

PROTOTYPING IN PROCESS ORIENTED MODELING AND SIMULATION

Jaap A. Ottjes, Hans P.M. Veeke
Sub Faculty of Mechanical Engineering and Marine Technology, Fac. OCP
Delft University of Technology
Mekelweg 2, 2628 CD Delft, the Netherlands
E-mail: J.A.Ottjes@wbmt.tudelft.nl, H.P.M.Veeke@wbmt.tudelft.nl

KEYWORDS

Discrete simulation, process-oriented, prototyping, logistics, modeling

ABSTRACT

This paper presents a method that can be used to design process-oriented discrete simulation models in a fast and flexible way. To that end an informal object oriented Process Description Language has been elaborated and an example of this is shown. In the modeling, the concepts of time management and concurrency of processes are included. A particular view on the process-interaction method in which the active elements are taken to be entities that do the processing, is discussed and applied. The method is intended for complex modeling purposes.

INTRODUCTION

When modeling existing or future complex logistic systems, for example harbors or production plants, it is essential to ensure that the model gives an adequate representation of the real system. Before deciding to use simulation usually a lot of work must be done. A very important component of this preliminary work is a functional analysis of the system under study. Such an analysis, leading to a process oriented view on the system, is very compatible with the process interaction method (Zeigler et al. 2000), but gap still remains between the intuitive process model and the final formal model implementation. Even if software that supports process modeling and predefined components is used a large programming effort is still needed to complete complex models. In practice this leads to the delay of the project, to model faults and to unnecessary model rigidity. Moreover, during modeling, and particularly during use of the model, advancing insight leads to new ideas and questions arise with respect to model adaptations. These too demand a flexible and rapid method of model prototyping.

In this paper in order to overcome the difficulties mentioned above an informal process description language (PDL) is proposed. The PDL is evolved in modeling practice and has

been used in the development of various models already (Duinkerken et al, 2001 and Veeke and Ottjes, 2002).

After positioning the process interaction approach in worldviews on discrete-event simulation, we discuss the process interaction view by taking an example of intuitive prototyping. Then a process description language is elaborated and applied to an example. The method has been introduced and tested in simulation education and is currently in use in research projects and consultancy practice.

WORLDVIEWS ON DISCRETE-EVENT SIMULATION

All discrete event simulations contain an executive routine for the management of the event calendar and simulation clock, i.e., the sequencing of events and driving of the simulation. This executive routine calls the next scheduled event, advances the simulation clock and transfers control to the appropriate routine. The operation routines depend on the worldview, and may be based on events, activities, or processes (Hooper 1986), (Overeinder 2000).

Event scheduling

In event scheduling each type of event has a corresponding event routine. The executive routine processes a time-ordered calendar of event notices to select an event for execution. Event notices consist of a time stamp and a reference to an event routine. Event execution can schedule new events by creating an event notice and placing it at the appropriate position in the calendar. The clock is always updated to the time of the next event, the one at the top of the calendar.

Activity scanning

In the activity scanning approach a simulation contains a list of activities, each of which is defined by two events: the start event and the completion event. Each activity contains test conditions and actions. The executive routine scans the activities for satisfied time and test conditions and executes the actions of the first selectable activity. When execution of an activity completes, the scan begins again.

Process interaction

The process interaction worldview focuses on the flow of entities through a model. This strategy views systems as sets of concurrent, interacting processes. A process class describes the

behavior of each class of entities during its lifetime. Process classes can have multiple entries and exits at which a process interacts with its environment. The executive routine uses a calendar to keep track of forthcoming tasks. However, apart from recording activation time and process identity, the executive routine must also *remember* the state in which the process was last suspended (Overeinder 2000). Especially the object-oriented approach appears to be suitable for process interaction simulation, (Robert, 1998).

Zeigler (Zeigler et al. 2000) describes process interaction simulation as a combined event scheduling-activity scanning procedure. The description of the dynamics of a model element can be implemented as a unit, rather than being separated into a number of unconnected events and activity routines. Therefore the program structure maintains a closer relation to the model structure and consequently the modeled real system. The state change of each element during its life cycle is described as a sequential program: its *process*. Time consuming statements in this program cause an interrupt, which make the element pause until a specified time advance has elapsed or until a specified condition becomes true. This mechanism puts special demands on the simulation software.

