Proceedings of the 1999 Winter Simulation Conference
P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans, eds.

KNOWLEDGE-BASED MODELING OF DISCRETE-EVENT SIMULATION SYSTEMS

Henk de Swaan Arons

Erasmus University Rotterdam
Faculty of Economics, Department of Computer Science
P.O. Box 1738, H9-28
3000 DR Rotterdam, THE NETHERLANDS

ABSTRACT

Modeling a simulation system requires a great deal of customization. At first sight no system seems to resemble exactly another system and every time a new model has to be designed the modeler has to start from scratch. The present simulation languages provide the modeler with powerful tools that greatly facilitate building models (modules for arrivals or servers, etc.). Yet, also with these tools the modeler constantly has the feeling that he is reinventing the wheel again and again. Maybe the model he is about to design already exists (maybe the modeler has designed it himself some time ago) or maybe a model already exists that sufficiently resembles the model to be designed. In this article an approach is discussed that deploys knowledge-based systems to help selecting a model from a database of existing models. Also, if the model is not present in the database, would it be possible to select a model that in some sense is close to the model that the modeler had in mind?

1 INTRODUCTION

Modeling is one of the most difficult and time-consuming parts of the simulation process. The design of the conceptual model is fundamental for the quality of the simulation results and requires a lot of knowledge and experience. The next step, designing and developing an implementation model based on the conceptual model also requires a lot of experience. Naturally, to a large extent, the implementation model heavily depends on the basic concepts of the conceptual model.

In order to reduce the amount of work in this stage of the simulation process, simulation languages provide increasingly powerful tools (e.g., modules for frequently occurring parts of a model, such as arrival processes, servers, conveyors, transporters, etc.) that facilitate the modeler in modeling simulation systems. Despite these advanced tools the modeler constantly has the feeling that he is reinventing the wheel again and again. Did he himself or someone else not build a similar model or sub-model in the past? Was such a model not built somewhere else. And if so, how could he retrieve this model in a systematical manner? In order to make all this possible, previously developed models have to be stored in a database of simulation models for later retrieval. Furthermore, once such a database exists, how does one retrieve the right model according to the present data? And if the right model is not in the database, would it be possible to select a model that in some sense is close to the model that the modeler had in mind? This article is concerned with these questions. A possible answer is expected to be found in the support of knowledge-based systems, in particular expert systems.

In brief, the approach discussed in this article is as follows. In the course of time many implementation models have been developed using some simulation language. These models could have been stored in a database with the intention to reuse them in a later stage. When a new model has to be designed a modeler could try to retrieve an already existing model in the database similar to the one that has to be developed. In order to be able to make this choice the models stored in the database need to be parameterized so that a query on this database could be formulated and hopefully will result in a non-empty dynaset. Since the modeler in principle has no knowledge or does not want to have knowledge of the parameters used to describe the models in the database an expert system could be used to transform the design specifications into the right query, taking all kinds of design considerations into account.

Globally there were three reasons to start this research. The first reason has been an article (De Swaan Arons 1983) that investigated the way in which expert systems could support simulation in general and modeling in particular. This article concerned how to determine the mathematical model of an oscillator. A starting-point has been that modeling requires much experience and that expert systems claim to do a good job in this field. Now the tools for building expert systems have become much more advanced, robust and easier to use (see for example AionDS (Platinum Technology 1996) that is briefly described in section 3.2.1) there is enough reason to look again at the applicability of expert systems in the field of simulation, in particular in modeling.

A second reason has been the impressive development of simulation languages in the last decade. Simulation languages have been on the market for a long time, but in the past they rarely supported graphical model building (for example, GPSS, SIMAN, etc.). Now most systems are graphical: Arena (the graphical successor of SIMAN (Pegden, Shannon, and Sadowski 1995)), Taylor II and recently Taylor ED (1998), ProModel (and its various versions such as ServiceModel and MedModel) are well-known examples. All of these simulation languages have in common that they offer a set of implementation modules for often occurring sub-models. However, even with these more advanced building blocks a great deal of the design work has still to be done by hand.

The last reason has been a feasibility study that was initiated in 1997. It concerned a so-called toolkit of existing simulation models in combination with a shop window in which these models would be made available to be used for new to be developed models (Toolkit 1997). Initially, a bottom-up approach was chosen in which the toolkit is filled with models developed in pilot studies. The final aim of the project has been to make the knowledge gathered in pilot studies and other projects in Rotterdam Harbor accessible in relation to logistic problems. In fact, the toolkit may be seen as a database of models.

