# A Petri net based simulation tool to evaluate the performance of railway stations

M.A. Odijk
Delft University of Technology, Delft
Dutch Railways, Utrecht
The Netherlands

W.M.P. van der Aalst
Eindhoven University of Technology, Eindhoven
The Netherlands

November, 1993

**Abstract**

This paper addresses the role of simulation in the design process of railway station track layouts. It appears that when irrelevant details are omitted in the modelling of a station, simulation is fast enough to be a powerful method to analyse several design alternatives with respect to several timetables in a relatively short time. The executable specification tool *ExSpect*, which is based on *high level Petri nets*, is used to model railway stations. ExSpect is also used to evaluate the performance of the modelled station by means of simulation.

## 1 Introduction

Designing track layouts of railway stations is difficult, because the timespan between the design stage and the completion of a track construction project is usually very large, say 15-20 years. At such a time distance we only have under-specified and uncertain information about the flow of trains that will visit the station by that time. Moreover, track construction projects are very expensive and also maintaining a track construction involves a lot of money. Hence, the problem of designing a station track layout that is likely to have a sufficiently large capacity to handle future train services, in the absence of concrete information.

In principle, the best possible design is looked for, thus having an optimization problem. However, we feel that this problem is intractable. Therefore, we tackle the design problem by studying a number of promising alternative designs and choose the best one among them.

The capacity of a station track layout can only be assessed in the presence of information



about the way trains make use of the layout. As regards the far future (15-20 years) suc information normally includes frequencies of train services and correspondences betwee trains. Possibly, simple production starting-points such as cross-platform transfers ma also be specified, but information about the exact arrival and/or departure times and th routes of the trains through the station are not available. In fact, routes can never b established as long as the final track layout design is not clear. Finally, some train charac teristics, such as the length and speed of trains, is assumed to be specified.

The set of all timetables that meet these starting-points is called a timetable structur and a station track layout design is evaluated by assessing its capacity to handle severa timetables of the timetable structure. Simulation proves to be a prime one alternativ to make such evaluation. In fact, the remainder of this paper deals with a tool calle *Simulation - Computer Aided Performance Evaluation* (S-CAPE), which simulates railwa stations given a timetable structure.

The executable specification tool *ExSpect* ([3, 5, 6, 9]) has been used to develop S-CAPE In fact, S-CAPE is build on top of ExSpect. The software package ExSpect is based o *high-level Petri nets*, i.e. Petri nets extended with 'time', 'colour' and 'hierarchy' (see ap pendix A). ExSpect is developed at Eindhoven University of Technology and, in fact, comprises a number of tools: a graphical editor, a type-checker, an interpreter, an anima tion interface and an analysis tool. The appendix provides a short introduction to ExSpec

This paper is organised as follows. We continue with modelling a railway station in a informal manner. Then we proceed with a formal specification in ExSpect. Such specifica tion is executable, so we are able to simulate the modelled railway station. The simulatio tool is tested on Arnhem Central Station, which is a major station in the Netherlands The results of these tests are reported in section 4. Finally, some conclusions are listed i section 5.

## 2 Modelling a railway station

From abstract point of view, the track layout of a station is a concatenation of so-calle track elements. A track element is a (usually) small part of the layout that can be use by one train at a time. For instance, a switch plus its vicinity is often identified as a trac element. In practice a track layout is subdivided into track elements based on the safet installations present. In the example this means that the vicinity of the switch is bounde by signals or other methods that prevent two trains from entering the switch simultane ously. Still, in theory every possible subdivision of a track layout can define the trac elements. Figure 1 shows an example of a track layout together with a possible subdivisio of it into track elements.

