The modelling and analysis of queueing systems with QNM-ExSpect

Eindhoven University of Technology

Department of Mathematics and Computing Science

The Modelling and Analysis of Queueing Systems with QNM-ExSpect

by

W.M.P. van der Aalst

91/33

Computing Science Note 91/33
Eindhoven, December 1991

COMPUTING SCIENCE NOTES

Copies can be ordered from:
Mrs. F. van Neerven
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

The Modelling and Analysis of Queueing Systems with QNM-ExSpect

W.M.P. van der Aalst
Eindhoven University of Technology
Dept. of Mathematics and Computing Science

Abstract

In this paper the application of ExSpect to Queueing Networks (QN) is described. ExSpect is a simulation tool based on a formal framework closely related to Coloured Petri Nets (CPN). ExSpect supports hierarchical models and user defined modules. The fact that systems can be combined into larger systems is a very powerful feature and allows a structured top-down design. The module concept combined with the possibility to customise a system encourages the reuse of already specified components and the creation of 'toolboxes'. One of those toolboxes is the *QNM* toolbox, developed to analyse queueing networks. The design interface of ExSpect permits the user to define the queueing network in a graphical manner and automatically generates a simulation model. If simulation takes too long to produce reliable results one can decide to use an analytical model. On certain conditions, it is possible to generate the network structure and parameters for such an analytical model from the description in QNM-ExSpect. In this paper we show how to translate a model in QNM-ExSpect to a *BCMP* network.

Keywords: Simulation tools, High level Petri nets, Queueing networks.

1 Introduction

In the last twenty years, queueing networks have become popular in the field of performance analysis of computer systems, communication networks and production systems. A common feature of all of these systems is the fact that there is a limited resource which must be shared among a number of competing customers who require service. Examples of typical shared resources are CPUs, memory, I/O devices, transport aids or machines.

Since these resources are limited, customers may have to wait. These waiting customers form a queue in front of the shared resource. This is the reason these systems are called queueing systems. In other words: a queueing system is a network of queues and servers containing a number of customers circulating in the network. For example, a computer system can be seen as a network of queues and servers (CPU's,terminals) and a collection of customers (clients,tasks). Another example is a production system with a set of jobs needing some service performed by machines, vehicles and humans.
There exist two approaches for the analysis of queueing networks.
The most flexible and easy to use method is simulation. Simulation can be applied in many situations and, by nature, it enables the modelling and analysis of systems which are mathematically intractable.
Simulation is not the only way to analyse queueing systems, "pure" queueing systems also allow for analytical methods. In fact the main reason for which queueing networks have become so popular is due to the *product form* solution property that holds for a fairly large class of queueing networks (see [8]). The shortcomings of these analytical methods are mainly due to their lack of descriptive power in presence of phenomena such as synchronisation, blocking and the splitting of customers because they destroy the existence of a product form solution.

We propose a mixed approach. We have developed a "toolbox" containing building blocks. These building blocks allow for the modelling of a large class of queueing networks, in a graphical manner. The software package ExSpect automatically generates a simulation model allowing for all kinds of measurements. On certain conditions, it is possible to translate such a model into a BCMP network ([4]). Such a network can be analysed using analytical techniques. Even when these conditions are violated, the simulation results are still useful to obtain parameters for an approximated BCMP network or to compare these results with the results of an analytical technique.

Our approach is based on a formal framework which makes it easy to model complex computer and production systems. This framework has been developed at Eindhoven University of Technology ([5]). Every complex system that fits into this framework is called a *discrete event system* (DES). The framework is related to Coloured Petri Nets ([7]), therefore we can use Petri net theory to verify structural properties ([9]). To model dynamic systems we have extended Coloured Petri nets with a time concept ([5]).
A DES consists of two kinds of components: *processors* and *channels* (corresponding to respectively transitions and places in Petri nets).
A processor is connected to one or more input channels and zero or more output channels. To each input channel of a processor a certain (positive integer) weight is attached (in most cases weights are equal to 1). This allows a channel to be a multiple input channel of a processor. With each channel a type is associated and with each processor a function. The signature of the function of a processor p corresponds with the types and weights of input channels of p and the types of its output channels.
Channels may be shared by several processors as input or output channel. At any moment

channels may contain *tokens*. A token has a *value* that belongs to the type of the channel and a *time stamp*. There may be more tokens in the same channel with the same value and time stamp. So a channel actually contains a bag of tokens over its type.
At any moment a *transition* may occur, which means that the configuration of tokens, called the *state*, changes (our terminology attaches another meaning to the term transition than Petri net theory). Such a transition occurs instantaneously and is *executed* by the processors. A processor is *enabled* if it is able to select the right number of tokens from each of its input channels. The number of tokens to be selected from an input channel by a processor p is equal to the weight of the input channel for p. Only enabled processors may execute. The execution or firing of a processor means that the selected tokens are consumed (deleted) and that new tokens are produced for the output channels of the processor. The number of tokens produced and the values of these tokens depend on the function associated to the processor that fires. Note that a token can be consumed only once.
An *event* is a set of simultaneous processor executions. The *event time* at which an event may occur is the maximum of the time stamps of the tokens to be consumed. The *transition time* of a system in a certain state is the minimum of the possible event times. Being in a certain state, a system will select an event of which the event time is the transition time and execute it, causing a state transition. The time stamps of produced tokens will be at least equal to the event time. It is thus clear that the transition times of successive events will be non-descending.

In this paper we start with a short introduction to the language ExSpect, which allows us to specify a DES. This language is supported by a software package also called ExSpect. This software package contains a number of tools: a graphical design interface, a type checker, a run-time interface and an analysis tool. This way it possible to perform all kinds of analysis, for example simulation or Petri net based analysis ([1]).
In section 3 we describe the *Queueing Network Module*, a toolbox build on top of ExSpect. The design interface of ExSpect permits the user to define the queueing network in a graphical manner and automatically generates a simulation model. This approach is not new, many authors propose graphical special purpose simulation tools. Consider for example the Queueing Model Generator, described in [10].
There are, however, some important differences between these graphical special purpose simulation tools and QNM-ExSpect. With our approach it is possible to define new (user defined) building blocks. Because of the hierarchy construct, called system, it is possible to create new building blocks by composing existing ones. The hierarchy construct also allows for the structuring of large complex models. Because our approach is based on Petri nets, we provide a number of formal analysis techniques based on Petri net theory. Finally, we facilitate the transformation of a simulation model into an queueing model allowing for analytical analysis methods. If simulation takes too long to produce reliable results one can decide to use an analytical model. It is possible to generate the network structure and parameters for such an analytical model from the description in QNM-ExSpect. In section 4 we show how to translate a model in QNM-ExSpect to a *BCMP* network ([4]).