The process interaction worldview breaks down into two views corresponding to a different assumption as to what are active and passive elements in systems to be modeled, (Zeigler et al. 2000). In the prototypical *process interaction* worldview the active elements are taken to be entities that do the processing, e.g. the machines, servers etc. In the second approach the active elements are the flowing elements, such as customers, work pieces, packets, etc. The serving elements are called resources. This "flow oriented" approach is often referred to as the *transaction* worldview. Fishman (Fishman, 2001) mentions that the concept of active processes of flowing elements and passive resources is applied in most simulation programming languages that are based on the process interaction approach. He further argues that having the opportunity to also describe processes of active permanent elements increases the flexibility of modeling. The first simulation language that allowed both approaches in an object oriented way was "Simula", (Birtwistle and Dahl, 1973).

In this paper we will base ourselves on the process interaction approach as defined by Zeigler.

THE PROCESS INTERACTION APPROACH

Applying the process-interaction modeling can be summarized in three steps:

Step 1: decompose the system into relevant classes of elements, preferably patterned on the real-world elements of the system. It is obvious that the results of the functional system analysis may play an important role in choosing the classes.

Step 2: Identify the attributes of each element class. A class is characterized by its attributes. An instance of a class will be called an element. The state of each element is defined by the state or value of its attributes.

Step 3: distinguish the "living" element classes and provide their process descriptions. A process governs the dynamic behavior of each element. An element cannot do two things "at

a time". If that appears to be the case after the decomposition, the element class should be split up further until only "single" processes remain. These single processes are also called "leaf processors" (Wortmann A.M., 1991) We will use the term "**process**" for a single process. Again the results of the functional analyses may help. In order to illustrate the process modeling, we will elaborate an example of a simple job shop in an "intuitive" way just to

The job shop: An intuitive process model.

If the working of a system is known it is relatively easy to make an intuitive process model. Applying the 3 building rules the job shop process model would look like this:

The element classes are *machines* contained in *machine groups* and *jobs* containing a set of *tasks*, each with its own execution time on its machine group. Further we need a *generator* of jobs. The active elements are the machines and the job generator, the latter representing the model environment.

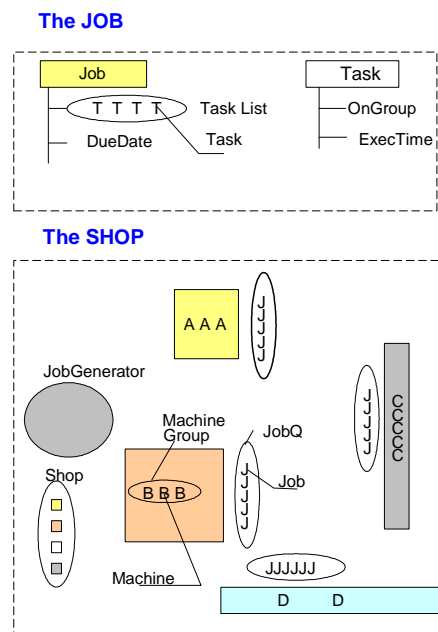


Figure 1. The Job Shop schematically. Jobs consist of a sequential list of tasks to be executed. The machine groups contain a number of identical machines. Each group has a job queue with jobs waiting to be executed.

Process of the job generator:

Wait until next job arrives (this statement consumes system time)
Create a new job with its tasks
Pass the job to the machine group of its first task
Repeat this process

Process of a machine:

Wait for jobs to be processed by your group (this may consume system time)
Take a job from the jobs that are waiting for this group
Execute the relevant task of that job (this statement consumes system time)
If the job is ready, put it in the 'ready buffer'
Else pass the job to its next group

Repeat this process

Model control:

Create the machine groups with their machines and start the machine processes

Create the job generator and start its process

Wait the simulation runtime

Present results

In the processes of the active classes “time consuming” statements appear. These are statements that “consume” system time.

