1). Experiments have confirmed this.
Figure 11: The results when machine 2 is in A mode for some time

You should be aware of the different scaling on the vertical axes when comparing the results from figures 10 and 11. The reason for this difference is simple: the bargraph application tries to use a window optimally. This means here that the maximal walk-through time on the vertical axis of figure 10 is 61, whereas it is 140 in figure 11. So there is really a dramatic increase of the walk-through times in the second case: they are about 4 times as high when machine 2 is in A-mode for some time.
3 Case 2: The Transport Company XXX

In the Netherlands, five research centres cooperate in the so-called ISDF-project. In this project, a specification problem has been formulated in order to serve as a case study and as a large example throughout the research. In this section, we will describe this case and give a possible solution using ExSpect. We will not deal with all the aspects of the specification, but we will go into enough detail to give the reader a sufficient insight in how ExSpect can be used for specifying large systems.

3.1 An Informal Description of the Case

In this subsection, we will give an informal description of the case to be solved. The organisation to be modeled is a transport firm.

The primary function of this firm is the distribution and temporary storage of goods. For the transport of the goods, the firm has a number of different trucks at its disposal; for the temporary storage of the goods, the firm possesses a number of depots, with different configurations and capacities.

The company needs different kinds of trucks as there are different kinds of cargo, viz.
1. Two kinds of solid goods, viz. plain goods, like boxes, and refrigerated goods.
2. Two kinds of liquid goods, viz. dangerous liquids like acid, and harmless liquids, like milk.

Because of the variety of goods, the firm possesses three kinds of trucks, viz. plain trucks, refrigerated trucks and tankers. The trucks for transportation of liquids have to be cleaned after they have been used, as liquids are not wrapped up. An exception to this rule is raised when the new cargo is the same as the previous one. Cleaning a tanker takes about half an hour when the cargo was not dangerous and an hour otherwise. A truck may contain several cargos of possibly different clients, provided that the cargos do not interfere with each other (e.g. two different kinds of fluids).

Each depot can contain three kinds of stores: plain stores, cold-stores and tanks. The tanks must again be cleaned between storing two different fluids. This takes about two hours for dangerous goods and about one hour otherwise. Of course, depots may contain several cargos of possibly different clients. Finally, in general the depot has several stores, so it is possible to store different kinds of goods in one depot at the same time.

The transport firm consists of an administration department, a planning department and an execution department. The administration registers all kinds of information, e.g. the arrival of a client order or the absence of a driver. The planning department checks whether an order can be accepted and schedules it for execution if this is the case. The execution department takes care of the actual execution of the order. Each depot has a local administration which registers the goods which are present at the depot or which are to be expected in the future.

---

4 ISDF stands for Information System Description Formalism. It aims at defining an integrated framework for specifying Information Systems and analyzing those specifications. ExSpect serves as a prototype framework in this project.
Our firm works according to the following global procedure:

1. A client contacts the administration and places an order. If the client is solvent, the order is preliminarily accepted. Otherwise, the order is postponed until the client has made an initial payment. Moreover, the administration performs a rough check whether the capacity is sufficient for accepting the order.

2. If the order is preliminarily accepted by the administration, the planning department checks whether there is enough capacity to accept the order definitively. This means that there has to be sufficient truck capacity and sufficient depot capacity. The planning department has to take into account that the transport may take several days and that a cargo can be stored temporarily in one or more transit depots before it reaches its final destination. The planning department sends a message to the client, confirming the acceptance of the order or rejecting the order because of insufficient capacity. Of course, the planning department tries to make such a schedule that the costs of executing an order are minimalized.

3. Each 24 hours, the planning department sends a plan to the execution department. The execution department modifies the plan by taking into account absent drivers and trucks which are out of order. Because of this, it is possible that the plans cannot be executed completely. Therefore, the execution department sends a message at the end of each day to the planning department saying which plans have been realized and which have not.

4. The planning department also sends messages to the depots about which trucks are to be expected. When making the plans, the planning department has to account for the capacity of the depots, the number of people working at the depots, the opening- and closing times, etc.

5. When an order has been realized, the execution department sends a message to the administration. The administration sends a bill to the client and stores the order as being finished.

3.1.1 Minor Constraints and Assumptions

We have just seen the general constraints which the transport firm has to fulfill. There are also some minor constraints. Moreover, we have made certain assumptions when making the specification. All this is mentioned in this section; we will start by treating the minor constraints and thereafter deal with the assumptions we have made.