Before we briefly discuss the various sections a remark has to be made about the use of the words model and sub-model. For the sake of brevity in the following for both concepts the word *model* will be used, unless there is a good reason to deviate from this rule. It must be emphasized however, that the following will mainly concern sub-models.

In Section 2 models in general and implementation models in particular are discussed. Also the notion database of models is introduced and how they can be dealt with. The various aspects of knowledge-based modeling are discussed in Section 3 among which the approach chosen in this article, and how expert systems can support this. To make the use of expert systems understandable in this section also some introductory remarks are made to some aspects of expert systems in general and to one knowledge-based tool in particular. Finally, in section 4 some conclusions are drawn.

2 DATABASE OF MODELS

In the following it is useful to emphasize the difference between a conceptual and an implementation model. A conceptual model is seen as a model that is formulated completely independent of any programming or simulation language. An implementation model is associated to some specific programming or simulation language, although it is still a model on paper. A database could contain both types of models. An example is the mathematical model of the oscillator discussed previously. The conceptual model of this oscillator consists of a set of differential equations and an expert system was utilized to select the correct set of equations from a number of possible sets (a kind of database) based on the observed characteristics of the system independent of any implementation language. It is also possible that a database contains implementation models written in some simulation language and such a database is supposed to be used in combination with that simulation language. In this article we assume that the database only contains implementation models that are used in combination with the simulation language Arena. In most simulation languages modules are provided that can be used on various levels of aggregation.

An implementation model in Arena can be fully represented by a number of parameters. Such an implementation model consists of a number of modules each of which can be further specified. For example, an Arena model could contain 2 Arrive, 2 Server, 2 Inspect and 3 Depart modules, see Figure 1. This simple model is taken from the book Simulation with Arena (Kelton, Sadowski, and Sadowski 1999). Two different types of entities enter the system and each of these has its own Arrive module: Part A Arrive and Part B Arrive. After arrival they proceed to their own workstation for preparation: Part A Prep and Part B Prep. After preparation both parts proceed to the same workstation Sealer. On this workstation the parts are not only processed (sealed) but also tested. If they pass this test they exit the system through Shipping, otherwise they proceed to a second workstation for further rework and testing. If they pass this test they exit the system through Salvaged Parts, otherwise through Scrap.

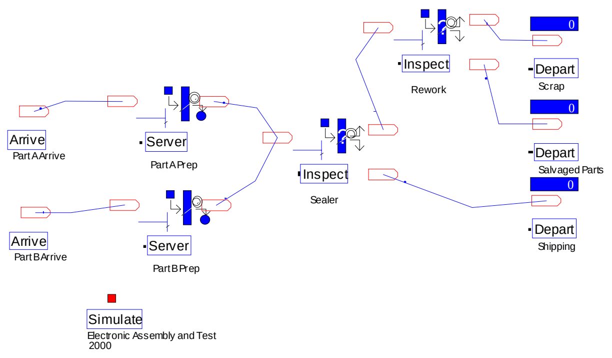

Figure 1: An Implementation Model in Arena

All modules can be further specified. For example, Part A Arrive specifies a batch size of 1, the distribution function for the interarrival times is exponential: EXPO(5), Arrival Time is defined as the mark time attribute, the attribute Sealer Time defines the distribution function (TRIA(1,3,4)) of the Part A processing times at the Sealer, the route to the Server Part A Prep is specified and the time that this will take (2 time units). Some of these data are presented in Figure 2, but the actual number of parameters is much larger. The implementation model could be described by the complete set of these parameters (of course also those of other modules) and the question is if this model can be retrieved when a more or less identical model has to be designed.

An obvious approach is to formulate and run a query in which all necessary parameters have the desired values and subsequently to find out if the database indeed contains the corresponding implementation model. Such an approach requires sufficient knowledge of all possible options that the Arena modules offer and it is just this kind of knowledge that can be stored in an expert system.