Finally, we give an example that illustrates our approach.

2 ExSpect

At Eindhoven University of Technology the language ExSpect, has been developed for the specification of discrete event systems. Since it is executable, it can be used for prototyping and simulation.
A typechecker and an interpreter have been implemented. The typechecker checks the specification for correctness, completeness and consistency. The interpreter can be used to execute such a specification.
The user can view or influence a running simulation via the ExSpect (run time) user interface. This interface is asynchronous, which means that the user and the running simulation model do not have to wait for each other. Another attractive feature is the possibility to synchronise the (scaled) simulation time with the real time. These two options make real-time interactive simulation possible.
The software is written in the language C and runs under UNIX at a SUN workstation in a window environment. Also a stripped version of the software is available for the personal computer (PC). More information about the ExSpect system is presented in [5] and [3].
The language ExSpect consists of two parts: a functional part and a dynamic part. The functional part is used to define types and functions. The type system consists of some primitive types and a few type constructors to define new types. A 'sugared lambda calculus' is used to define new functions from a set of primitive functions. ExSpect is a 'strongly typed' language since it allows all type checking to be done statically. A strong point of the language is the concept of type variables: it provides the possibility of polymorphic functions. A more formal discussion regarding the functional part of ExSpect is given in [6].
The dynamic part of ExSpect is used to specify the network of processors, channels and stores and therefore the interaction structure. A store is a special kind of channel: it always contains precisely one token. The behaviour of a processor is described by functions.
ExSpect has five primitive types: `void`, `bool`, `num`, `real` and `str`. The type constructors are set (`$`), cartesian product (`><`) and mapping (`->`). From a set of types and type operators we can form type expressions that symbolize new (composite) types. We can attach names to type expressions, thus defining new types. The following example illustrates some type definitions:

```
    type operation from str;
    type load from real with [x] x > 0.;
    type job from operation >< load;
    type process_id from num;
    type processor_status from process_id -> job;
```

Note that we can add a `with` part to restrict a type.

Likewise we can construct new functions. Our set of basic functions includes all well-known set-theoretical, logical and numerical constants and functions. Because of some 'sugaring' it is possible to write these functions in their usual symbolic infix or 'circumfix' notation. As an example we show two function definitions operating on the types defined above:

```
p_t := { << 14220 , << 'disk_io', 0.1256 >> >>,
         << 14223 , << 'swapper', 0.1348 >> >>,
         << 14227 , << 'mail', 0.0673 >> >>,
         << 14097 , << 'rpc', 0.0089 >> >>,
         << 10034 , << 'deamon', 0.0056 >> >>,
         << 14090 , << '-shell', 0.1562 >> >>,
         << 10235 , << 'disk_io', 0.2288 >> >>
       }: processor_status;

processor_load[ x:processor_status]
:= if dom(x) = {}
     then 0.
     else pi2(pi2(pick(x))) + processor_load(frest(x))
   fi : real;
```

The functions pi1, pi2 (projections), dom, pick and frest (respectively domain, taking and deleting from a mapping) are examples of basic functions. The result of the function application processor_load(p_t) equals 0.7272 .
It is also possible to define polymorphic functions using *type* variables:

```
range[ x:T -> S ]
:= if dom(x) = {}
     then {}
     else ins(pi2(pick(x)),range(frest(x)))
   fi : T -> S;
```

T and S are type variables, therefore the **range** function can be used to calculate the range of *any* mapping. The range function uses recursion and the ins (inserting into a set) function to calculate the result.
Processor definitions are split in a header and contents part. The header part (sometimes called signature) contains the processor name, its interaction structure and its parameters. The interaction structure is given by (possibly empty) lists of input channels, output channels and stores. The contents part consists of concurrent (conditional) assignments of expressions to output channels and stores. Consider the following processor definition:

```
proc bus [ in send:T, out receive:T , val d:time ]
:=
receive <- send delay d;
```

This processor models the transportation of a job via a bus as a simple delay. The time between the moment the job is sent and the moment the job is received is set by a value parameter `val d`. If the keyword `delay` is omited, a delay of zero is assumed. Note that `T` is a type variable, this makes the processor `bus` polymorphic. The only restriction is that the type of the channel connected to `send` "matches" the type of the channel connected to `receive`.
A *store* is a special kind of channel: it always contains precisely one token. One can think of a store as a variable. We define a *system* as an aggregate of processors, connected by channels and stores. A system can also contain other (sub) systems. This way it is possible to make hierarchical models, which is extremely useful when specifying large and complex processes. Moreover, the system concept allows both bottom-up and top-down design.
If a system has no interaction with its environment we call it a closed system else an open system. Open systems communicate with the outside world via input and output channels and stores. Therefore, a system definition consists of a header similar to a processor header and a contents part. A system can have value, function, processor and even system parameters. So it its possible to define generic systems. This way a system can be customised or fine-tuned to a specific situation. The contents part is a list of all the objects (processors, systems and local stores and channels) in the system and their relations.
As an example we show a simple system to model a central processor unit (CPU) (see figure 1). Note that we represent a processor by a triangle and a channel by a circle. The cpu system has one input channel (`injob`) to accept incoming jobs and one output channel (`outjob`) to release finished jobs. The ExSpect specification of this system is shown in the box.

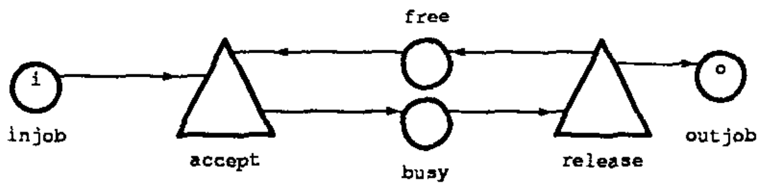


Figure 1: The cpu system

The cpu system has a one function (`fun`) parameter which is used to calculate the time needed to perform a certain operation. Processor `accept` is activated if there is at least one job and the CPU is free. The CPU is free if there is a token available in the channel `free`. If `accept` is activated the operation corresponding to the token consumed from `injob` is executed. This is represented by the production of a token on the internal channel `busy` with a delay specified by the function parameter `processingtime`. Now the CPU is occupied until the processor `release` is activated.
As we saw there are four kind of definitions; type, function, processor and system definitions. These definitions are stored in modules. Definitions become visible outside a module

```
sys Machine [ in injob : job,
              out outjob : job,
              fun processingtime[x: job]: time
            ]
:=
channel free : signal init signal,
channel busy : job,
accept(in injob, free, out busy),
release(in busy, out outjob, free)
where
  proc accept [ in injob : job, free : signal,
                out busy : job
        ]
  :=
  busy <- injob delay processingtime(injob);

  proc release [ in busy : job,
                 out outjob : job, free : signal
        ]
  :=
  free <- signal,
  outjob <- busy;
end;
```