A time consuming statement expresses a period of action or waiting of the element. Because time periods used by several active elements may overlap, a sequencing mechanism is needed to synchronize the activities and to manage the event calendar. This mechanism must be supported by the simulation package that is used. If the process interaction modeling method is applied, it is desirable that the formalization (coding) of the model links up with the process approach. Still there remains a gap between the informal intuitive way of describing the model as we just showed and a formal implementation in a process oriented simulation language. Therefore now a ‘Process Description Language (PDL)’ will be presented to informally describe systems in a process oriented manner. The purpose of this is to provide a method to be used to design and formulate a process oriented model close to formal implementation in a flexible way. In this stage irrelevant constraints and formalisms of a specification language or simulation language are not considered. Nothing should hinder flexibility and creativity of designers; moreover the model should still be accessible to a broad audience.

A PROCESS DESCRIPTION LANGUAGE

For a PDL special concepts are needed. The most important concept however must be in the mind of the modeler: He/she should think in terms of processes!

Two other important concepts form a basis for the process modeling and for the PDL: *Time management* and *concurrency of processes*.

Time management

In the real world a process consists of a chain of activities. Depending of the goals of the modeling some of these activities take time, for example *Execute Task*. In the context of a specific model other activities may be considered to be timeless, for example *switch the light on*. In the PDL we will model the time consuming behavior by introducing a set of “time consuming” clauses. These clauses appear in the process descriptions.

Concurrency

When several processes are running parallel in time and interact, these processes should run synchronized. As it is assumed that this synchronization is taken care of by the software to be used, the process modeler has to focus only on the process and its interactions.

Elements and Sets

A Process is executed by an element, for example the driver of a car. Moreover there may be a number of elements, for example drivers, behaving according one specific process description. Therefore we need the concept of *classes of elements*. We define one basic element class that owns all specific features needed in our process-modeling concept.

We call that class ‘*SimElement*’

An element of class **SimElement** owns certain properties and methods. In terms of object oriented programming it is the parent class of any model element class we need in the process model. For simplicity reasons we call the set of properties and methods of an element class its **attributes**. Every model element class inherits all attribute types of **SimElement** class, so the process modeler may assume the presence of these attributes as soon as an element instance is created. In addition, specific model elements descending from **SimElement**, for example a driver class, may have its additional -class specific- attributes, for example its speed.

The **SimElement** class needs certain process related attributes:

Class **SimElement** with the following methods as attributes:

Process related methods

- Process* Describes the process of the element class
- Start(t)* |
- Interrupt(t)* |
- Resume(t)* | t means “at time t”
- Proceed(t)* |
- Cancel* |
- Finish*
- Create/Arrive/Generate* creates the element in the model as an instance of its class
- Destroy/Depart/Deliver* removes the element from the model

Further features of **SimElement** are methods related to ‘sets’ in which a **SimElement** may reside

Set related methods of **SimElement**

- Enter(Set, Location)* Location is in (Head, Tail, After(), Before(), SortedOn())
- Leave(Set)*
- IsIn(Set)*
- Successor(Set)*
- Predecessor(Set)*
- SetTime(Set)*

Any model element class inherits the attributes and methods of the **SimElement** and, in addition to that, it may have its own class specific attributes and methods. The inherited Process method has to be provided by the modeler for each active model class.

Class **Set**

A second class that is very important in our PDL is the **'Set'** class. A Set is a collection of model elements and owns its set-attributes. Sets are extremely useful in the control part of a model. Control decisions often come down to choosing the correct element in a Set. For example the assignment of a transport vehicle to a load could be achieved by combining the correct vehicle from a Set of vehicles with a specific load from a Set of loads. Often the main goal of a simulation project is the development and validation of adequate decision algorithms.

Sets may or may not be ordered. If a Set is used as a traditional waiting queue, one is normally interested in statistics of waiting times in and length of the Set. In that case the Set may have the name 'Queue' as an alias.

Next some important attributes of the Set class are listed. These attributes are partly complementary to the Set related attributes of SimElement

Attributes of Class Set

- Create
- Destroy
- Length
- Add(Element, Location) ;Location is in (Head, Tail, After(), Before(), SortedOn())
- Remove(Element)
- First
- Last
- Successor(Element)
- Predecessor(Element)
- Contains(Element)
- IsEmpty
- IsNotEmpty
- MeanLength
- MaxLength
- MinLength
- MeanTime
- MaxTime
- MinTime

Process description clauses related to system time

"Now" indicates the current time in the system. In a model or a distributed model there is only one common time.

Time-consuming clauses are indicated with **'Advance'**.