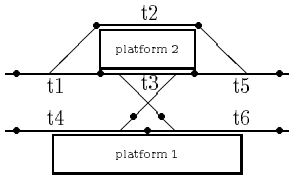


Figure 1: *A station track layout together with a possible subdivision of it into track elements. The track elements are named t1 through t6 and they are bounded by fat-printed dots.*

The subdivision of a track layout into track elements makes it possible to express routes to and from platforms in terms of track elements. For instance, $(t1, t3)$ can is the route a train follows if it wants to halt at platform 2. Conflicts between trains can be identified by the non-emptyness of the intersection of their routes. Hence, the track layout is modelled implicitly by a set of routes, thereby abstracting from the true geographics.

A train which enters the railway station requests a route to a platform, halts at the platform, waits for a while and then claims an exit route. A non-halting train, like freight trains, request a complete route through the station. The track elements are free or occupied according to the section-release route-locking principle, see [7]. This principle means that all track elements of a route are occupied at once and they are released one by one in order of usage by the train.

# 3 Formal specification

In order to facilitate the simulation of a train station, a software tool, called S-CAPE, has been developed. S-CAPE is build on top of ExSpect and runs under UNIX on a SUN workstation. However, users can simulate various alternatives without knowing the ExSpect specification language (see appendix A).
A simulation run consists of a number of subrun, each of which starts with the generation of a timetable from a timetable structure. The timetables are then simulated one by one and the output statistics are aggregated so as to make a statement about the performance of the track layout - timetable structure combination. These output statistics involve process times, occupation rates, blocking probabilities, etc.

The implementation of S-CAPE consists of several parts (subsystems) all of which operate in an interactive fashion. The most important parts are the so called *Dispatcher* and *Train Movement*. The former controls the flow of trains through the station. The latter handles the trains in a more physical manner, i.e. it simulates the movements of the trains that are released by the Dispatcher. Together, these subsystems form the heart of the sys-



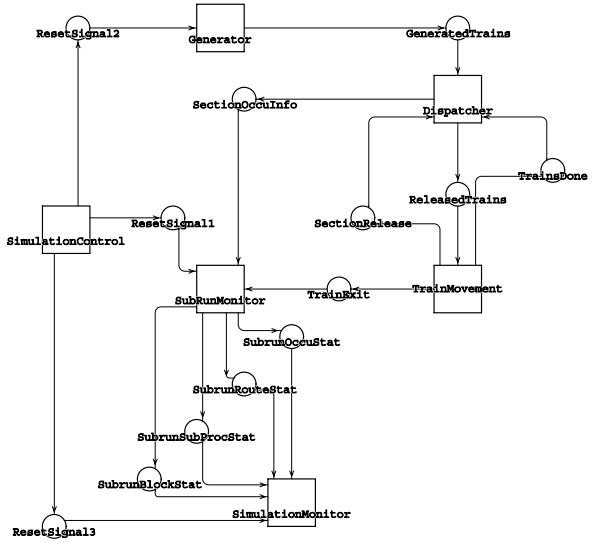


Figure 2: *ExSpect interaction structure of S-CAPE. A box represents a subsystem with certain functionality, whereas a circle denotes the transfer of information.*

tem. Around the heart the station's environment is modelled by the subsystems *Generat* and *Subrun Monitor*. The Generator generates the incoming trains, whereas the Subru Monitor monitors a subrun. Also, the Generator samples a timetable from the timetabl structure at the beginning of a subrun. The whole of subsystems as described so far surrounded by a third layer of subsystems, the so-called *Control Layer*. The subsystem in this layer are *Simulation Control* and *Simulation Monitor*. Simulation Control trigge the Generator and the Subrun Monitor after each subrun and it triggers the Simulatio Monitor after the last subrun. Figure 2 shows the interaction structure of the subparts S-CAPE graphically.

To show an example of how a subsystem may look like in ExSpect, let's examine the Di patcher in more detail, see figure 3. The Dispatcher is in fact the engine of the systen It is also the most complex subsystem. There are several stores that contain informatio about the state of the entire system. Store `SectionOcc` holds information about the stat of each track element (free or occupied). It is updated by its neighbouring processors a soon as a track element changes its state. Another store, `QueueOut`, registers the presenc of trains at platforms, whereas `QueueIn` keeps track of the queueing process at the entranc tracks.