Minor Constraints

1. An order has to be executed on the day that has been agreed upon with the client.
2. Roads can be blocked.
3. After a month, a truck is in repair for some time to be checked.
4. Drivers work 40 hours a week. They may not work more than 8 hours a day and they have to rest after this for at least 6 hours. When it is absolutely necessary that a driver works longer, it is allowed. However, the company has to pay him extra then, so we want to avoid this as much as possible.
5. A driver has to be authorized to drive the trucks he does. This also holds for transporting dangerous goods. Not all roads may be used to transport dangerous goods.
6. We want to minimalize the costs, so we want trucks to drive as less as possible and have a cargo as large as possible.

Assumptions
1. An order can be contained by one truck. If a client wants to place larger orders, he has to split them himself. (It would also be possible to adapt the specification such that orders are split when they enter the administration, when necessary).
2. The planning department works on a daily basis, not on an hour basis.
3. An order has to placed at least one day before its execution date.
4. Data about absent drivers or trucks which are out of order come from outside the system; they fall outside the scope of this specification.
5. We assume that the depots are open 24 hours a day. If necessary, the specification can be changed in a relative easy way to overcome this assumption.
6. We do not want to specify a planning algorithm. So we did not work out a planner, but we assume that the routes come from outside and are stored in a table somewhere in our company. These routes take into account the aforementioned constraints about minimalizing truck kilometers and roads which may not be used for transporting dangerous goods.

3.2 The ExSpect specification

In this section, we will give a top-down description of our ExSpect specification. The description is top-down, because the specification has been developed this way; it illustrates the way you would usually build an ExSpect specification.

On the highest level of abstraction, the world to be modeled consists of three parts (cf. figure 12), viz. the company itself, the roadnet which contains the depots of the company and the roads connecting the depots, and the outside world, which contains the clients who generate orders.

As the outside world contains the clients placing orders, it has been modeled as a system that generates random orders and communicates with the company about these orders. The roadnet will be described later; we will first develop the company specification.

The company, from now on indicated as XXX, receives orders and payments from outside and generates messages for its clients. It sends questions and plans to the depots in the roadnet and receives answers from these depots. The goods which have reached their destination depot leave the roadnet over the channel outerworld.

The company again consists of three subsystems, viz. the administration, the planning department and the execution department (cf. figure 13). The administration and the planning department together constitute the company's office.

The administration receives orders and payments from the clients and it sends messages to the clients. Accepted orders are sent to the planning department, which adapts the plans on basis of a received order. The planning department sends plans and questions to the depots and concept plans to the execution department. This department sends final plans to the depots and collects information about finalized orders from the depots. This information is being sent to the planning department and to the administration, which sends a bill to the client.
Figure 12: The Main System

Figure 13: The transport company XXX
Let's work out the administration now (cf. figure 14). Orders come in and are judged in the system `receiveorder`. If the company has globally sufficient capacity and the client is solvent, then the order is accepted immediately. Otherwise, a message is being sent to the client, stating that the company cannot handle the order or that the client has to pay first. In the latter case, the order is postponed and will be dealt with as soon as a payment has arrived. When an order is accepted, this is registered in the store `beingtreat` and the order is sent to the planning department. Upon getting the message that the order has been finished completely, it will be removed from this store by the processor `dispatchorder`.

![Figure 14: The administration of XXX](image)

The next part of the firm we are going to work out, is the planning department (cf. figure 15). This system consists of three subsystems, viz. `checkcapacity` which has to check whether there is enough truck- and depot capacity to deal with a certain order, `makeplan`, which makes the plans and sends them to the depots and to the execution department. The third system in planning is `reportfinish`. This system sends messages to the administration about finalized orders.

For checking the capacity, `checkcapacity` has to be able to send questions to the depots. These questions are sent via the output channel `question`, whereas answers arrive on the input channel `answer`.