if they are preceded by the keyword `export`. The typechecker creates a header file containing the declarations of exported definitions. You can use these definitions by including this header file, this way it is possible to reuse definitions easily. If you make a module with type definitions and operations acting on these types then the internal representation of these types remains hidden in the module. A module hides the format of a particular data structure. You cannot access the information by directly manipulating the module's data structure. This way ExSpect enables information hiding.
The fact that systems can be combined into larger systems is a very powerful feature and allows a structured top-down design. The module concept combined with the possibility to customise a system encourages the reuse of already specified components and the development of 'toolboxes'.

3 The Queueing Network Module

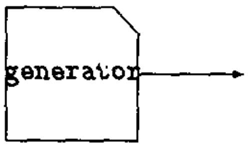
We have used ExSpect to specify a broad class of systems; protocols, information systems, logistic systems and flexible manufacturing systems. The application of ExSpect to logistic systems is described in [3], the application to flexible manufacturing systems is described in [2].

Queueing networks have proved to be useful in practical situations. Therefore we have created the Queueing Network Module (QNM). This is a part of the environment of the simulation language ExSpect. The combination of ExSpect and QNM is called *QNM-ExSpect.*
One can think of QNM as an *interface* on top of ExSpect. This interface prevents the user familiar with queueing networks from having to learn a new formalism. It fully utilises the features of the language ExSpect such as polymorphism, value and function parameters

and information hiding via the module concept.
The Queueing Network Module consists of a number of building blocks such as a server, queue, generator, .. . These building blocks are generic ExSpect systems (see section 2). Each system has a number of parameters. If an input channel or an output channel represents the flow of customers (called entities), then the type is described by a type variable (for instance T, S or U). This allows the user to define the relevant attributes of the entities. If (s)he does not want to bother about this, (s)he should include the default entity type client.

3.1 Building blocks

3.1.1 Generator

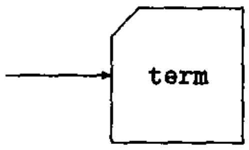


The **generator** generates the entities (customers, clients) of the model. These may be customers entering a bank, jobs arriving at a production unit, or a request for some processing power generated by a terminal in a computer network. An entity may have user-declared attributes; all ExSpect types are permitted. Every generator has a name. The time between arrivals is specified by a probability distribution. It is also possible to specify the number of entities generated each time. The value of the entity generated may depend upon the name of the generator, the number of entities already generated and the actual generation time.

```
sys generator[
      out o:T,
      val name:str, seed:real, nofbatches:num, batchsize:num,
      fun interarrivaltime[r:real]:real, value[name:str,n:num,t:real]:T,
          class[x:T]:class
          ];
```

The box shows the signature of the **generator** system. There is one output channel o whose type is polymorphic. The name of the generator and the initial seed are specified via the value parameters **name** and **seed**. The value parameters **nofbatches** and **batchsize** are used to indicate the total number of arrivals and the number of entities generated per arrival respectively. The first arrival is always at time 0.00 . Set **nofbatches** to 1 if you want to generate *closed clients*. Closed clients are entities which never leave the system, i.e. a closed client visits one or more servers infinitely often. Set **nofbatches** to **INF** for the unlimited generation of *open clients*. An open client enters the system, visits one or more servers and then leaves the system. The function parameter **interarrivaltime** is used to calculate the time between two arrivals. This interarrival time may depend upon **r**, a random number generated by the random generator inside the **generator** system. Such a random number is a real between 0 and 1. The value of an entity generated by the system is specified by the function parameter **value**. The result type of this function has to be of the same type as the type of the output channel. Note that the result may depend upon the name of the generator (**name**), the number of entities already generated (**n**) and the actual generation time (**t**). The final function parameter **class** is only used for reporting. This function assigns a customer to a specific class. All results (see section 3.3) are expressed in terms of these classes, for example the number of customers generated in each class.

3.1.2 Terminal block

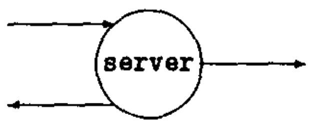


If an entity enters the system called **term** it disappears. In other words: the entity leaves the system. Note that the *scope* of the queuing system is given by the location of **generator** and **term** blocks. Remember, *closed clients* never leave the system.

```
sys term[
        in i:T,
        val name:str,
        fun rep[x:T,t:real]:real,
            class[x:T]:class
        ] ;
```

The **term** system has one input channel **i** of an arbitrary type. The name is defined by the value parameter **name**. Function parameter **rep** can be used to customise the measurements, for example to measure the throughputtimes of the entities. The value of **rep** may depend upon the value of the arriving entity **x** and the present time. The result is always a **real**, use the *bargraph* program to observe these measurements. The function parameter **class** is used for reporting. This function assigns an entity (customer) to a specific class. All results are expressed in terms of these classes, for example the throughput times of the entities.

3.1.3 Server

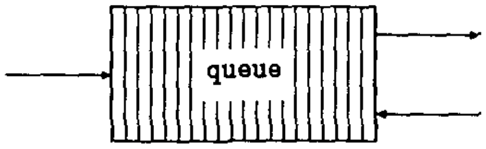


The entities in the network travel from server to server until they leave the system. At each server they offer a certain amount of work (the *workload*) and they wait until the server has completed the service. One can think of a server as service point or a workstation. A server is always connected to a queue (sometimes indirectly via an **assemble** and/or an **assign** system. If the server is free and there are entities waiting to be processed by this server then the **server** system starts serving this job. The service time is given by a probability distribution which may depend upon the value of the entity. One can also use the **server** system to model a number of identical parallel servers or an infinite server.

```
sys server[
        in i:S,
        out o:T, sig:signal,
        val name:str, seed:real, nofservers:num,
        fun servicetime[x:S,r:real]:real, transform[x:S]:T,
            class1[x:S]:class, class2[x:T]:class
          ] ;
```