At this point it is useful to make a remark about the notion interpolation and extrapolation of models. Even if the database will contain a very large number of models it will occur frequently that the desired model will not be present in the database. Usually there will be differences that make the query generate an empty dynaset. Much more often the database will contain models that are more or less similar to the one that the modeler is after. When this is the case the desired model could possibly be positioned in between of these models and it could be considered as an interpolation of these models. It is up to the expert's judgement if this is possible and this part of the job is a task that seems very well suited to be carried out by an expert system.

Arrive 1 (Part A Arrive)		**Server 1 (Part A Prep)**	
Batch size	*1*	Resource	*Part A Prep_R*
Time between	*EXPO(30)*	Capacity type	*Capacity*
Attribute: Sealer Time	*TRIA(1,3,4)*	Capacity	*1*
Route / Station Name	*Part A Prep*	Process Time	*TRIA(1,4,8)*
Route Time	*2*	Route / Station Name	*Sealer*
…	…	Route Time	*2*
…	…	…	…

Figure 2: Parameters Specified for an Arrive Module and a Server Module

It must be noted that even simple models are described by a large number of parameters, which will not be enumerated here. We restrict ourselves by only mentioning that some important parameters are the number of Arrive, Server, Inspect and Depart modules. Furthermore, also the route that the various entities have to follow is essential.

3 KNOWLEDGE-BASED MODELING

In this section we will describe how expert systems can be deployed to support the modeling process. On the basis of a few examples in 3.1 the approach is demonstrated. In 3.2 a brief introduction is given to the knowledge system tool AionDS, in 3.3 some remarks are made about the query and in 3.4 the notion inter- and extrapolation of models is looked at in some more detail.

3.1 A Global Approach

The approach globally exists of the following steps and the whole process is assumed to be controlled by the expert system:

1. The modeler is asked - directed by the expert system - to provide the characteristics of the model that has to be developed. It is certainly no questionnaire, only those questions are asked that are relevant to the expert system's aims.
2. The expert system 'converts' these characteristics into parameter values.
3. Based on these parameter values a query will be formulated and executed on the database.

The result will be a dynaset containing zero or more implementation models.

The expert system will control the process. In order to be able to do so it will complete an agenda containing the following actions. First it will determine which types of modules the model will have to contain and how many instances of each module. In the example of Figure 1 the model has to contain Arrive, Server, Inspect and Depart modules; Arrive, Server and Inspect with two instances each, and Depart with three instances. Furthermore, a Simulate model will contain details about the number of replications, etc. For the time being we will restrict ourselves to the modules Arrive, Server, Inspect en Depart. After this stage has been completed, the expert system will proceed by trying to determine the routes of the entities in the system. For the design of the model it is important to know if the various entities will follow different routes. If so, a Sequences module will have to be added that accurately stores route information. Finally, the other parameters will have to be defined as well, such as the batch size, the resources, the capacity of the Servers, whether or not schedules and failures have to be defined and if so, how they look like.

As an example, let us again consider Figure 1. There are two types of entities: Part A en Part B. Both entities have different distribution functions for the interarrival times and therefore two different instances of the Arrive module have to be specified. Since both entities have to be processed by the same Sealer and their processing times are different, the distribution functions of Sealer Time will have to be specified separately for both entities; in Arena this needs to be done in the Arrive modules. In this model there are two workstations that process incoming entities and transfer them to only one other module. This route information can be dealt with by a Server. There are also workstations that - after processing - dynamically transfer the entities to one or two possible modules. In this case Inspect modules are used. From the figure it is clear that three Depart modules are needed. The reasoning behind all this can be formulated in terms of a number of business rules.

Once the type of modules and the number of instances of each type have been specified also the routes of the entities between the modules can be specified. Since the route can be specified per module a simple questionnaire can be used to determine what routes are followed by which entities. For example, for entities of Part A the following route will apply:

Part A Arrive →Part A Prep →Sealer →Rework *or* Shipping
If Rework →Scrap *or* Salvaged Parts

Part B will follow the route:

Part B Arrive →Part B Prep →Sealer →Rework *or* Shipping
If Rework →Scrap *or* Salvaged Parts

In Figure 3 another, equally simple implementation model is outlined in which the route of entities depends on the type of entity (this example is taken from Arena course material (Course Arena 1997)). Entities enter the system at Arrive module Part Arrivals. There are two types of entities (Typical and Special with the same distribution function for the interarrival times) that will follow different routes.