If we look into more detail into the `checkcapacity` system, we see four processors and one subsystem (cf. figure 16). Upon arrival of an order, the first processor, `fixroutes`, computes all possible routes for it and sends them to `searchroute`. This processor formulates questions for one of these routes and then waits until answers have arrived. The questions are distributed over the depots by `putquestions`, whereas the answers are processed by the system `processanswers`. If this gives a positive answer, `searchroute` sends this investigated route to `send2plan`. If the final answer is negative, another route is tried, provided that there is one. If there is no alternative left, a message is sent that the order has to be rejected after all, due to a lack of resources. In case of a positive answer, the order and the final route are sent to the `makeplan` system, which is the second constituent of the planning department. The `makeplan` system (cf. figure 17) consists of three processors and two internal stores. The store `prelimresult`
contains the preliminary plans, which have not been sent to the depots yet. The store occupation records the actual occupation of the resources of XXX. Upon arrival of an order and a route, computeplan adapts those two stores to cover this order. Upon arrival of a message about a finalized order, the processor regrelease registers that the relevant resources are free again. Finally, the processor passplan sends each 24 hours a conceptplan to the execution department and plans to the depots, based upon the contents of prelimresult. Upon transmitting these plans, passplan also empties prelimresult.

The third constituent system of the planning department is reportfinish. This system sends a message to makeplan and to the administration upon arrival of a finalized order.

The last component of the company is the execution department. It modifies incoming concept plans, based upon data about absent drivers and trucks which are out of order. Moreover, it reports every 24 hours to the planning about finalized orders.

We now want to say something more about the definition of a depot (cf. figure 18). A depot consists of three parts. The main part is the physical system, which receives trucks and stores the cargo and the truck for a while. The second part is the one which receives the questions and plans for the depot and sends them to the third part, the depot administration (depotadm).

When a truck arrives at a depot\(^5\) the processor receivecargo sends a message to the depot administration. Moreover, it is checked whether the truck only passes by or whether it has to stay for some time at the depot. In the latter case, the truck and its cargo are stored in the relevant storehouse and garage. In the other case, the truck is passed on to the outside immediately.

\(^5\) There are depots with one and with two entrances/exits. Here you see a picture of a depot with two entrances (ride1 and ride2) and two exits (nextride1 and nextride2). This means that two roads can be connected to this depot.
Figure 16: The checkcapacity system
Figure 17: The system makeplan

Figure 18: A depot
When a stored cargo has to leave the depot again, the system `sendcargo` loads it onto a truck and sends it to the roadnet. This is also reported to the depot administration.

The depot administration (cf. figure 19), contains two stores expected and present, respectively. In these stores, information is kept about the cargos which are in the depot at the moment or are to be expected in the future. Moreover, there are four processors: `storeplan` registers incoming plans and modifies expected. The processor `storearrival` registers the presence of a cargo at the depot, whereas `storedeparture` registers the departure of such a cargo. Finally, `reply` gives answers to incoming questions, based upon the data in the two stores.

![Figure 19: The depot administration](image-url)
References


Proceedings of the 14th Symposium on Operations Research, 6-8 September 1989, University of Ulm, Ulm.


---

*Chapter 2 is an extension of ExSpect, the Functional Part*
<table>
<thead>
<tr>
<th>No.</th>
<th>Author(s)</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>85/01</td>
<td>R.H. Mak</td>
<td>The formal specification and derivation of CMOS-circuits.</td>
</tr>
<tr>
<td>85/02</td>
<td>W.M.C.J. van Overveld</td>
<td>On arithmetic operations with M-out-of-N-codes.</td>
</tr>
<tr>
<td>85/03</td>
<td>W.J.M. Lemmens</td>
<td>Use of a computer for evaluation of flow films.</td>
</tr>
<tr>
<td>85/04</td>
<td>T. Verhoeff, H.M.L.J. Schols</td>
<td>Delay insensitive directed trace structures satisfy the foam the foam rubber wrapper postulate.</td>
</tr>
<tr>
<td>86/01</td>
<td>R. Koymans</td>
<td>Specifying message passing and real-time systems.</td>
</tr>
<tr>
<td>86/02</td>
<td>G.A. Bussing, K.M. van Hee, M. Voorhoeve</td>
<td>ELISA, A language for formal specification of information systems.</td>
</tr>
<tr>
<td>86/03</td>
<td>Rob Hoogerwoord</td>
<td>Some reflections on the implementation of trace structures.</td>
</tr>
<tr>
<td>86/04</td>
<td>G.J. Houben, J. Paredaens, K.M. van Hee</td>
<td>The partition of an information system in several systems.</td>
</tr>
<tr>
<td>86/05</td>
<td>J.L.G. Dietz, K.M. van Hee</td>
<td>A framework for the conceptual modeling of discrete dynamic systems.</td>
</tr>
<tr>
<td>86/06</td>
<td>Tom Verhoeff</td>
<td>Nondeterminism and divergence created by concealment in CSP.</td>
</tr>
<tr>
<td>86/07</td>
<td>R. Gerth, L. Shira</td>
<td>On proving communication closedness of distributed layers.</td>
</tr>
<tr>
<td>86/09</td>
<td>C. Huizing, R. Gerth, W.P. de Roever</td>
<td>Full abstraction of a real-time denotational semantics for an OCCAM-like language.</td>
</tr>
<tr>
<td>86/10</td>
<td>J. Hooman</td>
<td>A compositional proof theory for real-time distributed message passing.</td>
</tr>
<tr>
<td>86/11</td>
<td>W.P. de Roever</td>
<td>Questions to Robin Milner - A responder's commentary (IFIP86).</td>
</tr>
<tr>
<td>86/12</td>
<td>A. Boucher, R. Gerth</td>
<td>A timed failures model for extended communicating processes.</td>
</tr>
<tr>
<td>86/13</td>
<td>R. Gerth, W.P. de Roever</td>
<td>Proving monitors revisited: a first step towards object oriented systems (Fund. Informatica</td>
</tr>
</tbody>
</table>
Specifying passing systems requires extending temporal logic.