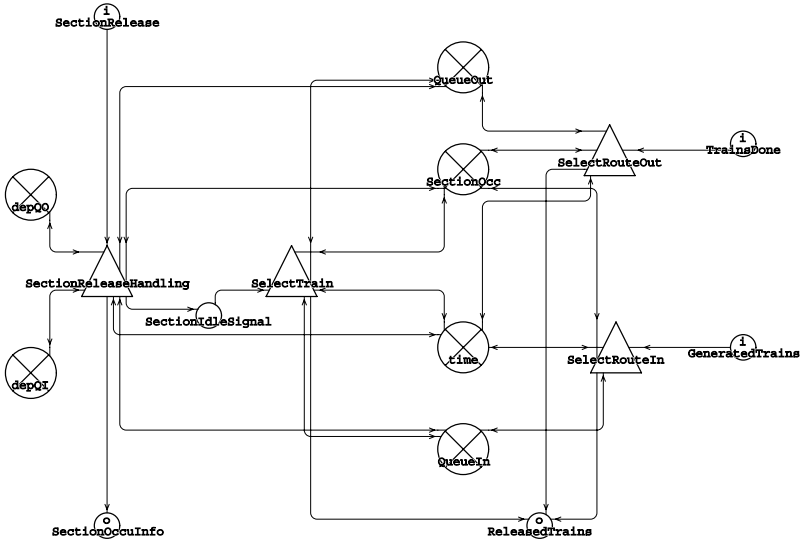


Figure 3: *Contents of subsystem Dispatcher.*

Dispatcher handles two flows of tokens. By means of the input channels `GeneratedTrains` and `TrainsDone` and output channel `ReleasedTrains` Dispatcher receives ingoing trains from Generator, it receives outgoing trains from Train Movement and it sends outgoing trains back to TrainMovement. The other flow concerns the handling of track elements. This is done via the input channel `SectionRelease` and the output channel `SectionOccuInfo`. The signature, i.e. the formal interaction structure, of the Dispatcher looks like:

```
sys Dispatcher[
   in GeneratedTrains : Train,
      TrainsDone : Train,
      SectionRelease : SectionId,
   out ReleasedTrains : Train,
       SectionOccuInfo : SectionId >< Clock ]
```

Each time a track element becomes free, processor `SectionReleaseHandling` updates `SectionOcc`, gives a signal to processor `SelectTrain` when this is useful and it puts information about the track element in channel `SectionOccuInfo`. If a token is produced for channel `SectionIdleSignal`, `SelectTrain` starts looking for an outgoing train for which a route has become available. If `SelectTrain` succeeds, the train found is released and put in output channel `ReleasedTrains`. Also the stores `QueueOut` and `SectionOcc` are updated



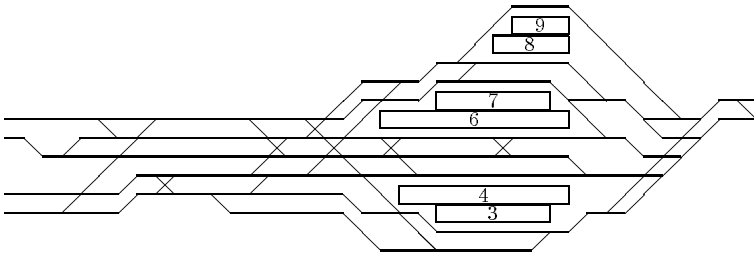


Figure 4: *Track layout of Arnhem CS.*

in this case. If the request was denied, `SelectTrain` starts looking for ingoing trains tha are at poll position of some queue and for which a free route exists. If it succeeds, th train is selected and `QueueIn` and `SectionOcc` are updated, otherwise `SelectTrain` res its case.