For Typical:

Part Arrivals → Prep → Paint → Dry → Warehouse Typical

For Special:

Part Arrivals → Paint → Paint → Dry → Warehouse Special

At arrival, in the Arrive module it is specified what percentage of the incoming entities will be Typical and what percentage will be Special.

Although the two types of entities (Typical and Special) follow different routes through the system, both types have the same distribution function for the interarrival times and therefore it is sufficient to have only one Arrive module. Since three types of processing are specified also three workstations are necessary: Prep, Paint and Dry, be it that not all of them are used by both types of entities. The routes depend of the type of entity, which can be read from Figure 3. From the modules Part Arrivals, Paint and Dry two different routes emerge. After arrival entities of type Typical will proceed to Prep and entities of type Special will proceed directly to Paint. Furthermore, some entities leaving the module Paint will go back to Paint (Special) and those of type Typical will immediately proceed to Dry. Finally, entities of type Typical will leave the system through Depart module Warehouse Typical while those of type Special will exit through Depart module Warehouse Special. To make all of this possible, in the module Part Arrivals we have to discriminate between entities of types Typical and Special (how many percent is of one type and how much of the other). Furthermore, for both types different routes have to be specified. Arrive, Server and Inspect modules cannot discriminate between the two types with respect to the route they have to follow. Therefore, a separate module has to be used: the Sequences module. In the figure we also see that a Transporter is used but we leave this aspect out of consideration now.

Again, also these descriptions can be formulated in terms of business rules, based on which the expert system can decide which modules (and how many instances of each module) the model has to contain and whether or not a Sequences module needs to be added.

3.1.1 Determining the Modules

In this section a number of heuristics will be formulated in an informal manner. These are used to specify the number of instance of Arrive, Server, Inspect and Depart modules. Also some other parameters (such as whether or not a Sequences module needs to be added to the model or whether some attributes have to be defined) will be dealt with.

The following considerations help to determine the number of Arrive, Server, Inspect and Depart modules.

1. If an entity type has a distribution function for the interarrival times different from other entity types then a separate Arrive module is required;
2. The number of Arrive modules equals the number of different distribution functions;
3. If a workstation has only one single processing task then the workstation has to be represented by a Server module;
4. If a workstation has both a processing and a testing task then the workstation has to be represented by an Inspect module;
5. The number of Server modules equals the number of workstations that are represented by a Server module;
6. The number of Inspect modules equals the number of workstations that are represented by an Inspect module;
7. If a Server / Inspect module has to process 2 or more entity types that have different

processing times on this Server / Inspect module then define in the Arrival modules an attribute that specifies the processing time on this Server / Inspect module for the corresponding entity type;

8. If a Server / Inspect module has to process 2 or more entity types and the route information of how to proceed depends on the entity type then a Sequences module needs to be added to the model;
9. If an Arrive module generates two or more entity types for which the route information depends on the entity type then a Sequences module needs to be added to the model;
10. The number of Depart modules equals the number of entity types that leave the system (note that on their way through the system one single entity type can split up into two or more entity types or also, different entity types can be merged to a single one).

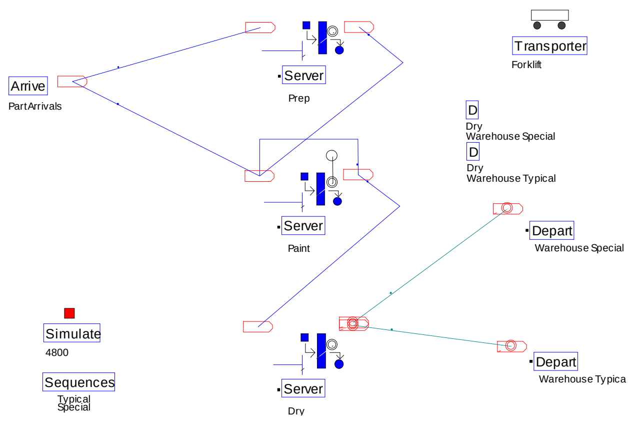

Figure 3: Different Entities Follow Different Routes

Naturally, this list is only a small part of all considerations that could help to determine the number of the modules. For example, whether or not an entity type has to have an own distribution function could be concluded from other considerations.

Some of the above items could easily be formulated in the business rule format as described in 3.2.2.