Advance indicates three types of time intervals:

Advance(*t*): time scheduled interval; process continues after time *t* has elapsed

Advance(condition): state scheduled interval; process proceeds as soon as condition becomes false

Advance: indefinite interval; process has to be resumed from another process

Example 1: **Advance(10)**. //wait for 10 time units; then resume process

Example 2: **Advance**(until traffic light is green). //wait until traffic light becomes green; then resume process

If it would clarify the model description, aliases of **Advance** may be used. They have to be announced at the start of the model, for example: **Advance**=(Work, Wait, Drive, Hold, Suspend, Standby,...)

Process control.

In real world processes are either repetitive or finite. In order to model this behavior properly the next process control clauses may be used.

Loop repeat indefinitely
Loop(n) repeat n times
Loop(condition) repeat while condition is TRUE

Examples:

Loop(SetX.IsNotEmpty) // Empty a Set

1. **SetX.Remove(SetX.First)**

Miscellaneous statements.

The 'if' clause

1. If (condition)
 - 1.1. Statement
 - 1.2.
 - 1.3. Statement
2. Else
 - 2.1. Statement
 - 2.2. ...
 - 2.3. Statement

For file IO, the following clauses are used:

Read // a value (of arbitrary type) to be read from some data source

Write // a value (of arbitrary type) to be written to some destination file.

Distributions

Distribution can be any theoretically well-defined distribution type (uniform, exponential,...) or tabularized or discrete distribution. Depending on the type a set of parameters defining the distribution must be added.

Sample(Distribution) // sample from an arbitrary or a specific distribution.

The Process Model

A *Process* model consists of three sections:

- A *definition section* to describe the element classes and their attributes.
- A *process section* containing the *process* descriptions of the active element classes.
- A *model control section* to initialize the model and to control simulation runs.

The Definition section

To define the elements of a model the following definition clauses are used:

- ChildClass: (ParentClass) to define an object class, derived from a parent class

Examples:

CShip: (**SimElement**)

CContainerShip: (Cship)

CSuperTanker: (Cship)

CShip is defined as a child class of the **SimElement** class, CContainerShip and CSuperTanker are defined as child classes of CShip.

- ClassReference: Class; to define a variable referring to an element.

Example:

```
MyShip      :Cship
TheShip     :Cship
Mammoth    :CSuperTanker
MyFleet    :Queue
```

MyShip, TheShip, Mammoth and MyFleet must be assigned names and values, that refer to "instances" of the class CShip, CSupertanker and Queue respectively. Remember: a class definition is nothing more than a blueprint of an object; an instance is one physical individual (in computer memory) according to the blueprint.

An instance is created by the class-method 'Create', 'Arrive' or 'Generate' and the method generates a reference to the individual. So the following is a correct description:

```
MyShip      = CShip.Create
TheShip     = MyShip
```

After this sequence MyShip and TheShip refer to the same individual.

Attributes of a class may be of any type.

For qualifying of attributes (i.e. expressing which instance is the owner of the attribute) the "dot" notation may be used.

Example: "MyShip.Length" means the length of MyShip.

If readability is improved also "Length of MyShip" may be used.

For all attributes a further explanation is advisable. The explanation should be preceded by a //

- Attributes of an instance are assumed to exist at the moment the instance is created. Attributes, that are class references, state the referred existing object explicitly by means of an assignment or have to be assigned at creation of the object instance (see example above).

- Attributes of value types only need to be specified if this supports the readability of the model description. For numerical and string values a specification is not needed. The name of an identifier always should explain its meaning or function.
- Attributes may also be class methods (function or procedure). These methods also have to be elaborated in process language if this elucidates the process model. This may be especially the case if a method contains control functions.

The Process section

In this section the process descriptions are given for all process element classes defined in the definition section. Each process description is headed by the process element class name followed by ".Process"

A process description can use all methods described earlier

The model control section

This section is meant to initialize the model and to control simulation runs with the model. It is a special *Process* description of the overall 'model element' and all clauses of the *Process* section are available here. Mostly the model control section will be used to create and start the "permanent" elements of the model and, in case of a distributed model, to connect to the distributed environment.

Distributed modeling

If the model is to be distributed consisting of sub- or member-models running synchronized on different computers, then the process model presented here does not change in essence. (Ottjes and Veeke, 2001). Synchronization of the sub models is a technical matter that does not influence the process model.