When a newly generated train puts a call for service, i.e. a token is produced for inpu channel `GeneratedTrains`, processor `SelectRouteIn` checks if this train has a free rout If so, the train is send to `ReleasedTrains` and store `SectionOcc` is updated. If not, th arriving train is put on hold by updating store `QueueIn`. The same story applies to outgo ing trains by substituting `GeneratedTrains` and `QueueIn` for `TrainsDone` and `QueueOut` respectively.

# 4 Simulation results

We tested S-CAPE on a Arnhem CS, which is a large railway station in the Netherlands The future layout of Arnhem CS is shown in figure 4. The timetable structure we use loaded the station each subrun with about 30 trains per hour. The tests concentrate mainly on the sensitivity of the simulation model to the type of stochastic distribution that were used to model the arrival disturbances and the halting times. The appear that this sensitivity is in the order of 4 to 5 percent of the output statistics when varyin the variation coefficient of the stochastic distributions considerably. Also the sensitivit of the simulated system to the subdivision of the track layout into track elements wa investigated.

# 5 Conclusions

This paper showed how the Petri net based specification language ExSpect can be used t specify and simulate railway stations. Since the concept of Petri nets allows for the reus of already made building blocks, ExSpect is extremely useful in the experimental stage

the development of a simulation tool like S-CAPE.

With S-CAPE the performance of a railway station with respect to a timetable structure can be evaluated. In this paper, the track layout of the station is the target. However, if we consider the timetable structure to be our target, we see that we can use S-CAPE just as well to assess the quality of timetable structures. Hence, in the short run, timetabling problems can also be solved with S-CAPE. A similar experience is reported in [1], where railway station development projects are evaluated using Petri net based analysis methods.

## References


[1] W.M.P. van der Aalst and M.A. Odijk. *Analysis of Railway Stations by Means of Interval Timed Coloured Petri nets.* In preparation.

[2] W.M.P. van der Aalst, *Modelling and Analysis of Complex Logistic Systems*, in Integration in Production Management Systems, H.J. Pels and J.C. Wortmann, eds., vol. B-7 of IFIP Transactions, Elsevier Science Publishers, Amsterdam, 1992, pp. 277–292.

[3] ——, *Timed coloured Petri nets and their application to logistics*, PhD thesis, Eindhoven University of Technology, Eindhoven, 1992.

[4] ——, *Interval Timed Coloured Petri Nets and their Analysis*, in Application and Theory of Petri Nets 1993, M. Ajmone Marsan, ed., vol. 691 of Lecture Notes in Computer Science, Springer-Verlag, New York, 1993, pp. 453–472.

[5] W.M.P. van der Aalst and A.W. Waltmans, *Modelling logistic systems with EXSPECT*, in Dynamic Modelling of Information Systems, H.G. Sol and K.M. van Hee, eds., Elsevier Science Publishers, Amsterdam, 1991, pp. 269–288.

[6] ASPT, *ExSpect 4.1 User Manual*, Eindhoven, 1993.

[7] J. Bourachot. *Computer-aided planning of traffic in large stations by means of the AGAIF model.* In: Rail International, May 1986, pp. 2-18.

[8] K.M. van Hee, *Information System Engineering: a Formal Approach*, Cambridge University Press, (to appear) 1994.

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





[10] K. Jensen, *Coloured Petri Nets: A High Level Language for System Design an Analysis*, in Advances in Petri Nets 1990, G. Rozenberg, ed., vol. 483 of Lecture Note in Computer Science, Springer-Verlag, New York, 1990, pp. 342–416.

[11] ——, *Coloured Petri Nets. Basic concepts, analysis methods and practical use.*, EATC monographs on Theoretical Computer Science, Springer-Verlag, New York, 1992.

[12] T. Murata, *Petri Nets: Properties, Analysis and Applications*, Proceedings of th IEEE, 77 (1989), pp. 541–580.