3.1.2 Determining the Routes

Once the number of Arrive etc. modules have been specified, it is possible to determine the routes that the various entity types have to follow. Here a difference must be made between whether or not a Sequences module has to be used. In the previous section it was concluded that in some cases a Sequences module has to be used. However, not always all route information needs to be specified in the Sequences module. Route information can partly be specified in the Sequences module and partly in the route section of the Server or Inspect modules.

3.1.3 Determining Remaining Parameters

The category of remaining parameters is diverse and may be quite large. Examples of these type of parameter are batch size, capacity type (schedule or capacity), process time, resource, etc. To determine these parameters a similar kind of reasoning can be used.

In order to determine the process time of a Server or an Inspect module a number of business rules can support to find the appropriate distribution function. For example, the lognormal distribution function is frequently used to represent processing times that have a distribution skewed to the right; the triangular distribution is commonly used in situations in which the exact form of the distribution is not known, but estimates (or guesses) for the minimum, maximum and the most likely values are available; and the uniform distribution is used when all values over a finite range are considered to be equally likely. It is sometimes used when no information other than the range is available (Kelton, Sadowski, and Sadowski 1999). So, in cases that no distribution function is available beforehand, nor empirical data from which a distribution function can be derived from, an appropriate choice can be made supported by some expert help from an expert system. If sufficient data are available then the Arena tool Input Analyzer can be used (the expert system could offer help to carry out this task automatically).

To determine whether a Server has either the Capacity Type capacity or schedule, it is important to know whether a Server is available continuously or at scheduled times. Also this can be determined by a number of relevant business rules, again with the result that the parameter Capacity Type has either the value capacity or schedule.

3.2 AionDS Support

In this section the AionDS knowledge tool will briefly be discussed: its global working, the agenda mechanism, the inference engine and the business rules.

3.2.1 Brief Introduction of AionDS

AionDS is a development environment for creating knowledge-based applications - applications that apply complex business logic and data modeling to solve problems. Using AionDS, one can develop object-oriented, knowledge-based applications quickly and easily (Platinum Technology 1996). First of all, applications built with AionDS have a knowledge base. This is a set of knowledge structures that represent application knowledge. Different knowledge structures are used for different kinds of application knowledge. Furthermore, AionDS is object-oriented (it has classes and instances which can be used to model modules and their instances as mentioned previously). A class defines two major parts: slots and methods. The slots can be used to represent the parameters of a module. As any knowledge-based environment, AionDS has an inference engine, a program that combines and applies relevant data, facts, and business rules in the knowledge base to reach a goal or to draw a conclusion based on relevant data. AionDS knows the notion state, which contains two main types of information: the high-level instructions that direct the inference engine and the business logic (the business rules) that examines and makes decisions about data. Each state has an agenda, which defines its high-level (control) instructions. When an application runs, the inference engine follows the agenda to see what should be carried out next.

Particularly the notions classes and instances, agenda, business rules and inference engine are useful in the context of this article.

3.2.2 Agenda, Inference Engine, Business Rules

An AionDS knowledge base generally consists of a number of states, each of which containing an agenda of actions to be carried out and - in most cases - a number of rules expressing the knowledge relevant to that state. Apart from the 'root' state Main a knowledge base that supports modeling discrete-event simulation models in Arena may consist of a state Analysis that specifies the tasks listed in section 3.1 that have to be carried out. For this reason the state Analysis would contain 'calls' to the subsequent states DetermineModules, DetermineRoutes and DetermineRemainingPararameters. The state DetermineModules would contain a number of business rules that determine the types of modules that will be necessary for the model to be developed and how many instances of each type. This reasoning process will be started by some command given in the agenda of DetermineModules. The result will be the number of instances of each type of module. The other two states determine the routes and the remaining parameters.

Business rules in AionDS are based on the instances and their slots defined in the OO domain description. For example, the following business rules describe which distribution function has to be used for the processing time of a Server1 being an instance of the class Server:

```
If
Server1.DistributionFunction is unknown and
Server1.ProcessingData is not 'present'
Then
Server1.DistributionFunction is 'triangular'

If
Server1.DistributionFunction is unknown and
Server1.ProcessingData is 'present'
Then
Server1.DistributionFunction is Get(DistributionFunction, …)
```

AionDS has an inference engine that supports various types of reasoning. It supports both backward and forward chaining and also a mixture of these.