The **server** system has an input channel i and an output channel o both with a polymorphic type. These are used to model the arrival and departure of clients. There is also an output channel, called sig, that is used to inform the preceding **queue** system (or **assign** or **assemble** system) that the server is ready to process another client. The output channel sig is of type **signal**, a predefined type with only one element, also called **signal**. The value parameter **name** is used to specify the name of the system and **seed** is used to set the random generator. The number of parallel servers inside the **server** system can be specified via **nofservers**, this value parameter is set to INF to model an infinite server. The service time of an entity is given by the function parameter **servicetime** which may depend upon the value of the entity (x) and a random number (r). Note that the service time may be fixed, calculated by an expression or random with a particular probability distribution. Since, a service can change the attributes of a client, a function parameter called **transform** is supplied. The input of this function is the value of the arriving entity (x), the output is the value of the processed entity. Note that the result type of this function and the type of the output channel o have to "match". The function parameters class1 and class2 are used to report the changing of classes and the servicetimes per class.

3.1.4 Queue

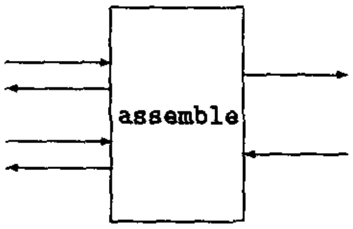


A queue is used to store the entities waiting to be served. The order in which the entities leave the queue is defined by the *service discipline*, for example FIFO (first-in-first-out), LIFO (last-in-first-out) or SIRO (select-in-random-order). A `queue` system is always followed by a `server`, `assemble` or `assign` system.

```
sys queue[
        in i:T, sig:signal,
        out o:T,
        val name:str, seed:real,
        fun discipline[n:num,x:T,r:real]:real,
            class[x:T]:class
         ] ;
```

The `queue` system has an input channel `i` and an output channel `o` both of a polymorphic type. These are used to model the arrival and departure of clients in a queue. The input channel receives entities from `generator`, `server` and `selector` systems. There is also an input channel called `sig` used by a `server`, `assemble` or `assign` system to send a message to tell the queue that it is ready to accept new clients. If the queue contains entities (customers,clients) and there is a token in the input channel `sig`, then an entity is selected and sent to the `server`, `assemble` or `assign` system. The name of the queue is specified by a value parameter called `name`. The function parameter `discipline` is used to specify the service discipline; it may depend upon the arrivalnumber (`n`), the value of the client (`x`) and a random number (`r`). The function returns a real value for every queued entity, the queue always selects the entity with the *highest* value to leave the queue. If there are multiple entities having the same value, assigned by the function parameter discipline, then the order in which they are selected is not defined.

3.1.5 Assemble

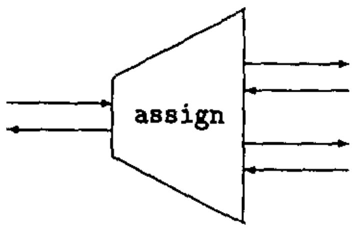


The **assemble** system is placed between two **queue** systems and a **server** (or **assign**) system. It denotes an operation in which two entities (parts) are assembled into a new entity.

```
sys assemble[
        in i1:S, i2:T, sigin:signal,
        out o:U, sigout1:signal, sigout2:signal,
        fun transform[x:S,y:T]:U
            ] ;
```

One queue is connected to the input channel `i1` and the output channel **sigout1**, the other queue is connected to the input channel `i2` and the output channel **sigout2**. The remaining channels (`o` and **sigin**) are connected to a **server** or **assign** system. It is also possible to connect these channels to another **assemble** system to model the assembly of three or more entities. There is one function parameter **transform** to calculate the value of the new entity. Note that the parameter types of this function correspond to the types of the input channels `i1` and `i2`. The result type equals the type of the output channel `o`.

3.1.6 Assign

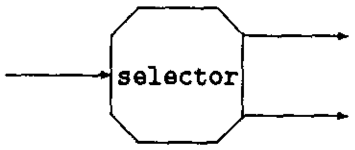


The **assign** system is placed between a **queue** or **assemble** system and two **server** systems. If both servers are able to accept new clients, then the server to process the next client on, is chosen by probability. If only one of the two servers is free then the free server is selected.

```
sys assign[
        in i:T, sigin1:signal, sigin2:signal,
        out o1:T, o2:T, sigout:signal,
        val seed:real, prob1:real, prob2:real,
            class[x:T]:class
          ] ;
```

One server is connected to the input channel sigin1 and the output channel o1, the other server is connected to the input channel sigin2 and the output channel o2. The remaining channels (i and sigout) are connected to a queue or assemble system. It is also possible to connect two assign systems to each other, this way it is possible to model the random selection of three or more (free) servers. The value parameter seed is used to set the random generator, the two other value parameters are used to specify the probability of selecting the first or the second server. The first server is selected with probability prob1 / (prob1 + prob2) the second with probability prob2 / (prob1 + prob2). Remember that if only one of the servers is free these probabilities do not apply.

3.1.7 Selector



The selector system has one input channel and two output channels. For every arriving entity an output channel is chosen by probability or by the value of the entity.

```
sys selector[
        in i:T,
        out o1:T, o2:T,
        val name:str,seed:real,
        fun select[x:T,r:real]:num,
            class[x:T]:class,
            ] ;
```

The selector system has one input channel i and two output channels o1 and o2 of the same polymorphic type. A generator or server system may be connected to i. The output channels can be connected to queue or term systems. It is also possible to connect two selector systems to each other to model the selection between three of more systems. The value parameter seed is used to set the random generator. The function parameter select is used to select the output channel, if the result is 1 then o1 is selected if the result is 2 then o2 is selected. Note that select may depend upon the value of the entity (x) and a random number (r). The final function parameter class is only used to report the routing per class. The function assigns a customer to a specific class.

3.2 The standard entity type: `client`

There is a predefined type called `client`, in many cases it is very convenient to use this type for the channels connecting the building blocks (except for the channels of type `signal`). The best way to do this is to copy the contents of the file `client.ex` into your module, because this file also contains some default function definitions which can easily be adapted.

A value of type `client` has three attributes; the name of the generator, an identification number and the time of creation. Some functions are supplied to access these attributes (see box on page 15). The default function definitions are straightforward. Three standard service disciplines are supplied; FIFO, LIFO and SIRO.