## A ExSpect

The simulation tool described in this paper uses the software package ExSpect. ExSpect based on a high-level Petri net model. This high-level Petri net model extends the classic Petri net with 'time', 'colour' and 'hierarchy'. The *software package ExSpect* can be use to create, modify or analyse specifications written in the *specification language ExSpect*. S the name ExSpect is used for two things: a software package and a specification languag

The software package ExSpect consists of a number of tools. A typechecker and an in terpreter have been developed. The typechecker checks the specification for correctnes completeness and consistency. The interpreter can be used to execute such a specificatio There are two modes of operation one for prototyping and one for simulation. Since th user can view or influence a running simulation, interactive simulation is possible.
The software is written in the language C and runs under UNIX at a SUN workstatio in a window environment. For more information about the ExSpect system the reader referred to [3, 5, 6, 9].

ExSpect uses a formalism based on high-level Petri nets. A *high-level Petri net* consists two kinds of components: *processors* and *channels* (corresponding to respectively trans tions and places in the classical Petri net model, see [12]).
A processor is connected to one or more input channels and zero or more output channel To each association of an input channel to a processor a certain (positive integer) weight attached (in most cases weights are equal to 1). This allows a channel to be a multiple inpu channel of a processor. With each channel a type is associated and with each processor function. The signature of the function of a processor $p$ corresponds with the types an 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 mome channels may contain *triggers* (tokens in Petri nets). A trigger has a *value* that belongs t the type of the channel and a *time stamp* that belongs to some ordered set $T$. There ma be more triggers in the same channel with the same value and time stamp. So a chann actually contains a certain bag of triggers over its type.
At any moment a *transition* may occur, which means that the configuration of trigger called the *state*, may change (our terminology attaches another meaning to the term tra

sition than Petri net theory). Such a transition occurs instantaneously and is *executed* by the processors. A processor may execute if it is able to select the right number of triggers from each of its input channels. The number of triggers to be selected from an input channel by a processor $p$ is equal to the weight of the input channel for $p$. The execution of a processor means that the selected triggers are consumed (deleted) and that new triggers for the output channels of the processor are produced. Note that a trigger can be consumed only once.
An *event* is an assignment of triggers to processors such that each processor can execute. The (simulated) *event time* at which an event may occur is the maximum of the time stamps of the triggers 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 triggers will be at least equal to the event time. It is thus clear that the transition times of successive events will be non-descending.
By not specifying any delays, we have a timeless model: events are ordered only by causality and triggers can be consumed only after production.
To represent a high-level Petri net we use a diagram technique. Each processor is represented by a triangle and each channel is represented by a circle. A *store*, denoted by a circle marked with a cross, is a special kind of channel always containing one trigger (token).

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 'strong 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 complete description of the functional part 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. 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 symbolise new (composite) types. We can attach names to type expressions, thus defining new types. The following example illustrates some type definitions:

```
type weight from real with [x] x >= 0.0;
type volume from real with [x] x >= 0.0;
type manufacturer from str;
type truck from manufacturer >< (weight >< volume);
type truck_id from num;
type fleet_of_trucks from truckid -> truck;
type cargo from weight >< volume;
```



Note that we can add a `with` part for restricting the type.
Likewise we can construct new functions. Our set of basic functions includes all wel known set-theoretical, logical and numerical constants and functions. These functions ar often polymorphic. Because of some 'sugaring' it is possible to write these functions i their usual symbolic infix or 'circumfix' notation. As an example we show two functio definitions operating on the types defined above:

```
transportable_by_truck[ c :  cargo, t :  truck ]
:=
(pi1(c) <= pi1(pi2(t))) and (pi2(c) <= pi2(pi2(t))) :  bool;

transportable_by_fleet[ c :  cargo, f :  fleet_of_trucks ]
:=
if f = {}
   then false     --i.e.  there are no trucks left
   else transportable_by_truck(c,pi2(pick(f)))
        or transportable_by_fleet(c,frest(f))
fi :  bool;
```