On the existence of sound and complete axiomatizations of the monitor concept.

Federatieve Databases.

A formal approach to distributed information systems.

Delay-insensitive codes - An overview.

Enforcing non-determinism via linear time temporal logic specification.

Temporele logica specificatie van message passing en real-time systemen (in Dutch).

Specifying message passing and real-time systems with real-time temporal logic.

The maximum number of states after projection.

Language extensions to study structures for raster graphics.

Three families of maximally nondeterministic automata.

Eldorado ins and outs. Specifications of a data base management toolkit according to the functional model.

OR and AI approaches to decision support systems.

Playing with patterns - searching for strings.

A compositional proof system for an occam-like real-time language.

A compositional semantics for statecharts.

Normal forms for a class of formulas.

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

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

88/09 A.T.M. Aerts
K.M. van Hee  A Tutorial for Data Modelling.

88/10 J.C. Ebergen  A Formal Approach to Designing Delay Insensitive Circuits.

88/11 G.J. Houben
J. Paredaens  A graphical interface formalism: specifying nested relational databases.

88/12 A.E. Eiben  Abstract theory of planning.

88/13 A. Bijlsma  A unified approach to sequences, bags, and trees.

88/14 H.M.M. ten Eikelder
R.H. Mak  Language theory of a lambda-calculus with recursive types.
<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>88/15</td>
<td>R. Bos, C. Hemerik</td>
<td>An introduction to the category theoretic solution of recursive domain equations.</td>
</tr>
<tr>
<td>88/16</td>
<td>C. Hemerik, J.P. Katoen</td>
<td>Bottom-up tree acceptors.</td>
</tr>
<tr>
<td>88/17</td>
<td>K.M. van Hee, G.J. Houben, L.J. Somers, M. Voorhoeve</td>
<td>Executable specifications for discrete event systems.</td>
</tr>
<tr>
<td>88/18</td>
<td>K.M. van Hee, P.M.P. Rambags</td>
<td>Discrete event systems: concepts and basic results.</td>
</tr>
<tr>
<td>88/19</td>
<td>D.K. Hammer, K.M. van Hee</td>
<td>Fasering en documentatie in software engineering.</td>
</tr>
<tr>
<td>88/20</td>
<td>K.M. van Hee, L. Somers, M. Voorhoeve</td>
<td>EXSPECT, the functional part.</td>
</tr>
<tr>
<td>89/1</td>
<td>E.Zs.Lepoeter-Molnar</td>
<td>Reconstruction of a 3-D surface from its normal vectors.</td>
</tr>
<tr>
<td>89/2</td>
<td>R.H. Mak, P. Struik</td>
<td>A systolic design for dynamic programming.</td>
</tr>
<tr>
<td>89/3</td>
<td>H.M.M. Ten Eikelder, C. Hemerik</td>
<td>Some category theoretical properties related to a model for a polymorphic lambda-calculus.</td>
</tr>
<tr>
<td>89/4</td>
<td>J. Zwiers, W.P. de Roever</td>
<td>Compositionality and modularity in process specification and design: A trace-state based approach.</td>
</tr>
<tr>
<td>89/5</td>
<td>Wei Chen, T. Verhoeff, J.T. Udding</td>
<td>Networks of Communicating Processes and their (De-)Composition.</td>
</tr>
<tr>
<td>89/6</td>
<td>T. Verhoeff</td>