3.3 Measurements

While simulating the model the runtime interface of ExSpect reports:

- the current location of the entities (customers) in the queueing system
- the current state of the servers
- the current state of the queues
- the average (and variance in) waiting time for each (customer) class for each queue
- the average (and variance in) service time for each (customer) class for each server
- the average (and variance in) queue lengths
- the average (and variance in) throughput time

There is also a *trace* option which allows for analysis by a statistical package or an analytical queueing model. In the next section we will discuss this subject.

Compared to other simulation tools using QNM-ExSpect has a number of advantages. For standard applications you do not have to program. It is very easy to customise the components and measurements. It is also possible to build your own components (systems) using the hierarchy construct (system). You can use your own predefined `client` type with a number of personal attributes. Finally the tool allows you to import and export data.

```
-- module CLIENT.EX
include 'basic.h';
include 'utils.h';
include 'stat.h';
include 'qn.h';

-- CLIENT
export export type client from (str >< num) >< real;

export clienttype[ x : client ] :=
   pi1(pi1(x)) : str;

export clientid[ x : client ] :=
   pi2(pi1(x)) : num;

export clienttime[ x : client ] :=
   pi2(x) : real;

--- DEFAULTS
----- USED IN GENERATOR
--------- INTERARRIVALTIME
export interarrivaltime[ r : real ] :=
   r : real;

--------- VALUE
export value[ name : str, n : num, t : real ] :=
   <<<<name,n>>,t>> : client;

----- USED IN TERM
--------- REP
export rep[ x : client, t : real ] :=
   t-clienttime(x) : real;

----- USED IN SERVER
--------- SERVICETIME
export servicetime[ x : client, r : real ] :=
   r : real;

--------- TRANSFORM
export transform[ x : client ] :=
   x : client;

----- USED IN QUEUE
--------- DISCIPLINE
export discipline[ n : num, x : client, r : real ] :=
   -(real(n)) : real;

export FIFO[ n : num, x : client, r : real ] :=
   -(real(n)) : real;

export LIFO[ n : num, x : client, r : real ] :=
   real(n) : real;

export SIRO[ n : num, x : client, r : real ] :=
   r : real;

----- USED IN ASSEMBLE
--------- TRANSFORM
export transform[ x : client, y : client ] :=
   x : client;

----- USED IN SELECTOR
--------- SELECT
export select[ x : client, r : real ] :=
   if r < 0.5 then 1 else 2 fi : num;
```

4 If simulation is not efficient enough

One of the logical consequences of the offered flexibility is a reduced performance. This will be improved by programming the standard module QNM in a conventional programming language (C, PASCAL). If simulation still takes too long to produce reliable results one can decide to use an analytical model.
In the development of an analytical model it is often necessary to use a higher level of abstraction. To be able to solve the model it is also necessary to impose some constraints on the network structure.
Nearly all analytical models allowing for exact analysis are based on queueing networks with a *product form solution.* This means that the steady-state probability distribution can be found by studying the individual queues of the network. See [8] for an introduction to these queueing models.
In 1975 Basket, Chandy, Muntz and Palacios ([4]) showed that mixtures of open and closed networks possess the desired product form solution. Furthermore a number of service disciplines where shown to allow non-exponentially distributed service times, while still conserving the product form solution. In this paper we restrict ourselves to this class of queueing networks, commonly referred to as the class of *BCMP* networks. This is one of the largest (known) classes of queueing networks having a product form solution and therefore allowing for an efficient computation of performance indices.

A BCMP network contains an arbitrary but finite number (M) of service stations. There is an arbitrary but finite number R of customer classes. Customers travel through the network and change class according to transition probabilities. Thus a customer of class r who completes service at service centre i will next require service at centre j in class s with a probability denoted $P_{i,r;j,s}$. The transition matrix $P = [P_{i,r;j,s}]$ can be considered as defining a Markov chain whose states are labeled by the pairs (i,r). The network may be closed with respect to some classes of customers and open with respect to other classes of customers. For open or mixed networks the arrival process (Poisson) may depend upon the state. A service station will be referred to as type 1,2,3 or 4 according to which condition it satisfies.

Condition 1 The service discipline is first-come-first-served (FCFS or FIFO); all customers have the same service time distribution at this service center, and the service time distribution is negative exponential. The service rate may depend upon the number of customers at the center. An exponential FCFS single job class service center with more than one server is equivalent to a similar service center with one server and suitably chosen service rates depending on the number of customers at the server.

Condition 2 There is a single server at a service center, the service discipline is processor sharing, and each class of customers may have a distinct service time distribution. The service time distributions have rational Laplace transforms.

Condition 3 The number of servers in the service center is greater or equal to the maximum number of customers queued at this center in a feasible state, and each class of customers may have a distinct service time distribution. The service time distributions have rational Laplace transforms.

Condition 4 There is a single server at the service center, the service discipline is preemptive-resume last-come-first-served (LCFS or LIFO), and each class of customers may have a distinct service time distribution. The service time distributions have rational Laplace transforms.

In terms of our building blocks this means that we have to impose some constraints on the network structure. Only the building blocks `generator`, `term`, `server`, `queue` and `selector` are allowed. A service station in a BCMP network corresponds to the combination of one `queue` and one `server` system. This combination has to correspond to a service station of type 1 or type 3. Because there is a 1 to 1 relation between servers and queues in a BCMP network we supply 4 service stations corresponding to one of the conditions. This way it is also possible to use the results for processor sharing and preemptive-resume last-come-first-served service centers.

If we only use these components, satisfying the conditions just described, and the number of different customers is small then a direct translation to a BCMP network is possible. If the number of different entities (customers) is to large to be handled by an analytical model, a direct translation to a BCMP network is not possible. In this case we define some coherent customer classes. Every customer (entity) is assigned to a specific class (use the `class` parameters). A small simulation gives us insight in the transition matrix P by observing the trace files (trace option). The simulation also provides information about the distribution of the service time per customer class. If this situation satisfies the conditions, then we have obtained the parameters of the corresponding BCMP network.

We use the trace option to estimate the the distribution of service times and the matrix P. The QNM module produces 4 kinds of files: *arrival files*, *service files*, *selector files* and *termination files*.
During a simulation all generated entities (customers) are written to the arrival file, in the following format: CLASS, GENERATION TIME. All completed services are written to the service file: CLASS, SERVER NAME, NEW CLASS, SERVICE TIME. Every time a `selector` system routes an entity to the next queue it reports: CLASS, SELECTOR NAME, PORT NUMBER. Finally the `term` system registers every arriving entity as follows: CLASS, TERM NAME, RESULT OF `rep`.
It is easy to verify that these files contain sufficient information to obtain the parameters of a BCMP queueing network.