The functions `pi1`, `pi2` (projections), `pick` and `frest` (respectively taking and deletin and element from a mapping) are examples of basic functions.
Processor definitions are split in a header and contents part. The header part (sometime called signature) contains the processor name, its interaction structure and its parameter The interaction structure is given by (possibly empty) lists of input channels, outpu channels and stores. The contents part consists of concurrent (conditional) assignments expressions to output channels and stores. A simple example is as follows:

```
proc transport_function[in leave:truck, out arrive:truck, val d:time]
:=
arrive <- leave delay d;
```

This processor can be used to model the transport as a simple delay. The time betwee the departure and arrival of a truck is set by a value parameter `val d`. Note that `delay` a keyword.
We define a system as an aggregate of processors, connected by channels and stores. store is a special kind of channel: it always contains precisely one trigger. A system ca also contain other (sub) systems. If a system has no interaction with its environment w call it a closed system else an open system. Open systems communicate with the outsid world via input and output channels and stores. Therefore a system definition consis of a header similar to a processor header and a contents part. A system can have valu function, processor and even system parameters. So its possible to define generic system This way a system can be customised or fine-tuned for a specific situation. The content

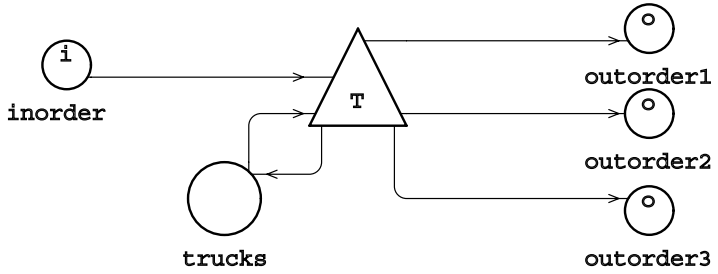


Figure 5: *A simple transport system.*

part is a list of all the objects (processors, systems and local stores and channels) in the system. As an example we show a simple transport system, see figure 5.
Note that we represent a processor by a triangle and a channel by a circle.
The transport system has one input channel to accept incoming transportation orders. There are a number of output channels (for each destination one). So every incoming order leaves the system after some time through one of the output channels on the right. The specification is shown in the box.
In this example the system has a function (`fun`) and a value (`val`) parameter. The value parameter `noftrucks` represents the number of trucks initially available. The function parameter is used to calculate the time it takes to transport a cargo, given the `origin` and `destination`. Processor `T` is activated if there is at least one order and one truck available. If all trucks are busy, `T` cannot accept new orders.
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 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 you can 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 can hide 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 creation of 'toolboxes'. The S-CAPE tool described in this paper is a toolbox developed to analyse railway stations.



```
type truck from void;
type order from void;
type coordinate from real;
type destination from coordinate >< coordinate;
type origin from  coordinate >< coordinate;
atruck := {} : truck;
consumer1 := <<1.,2.>> : coordinate >< coordinate;
consumer2 := <<3.4,2.>> : coordinate >< coordinate;
 ..
consumern := <<1.5,2.3>> : coordinate >< coordinate;

sys Transport [ in inorder: order >< (origin >< destination),
                out outorder1 : order,
                    outorder2 : order,
                      ...
                    outordern : order,
                val noftrucks : num,
                fun traveltime[x:origin >< destination] : real
              ]
:=
channel trucks : truck init {<<atruck,noftrucks>>},
T(in inorder, trucks, out outorder1, outorder2, .. outordern, trucks)
where
  proc T [ in io : order >< (origin >< destination), it : truck,
           out oo1 : order, oo2 : order, .. oon : order,
               ot : truck
         ]
  :=
  ot <- it delay traveltime(pi2(io)),
  if pi2(pi2(io)) = consumer1
    then oo1 <- pi1(io)
  else if pi2(pi2(io)) = consumer2
    then oo2 <- pi1(io)
  ..
  else oon <- pi1(io)
  fi .. fi;
end;
```