<td>Characterizations of Delay-Insensitive Communication Protocols.</td>
</tr>
<tr>
<td>89/7</td>
<td>P. Struik</td>
<td>A systematic design of a parallel program for Dirichlet convolution.</td>
</tr>
<tr>
<td>89/9</td>
<td>K.M. van Hee, P.M.P. Rambags</td>
<td>Discrete event systems: Dynamic versus static topology.</td>
</tr>
<tr>
<td>89/10</td>
<td>S. Ramesh</td>
<td>A new efficient implementation of CSP with output guards.</td>
</tr>
<tr>
<td>89/11</td>
<td>S. Ramesh</td>
<td>Algebraic specification and implementation of infinite processes.</td>
</tr>
<tr>
<td>89/12</td>
<td>A.T.M. Aerts, K.M. van Hee</td>
<td>A concise formal framework for data modeling.</td>
</tr>
<tr>
<td>Volume/Issue</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>--------------</td>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>89/14</td>
<td>H.C. Haesen</td>
<td>ELDA, data manipulatie taal.</td>
</tr>
<tr>
<td>89/15</td>
<td>J.S.C.P. van der Woude</td>
<td>Optimal segmentations.</td>
</tr>
<tr>
<td>89/16</td>
<td>A.T.M. Aerts, K.M. van Hee</td>
<td>Towards a framework for comparing data models.</td>
</tr>
<tr>
<td>89/17</td>
<td>M.J. van Diepen, K.M. van Hee</td>
<td>A formal semantics for Z and the link between Z and the relational algebra.</td>
</tr>
<tr>
<td>90/2</td>
<td>K.M. van Hee, P.M.P. Rambags</td>
<td>Dynamic process creation in high-level Petri nets, pp. 19.</td>
</tr>
<tr>
<td>90/4</td>
<td>A. Peeters</td>
<td>Decomposition of delay-insensitive circuits, p. 25.</td>
</tr>
<tr>
<td>90/8</td>
<td>M.B. Josephs</td>
<td>Receptive Process Theory, p. 16.</td>
</tr>
<tr>
<td>90/9</td>
<td>A.T.M. Aerts, P.M.E. De Bra, K.M. van Hee</td>
<td>Combining the functional and the relational model, p. 15.</td>
</tr>
<tr>
<td>90/10</td>
<td>M.J. van Diepen, K.M. van Hee</td>
<td>A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).</td>
</tr>
<tr>
<td>90/11</td>
<td>P. America, F.S. de Boer</td>
<td>A proof system for process creation, p. 84.</td>
</tr>
<tr>
<td>90/12</td>
<td>P. America, F.S. de Boer</td>
<td>A proof theory for a sequential version of POOL, p. 110.</td>
</tr>
<tr>
<td>90/14</td>
<td>F.S. de Boer</td>
<td>A proof system for the language POOL, p. 70.</td>
</tr>
<tr>
<td>90/15</td>
<td>F.S. de Boer</td>
<td>Compositionality in the temporal logic of concurrent systems, p. 17.</td>
</tr>
<tr>
<td>Year</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>90/16</td>
<td>F.S. de Boer, C. Palamidessi</td>
<td>A fully abstract model for concurrent logic languages</td>
</tr>
<tr>
<td>90/17</td>
<td>F.S. de Boer, C. Palamidessi</td>
<td>On the asynchronous nature of communication in concurrent logic languages: a fully abstract model based on sequences</td>
</tr>
<tr>
<td>90/19</td>
<td>M.M. de Brouwer, P.A.C. Verkoulen</td>
<td>Two Case Studies in ExSpect</td>
</tr>
<tr>
<td>90/20</td>
<td>M. Rem</td>
<td>The Nature of Delay-Insensitive Computing</td>
</tr>
<tr>
<td>90/21</td>
<td>K.M. van Hee, P.A.C. Verkoulen</td>
<td>Data, Process and Behaviour Modelling in an integrated specification framework</td>
</tr>
</tbody>
</table>