If a BCMP assumption is violated then we adapt the original network such that it becomes a BCMP network. Now we simulate this adapted queueing network and compare the results with the results of the original one. If the differences are small we can interpret the results for the BCMP network as an approximation. This way it is possible to estimate the effect of non-standard queueing disciplines, general service distributions and blocking.

There are also a number of software tools to approximate performance measures for a larger class of queueing networks. An example of such a tool is the *Queueing Network Analyser* ([11]). Most of these tools allow for non-Poisson arrival processes and non-exponential distributed service times. The general approach is to approximately characterise the arrival processes and service time distributions by the first two moments. Note that the trace files generated by QNM-ExSpect contain enough information to supply the required parameters. The Queueing Network Analyser (QNA) also has an option to allow the creating and combining of customers at the stations (following the completion of service).

5 An example

To demonstrate the approach presented in this paper, we present an example which shows the application of QNM-ExSpect to a jobshop producing rolled products. The jobshop receives iron bars from a blast-furnace plant. The jobshop transforms these bars into steel plates using rolling mills to flatten the iron bars and cutting machines. This transformation process takes a number of steps. The sequence of operations transforming an iron bar into a finished product is called a job:

```
type product from str;
type operation from str;
type date from real;
type duration from real;
type job from product ><                              -- product code
              (num -> operation >< duration) ><       -- operation sequence
              date ><                                 -- start date
              date;                                   -- due date
```

Instead of the default type `client` we use the `job` type. A job has four attributes; a product code, a sequence of operations, a start date and a due date. The product code specifies the type of product that has to be produced. The sequence of operations represents the (ordered) set of operations that have to be performed before the product is ready. For every operation we specify the estimated processing time. The start date is the date the job has been released. The due date represents the date the product has to be available.

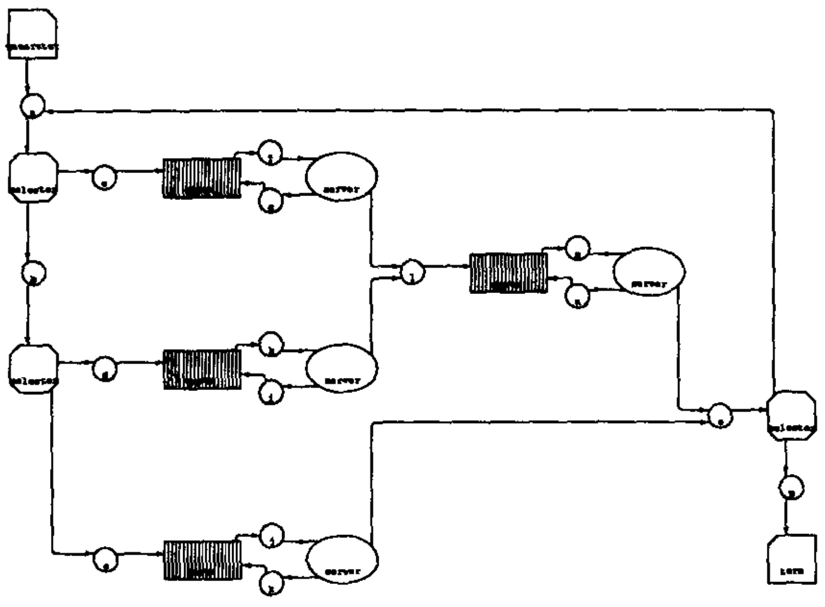


Figure 2: A jobshop modelled with QNM-ExSpect

A value of type job is shown below:

job					
product	operation seq.			date	date
	num	operation	duration		
'AA34234'	1	'weldingA436'	1.2	28.11	5.12
	2	'weldingB476'	1.6		
	3	'cuttingC132'	0.2		
	4	'weldingB462'	2.0		
	5	'cuttingC773'	0.4		

The jobshop described in this example has two rolling machines; a "two-high rolling mill" and a "universal rolling mill". For convenience, we will call these machines machine A and machine B. Every rolling operation is assigned to precisely one rolling mill, i.e. operations are machine specific. There is also one universal cutting machine (machine C). A rolling operation is always followed by a lubrication operation, performed by machine D. This machine applies a lubricant to make the product smooth.

Figure 2 shows the corresponding queueing network in terms of the QNM building blocks. Every **server** system corresponds to a machine. The service time distribution at a server depends on the type of operation. The selector system takes care of the routing of jobs. The service discipline of the cutting machine is fist-in-fist-out (FIFO). The two rolling mills have a queueing discipline to minimise the lateness of jobs. This service discipline is called **EarliestDueDate**: jobs with the earliest due date are selected first.

```
EarliestDueDate[ n :  num, x :  job, r :  real ]
:= - pi2(x) :  real;
```