3.3 The Query

Once the expert system has analyzed the modeler's design considerations and transformed into a set of parameters, a query can be formulated and executed. In its most simple form the database exists of a table with the parameters as fields. One of those fields links the record to a corresponding implementation model.

Suppose a modeler has in mind to design a model as outlined in Figure 1. In interaction with the modeler the expert system has specified that the model parameters would be - among others -: a = 2 (Arrive), s = 2 (Server), i = 2 (Inspect) and d = 3 (Depart). Furthermore the routes of the entities (of only one entity type) have been determined. That is to say, in further specifying the Part A Arrive module the parameters Route = yes, StNm = yes and Route Time = 2 are defined and the Station = Part A Prep. Since these are parameters of the Part A Arrive module the latter parameters are actually defined as slots of the instance Part_A_Arrive, thus: Part_A_Arrive.Route, Part_A_Arrive.StNm and so on. Other parameter values are also determined, such as the distribution functions of the interarrival times and of the processing times, etc. Then a query could be formulated on the database of implementation models:

```
a = 2 AND s = 2 AND i = 2 AND d =3 AND A1.Route = yes
AND A1.StNm = yes AND A1.Route_Time = 2 AND
A1.Station = S1 AND ……
```

If the desired model is in the database, i.e. if the database contains a model with the same set of parameter values, then the corresponding model will be included in the dynaset. If not, the dynaset will remain empty. Nevertheless, the database could contain a model that is very much similar to the one that has to be designed. In that case the expert system could try to find a model that is in a way close to the model that the modeler is after. Then interpolation of models could be a way out.

3.4 Interpolation and Extrapolation of Models

Two models will rarely be the same. If we look at the model outlined in Figure 1 many minor differences will make the model appear differently. For example, if the various modules have different distribution functions, but the other parameters are the same, the two models would probably be seen as similar. On the other hand, if the modeler is after a model such as in Figure 1, a model such as in Figure 2 would not be seen as similar because it is obvious that these two models are different. It is up to the expert system (and actually to the makers of the knowledge base) to decide which models in the database will be considered as close to the desired model. This part of knowledge-based has to be looked at much more closely in a later stage.

4 CONCLUSIONS

In this article an approach is discussed that is intended to make it possible that models that are once built could be retrieved afterwards. Essential is that existing models are stored in a database in a parameterized way. Based on input from the designer an expert system translates information of the desired model into a set of parameters that can be used to retrieve a corresponding model from the database. More often than not the desired model will not exactly be present in the database and then interpolation or extrapolation of existing models could be an option. Especially in this case an expert system could do an excellent job. For both tasks business rules have to be formulated according to the considerations discussed and experiments have to be carried out.

REFERENCES

De Swaan Arons, H. 1983. Expert systems in the simulation domain. *Transactions IMACS: Mathematics and Computers in Simulation* 25-1: 10 - 16.

Platinum Technology, 1996. Developing Applications with the Aion Development System. *Student Guide*, version 2.0 (course material related to AionDS version 7.0)

Kelton, W.D., R.P. Sadowski, and D.A. Sadowski. 1999. *Simulation with Arena*. Boston: McGraw-Hill.

Pegden, C.D, R.E. Shannon, and R.P. Sadowski. 1995. *Introduction to Simulation Using SIMAN*. New York: McGraw-Hill Inc.

Toolkit 1997. Logistic problems and simulation models. TNO Inro, Internal report 97/NL/112, (in Dutch).

Taylor ED 1998. User manual. Utrecht: F&H Simulation BV.

AUTHOR BIOGRAPHY

HENK DE SWAAN ARONS graduated in Applied Mathematics at Delft University of Technology in 1972. From that time till 1982 he was appointed lecturer at the Faculty of Mathematics and Informatics of this university with teaching and research in the field of parallel computation, modeling and simulation. Since 1982 he concentrated on research and teaching in the field of expert systems. He was the project leader of several research projects under which two Esprit projects on knowledge-based scheduling in manufacturing. In 1991 he got his Ph.D. degree in Computer Science at Delft University of Technology. The thesis was mainly concerned with the design, applicability and applications of expert system tools, in particular the Delfi3 system. Since 1992 he is an associate professor at the Department of Computer Science of the Faculty of Economics at Erasmus University Rotterdam. At present, his research focuses on simulation and expert systems, with the emphasis on economical applications.