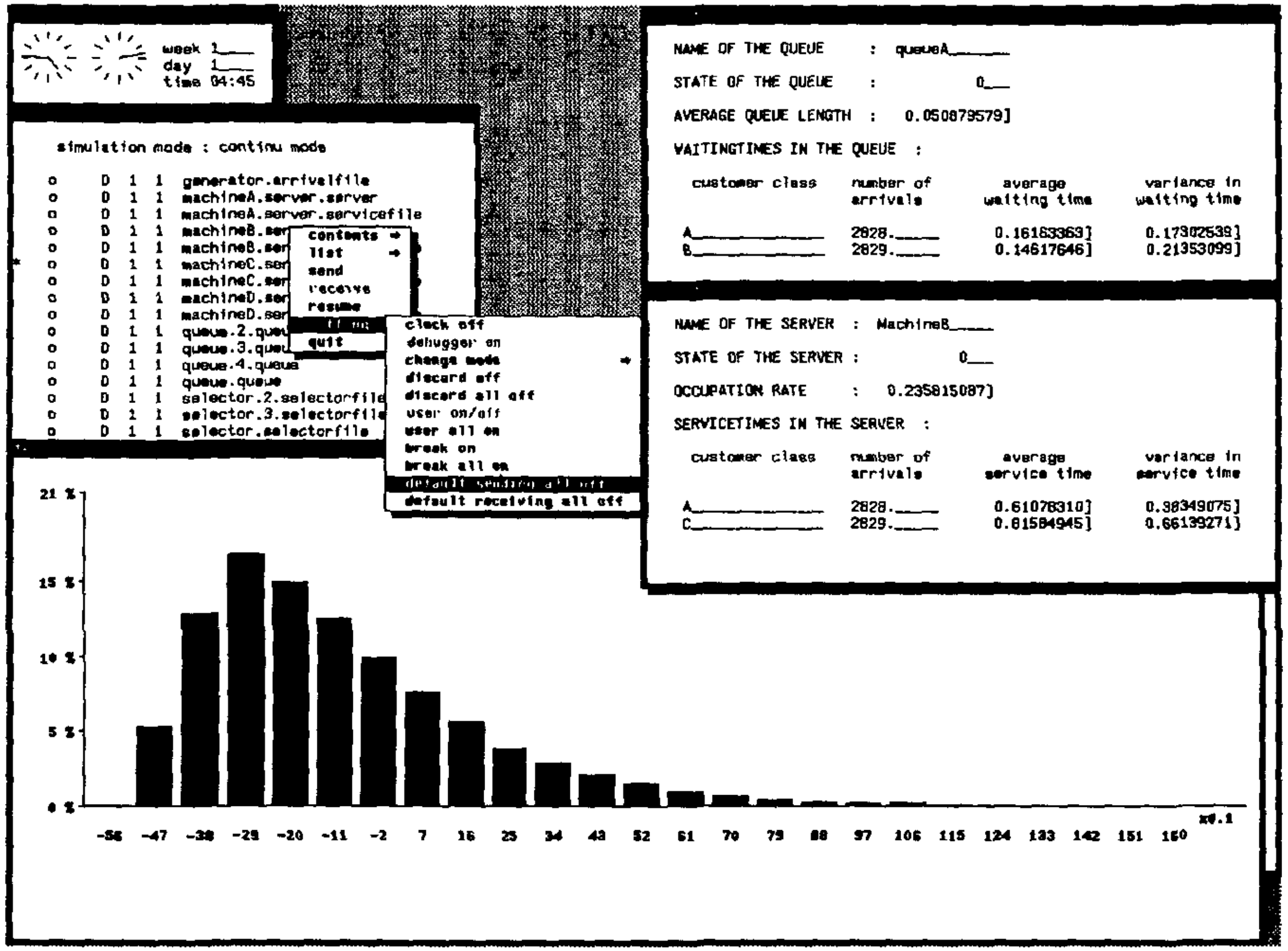
Note that jobs are selected in descending order of their due date (`pi2(x)`). Machine D uses priority scheduling, jobs are discriminated by the machine they come from. Products coming from machine A have priority over products coming from machine B, because they tend to be voluminous. Products coming from the same machine are serviced in FIFO order.
All results are expressed in terms of so-called classes. Jobs (products) are partitioned into three classes depending on the first character of the product code:

```
class[ x :  job ]
:= head(pi1(pi1(pi1(x)))) :  class;
```

The graphical representation (figure 2) of the jobshop was created with de *design interface* of ExSpect. The structure of the model is defined in a totally graphical way. This only takes a few minutes. To feed the model with parameters (distributions, queueing disciplines, etc.) also takes a few minutes. Then the model is ready to be simulated. The *runtime interface* allows the user to observe a running simulation. The performance measures defined in section 3.3 are reported in various windows. It is also possible to export all kinds of data to a statistical package or presentation software.
We have simulated this jobshop under various loads. Figure 3 shows a running simulation. We also compared these simulation results with results obtained using approximated analytical queueing models. The parameters of such an analytical model are based on a simulation run. The accuracy of the results obtained using approximated analytical queueing models depends on the example and the load, therefore it is hard to make general statements.
For more information on the modelling of jobshops with ExSpect, we refer to [2].

6 Concluding remarks

The DES formalism together with the ExSpect language and software offer a complete set of tools for the design, specification and simulation of complex systems.
The modelling effort depends on the availability of generic systems in standard modules. We aim at a "80/20"-situation, where 80 percent of the components needed are already available in standard modules and take up only 20 percent of your time. But the 20 percent you have to create yourself take up 80 percent of your time.
In case of queueing systems we have created a "100/100"-situation; all components needed are already available in the QNM module. This module fully exploits the graphical capabilities of ExSpect. The approach presented combines the advantages of a *simulation package* (focused on a limited field of applications) and a *simulation language* (flexible,



```
week 1___
day  1___
time 04:45
```

```
simulation mode : continu mode

o     D  1  1  generator.arrivalfile
o     D  1  1  machineA.server.server
o     D  1  1  machineA.server.servicefile
o     D  1  1  machineB.ser
o     D  1  1  machineB.ser
o     D  1  1  machineC.ser
o     D  1  1  machineC.ser
o     D  1  1  machineD.ser
o     D  1  1  machineD.ser
o     D  1  1  queue.2.queu
o     D  1  1  queue.3.queu
o     D  1  1  queue.4.queue
o     D  1  1  queue.queue
o     D  1  1  selector.2.selectorfile
o     D  1  1  selector.3.selectorfile
o     D  1  1  selector.selectorfile
```

```
NAME OF THE QUEUE      :  queueA______

STATE OF THE QUEUE     :            0__

AVERAGE QUEUE LENGTH   :   0.050879579]

VAITINGTIMES IN THE QUEUE  :

  customer class    number of      average        variance in
                    arrivals       waiting time   waiting time

  A_____________    2828.____      0.16163363]    0.17302539]
  B_____________    2829.____      0.14617646]    0.21353099]
```

```
NAME OF THE SERVER   :  MachineB____

STATE OF THE SERVER :            0__

OCCUPATION RATE      :   0.235815087]

SERVICETIMES IN THE SERVER  :

  customer class    number of      average        variance in
                    arrivals       service time   service time

  A_____________    2828.____      0.61078310]    0.38349075]
  C_____________    2829.____      0.81584945]    0.66139271]
```

Figure 3: A screendump of a simulation with QNM-ExSpect

but not easy to use). An other advantage is the possibility to create your own building blocks using hierarchy constructs. This is an important improvement compared to other graphical simulation tools.
If simulation is not feasible because it takes too long to produce reliable results, then it is possible to generate the network structure and parameters for an analytical model.

References

[1] W.M.P. van der Aalst. Interval Timed Petri Nets and their analysis. *Computing Science Notes, Eindhoven University of Technology, Holland*, 91/09, 1991.

[2] W.M.P. van der Aalst and A.W. Waltmans. Modelling Flexible Manufacturing Systems with EXSPECT. In B. Schmidt, editor, *Proceedings of the 1990 European Simulation Multiconference*, pages 330–338, Nuremberg, Germany, June 1990. Simulation Councils.

[3] W.M.P. van der Aalst and A.W. Waltmans. Modelling logistic systems with EXSPECT. In H.G. Sol and K.M. van Hee, editors, *Dynamic Modelling of Information Systems*, pages 269–288. Elsevier Science Publishers,North-Holland, 1991.

[4] F. Basket, K.M. Chandy, R.R. Muntz, and F.G. Palacios. Open, Closed and Mixed Networks of Queues with Different Classes of Customers. *Journal of the Association of Computing Machinery*, 22(2):248-260, April 1975.

[5] K.M. van Hee, L.J. Somers, and M. Voorhoeve. Executable specifications for distributed information systems. In E.D. Falkenberg and P. Lindgreen, editors, *Proceedings of the IFIP TC 8 / WG 8.1 Working Conference on Information System Concepts: An In-depth Analysis*, pages 139–156, Namur, Belgium, 1989. Elsevier Science Publishers, Amsterdam.

[6] K.M. van Hee and P.A.C. Verkoulen. Integration of a Data Model and Petri Nets. In *Proceedings of the 12th International Conference on Applications and Theory of Petri Nets*, pages 410–431, Aarhus, Denmark, June 1991.

[7] K. Jensen. Coloured Petri Nets: A High Level Language for System Design and Analysis. In G. Rozenberg, editor, *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 342–416. Springer-Verlag, New York, 1990.

[8] M. Ajmone Marsan, G. Bablo, and G. Conte. *Performance Models of Multiprocessor Systems.* The MIT Press, 1986.

[9] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[10] S. Raczynski. Graphical description and a program generator for queueing models. *Simulation*, 55(3):147–152, Sept 1990.

[11] W. Whitt. The Queueing Network Analyser. *The BELL Systems Technical Journal*, 62(9), Nov. 1983.

In this series appeared:

89/1	E.Zs.Lepoeter-Molnar	Reconstruction of a 3-D surface from its normal vectors.
89/2	R.H. Mak P.Struik	A systolic design for dynamic programming.
89/3	H.M.M. Ten Eikelder C. Hemerik	Some category theoretical properties related to a model for a polymorphic lambda-calculus.
89/4	J.Zwiers W.P. de Roever	Compositionality and modularity in process specification and design: A trace-state based approach.
89/5	Wei Chen T.Verhoeff J.T.Udding	Networks of Communicating Processes and their (De-)Composition.
89/6	T.Verhoeff	Characterizations of Delay-Insensitive Communication Protocols.
89/7	P.Struik	A systematic design of a parallel program for Dirichlet convolution.
89/8	E.H.L.Aarts A.E.Eiben K.M. van Hee	A general theory of genetic algorithms.
89/9	K.M. van Hee P.M.P. Rambags	Discrete event systems: Dynamic versus static topology.
89/10	S.Ramesh	A new efficient implementation of CSP with output guards.
89/11	S.Ramesh	Algebraic specification and implementation of infinite processes.
89/12	A.T.M.Aerts K.M. van Hee	A concise formal framework for data modeling.
89/13	A.T.M.Aerts K.M. van Hee M.W.H. Hesen	A program generator for simulated annealing problems.
89/14	H.C.Haesen	ELDA, data manipulatie taal.
89/15	J.S.C.P. van der Woude	Optimal segmentations.
89/16	A.T.M.Aerts K.M. van Hee	Towards a framework for comparing data models.
89/17	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra.

90/1	W.P.de Roever- H.Barringer- C.Courcoubetis-D.Gabbay R.Gerth-B.Jonsson-A.Pnueli M.Reed-J.Sifakis-J.Vytopil P.Wolper	Formal methods and tools for the development of distributed and real time systems, p. 17.
90/2	K.M. van Hee P.M.P. Rambags	Dynamic process creation in high-level Petri nets, pp. 19.
90/3	R. Gerth	Foundations of Compositional Program Refinement - safety properties - , p. 38.
90/4	A. Peeters	Decomposition of delay-insensitive circuits, p. 25.
90/5	J.A. Brzozowski J.C. Ebergen	On the delay-sensitivity of gate networks, p. 23.
90/6	A.J.J.M. Marcelis	Typed inference systems : a reference document, p. 17.
90/7	A.J.J.M. Marcelis	A logic for one-pass, one-attributed grammars, p. 14.
90/8	M.B. Josephs	Receptive Process Theory, p. 16.
90/9	A.T.M. Aerts P.M.E. De Bra K.M. van Hee	Combining the functional and the relational model, p. 15.
90/10	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).
90/11	P. America F.S. de Boer	A proof system for process creation, p. 84.
90/12	P.America F.S. de Boer	A proof theory for a sequential version of POOL, p. 110.
90/13	K.R. Apt F.S. de Boer E.R. Olderog	Proving termination of Parallel Programs, p. 7.
90/14	F.S. de Boer	A proof system for the language POOL, p. 70.
90/15	F.S. de Boer	Compositionality in the temporal logic of concurrent systems, p. 17.
90/16	F.S. de Boer C. Palamidessi	A fully abstract model for concurrent logic languages, p. p. 23.
90/17	F.S. de Boer C. Palamidessi	On the asynchronous nature of communication in logic languages: a fully abstract model based on sequences, p. 29.

90/18	J.Coenen E.v.d.Sluis E.v.d.Velden	Design and implementation aspects of remote procedure calls, p. 15.
90/19	M.M. de Brouwer P.A.C. Verkoulen	Two Case Studies in ExSpect, p. 24.
90/20	M.Rem	The Nature of Delay-Insensitive Computing, p.18.
90/21	K.M. van Hee P.A.C. Verkoulen	Data, Process and Behaviour Modelling in an integrated specification framework, p. 37.
91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.

91/15	A.T.M. Aerts K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.
91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.
91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. 25.
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.
91/29	H. Ten Eikelder R. van Geldrop	Correctness of Acceptor Schemes for Regular Languages, p. 31.
91/30	J.C.M. Baeten F.W. Vaandrager	An Algebra for Process Creation, p. 29.

91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.
91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.
91/35	F.S. de Boer J.W. Klop C. Palamidessi	Asynchronous communication in process algebra, p. 20.
92/01	J. Coenen J. Zwiers W.-P. de Roever	A note on compositional refinement, p. 27.
92/02	J. Coenen J. Hooman	A compositional semantics for fault tolerant real-time systems, p. 18.
92/03	J.C.M. Baeten J.A. Bergstra	Real space process algebra, p. 42.