THE JOB SHOP CASE

To conclude a complete example of a job shop model of figure 1 is described in PDL and given below and part of the implementation in the simulation package Tomas is shown. The process model of a job shop:

Definition Section

MachineGroup: (**SimElement**)

- MachineSet:Set //contains the Machines of the group
- IdleSet:Queue //represents the set of all idle machines
- JobQ:Queue //contains the Jobs to be executed by the group

Machine: (**SimElement**)

- MyGroup:MachineGroup //refers to the group the machine belongs to
- MyJob:Job //refers to the Job assigned to this machine
- MyTask:Task //refers to the task in execution by this machine

- ❑ SelectJob: Method // selects Job from MyGroup.JobQ
- ❑ **Process:** Method

JobGenerator: (**SimElement**)

- ❑ GJob:Job // refers to the instance of job being generated
- ❑ GTask:Task // refers to the instance of task being generated
- ❑ DueDateDistribution // distribution of due dates
- ❑ **Process:** Method

Job: (**SimElement**)

- ❑ TaskList: Set // contains all tasks being part of the job
- ❑ DueDate // required Job completion time .

Task: (**SimElement**)

- ❑ OnGroup:MachineGroup // the machine group the task is to be executed on
- ❑ ExecTime // execution time of a task

Process section

JobGenerator.Process

Loop

1. **Advance**(Sample('interarrival time of jobs'))
2. GJob = Job.Generate
3. GJob.DueDate = sample('DueDateDistribution')
4. Loop(Sample('number of tasks per job'))
 - 4.1. GTask = Task.Generate
 - 4.2. GTask.OnGroup = Sample('group to be executed on')
 - 4.3. GTask.ExecTime = Sample('execution time')
 - 4.4. GTask.Enter(GJob.TaskList)
5. GJob.Enter(TaskList.First.OnGroup.JobQ)

Machine.Process

Loop

1. Loop(MyGroup.JobQ.IsNotEmpty)
 - 1.1. MyJob = SelectJob
 - 1.2. MyGroup.JobQ.Remove (MyJob)
 - 1.3. MyTask=MyJob.TaskList.First
 - 1.4. MyJob.TaskList.Remove (MyTask)
 - 1.5. **Advance**(MyTask.ExecTime)
 - 1.6. If MyJob.TaskList Not Empty
 - 1.6.1. MyJob.Enter(MyJob.TaskList.First.OnGroup.JobQ)
 - 1.7. else
 - 1.7.1. MyJob.Free //this job is ready
2. Enter(IdleSet)
3. **Advance**(MyGroup.JobQ.IsNotEmpty)
4. Leave(IdleSet)

Machine.SelectJob :Method

- ❑ Jobx :Job // variable referring to Job

 1. Jobx = MyGroup.JobQ.First
 2. Return Jobx

OR

Machine.SelectJob :Method

- ❑ Jobx :Job // variable referring to Job

 1. Jobx = Job in MyGroup.JobQ with smallest DueDate
 2. Return Jobx

Model Control section

MAIN

1. Loop(Read number of machine groups)
 - 1.1. MachineGroup.Arrive
 - 1.2. Loop(Read number of machines in group)
 - 1.2.1. Machine.Arrive
 - 1.2.2. Machine.Enter(MachineSet)
 - 1.2.3. Machine.Start(Now)
2. Generator.Arrive
3. Generator.Start(Now)
4. **Advance**(20000)
5. Finish

Model Implementation

If, after completion of the process model, it is decided to really implement the model in a computer model, a simulation language is to be used, preferably a language that supports process interaction modeling. (Healy, 1997), (Veeke, Ottjes, 2000). As an illustration here some important parts of the implemented job shop model in Tomas (Veeke, Ottjes, 2002) are shown. In Tomas the SimElement class is called TomasElement and the Set class is called TomasQueue. Further in Tomas Advance(t) is coded as Hold(t) and Advance(Condition) is coded as: **While (Condition) Do Standby;**

//Definition of Classes MachineGroup, Machine,
//Job and Task in Tomas

```
TMachineGroup = class(TomasElement)
  GroupNr :integer;
  MachineSet:TomasQueue; //contains Machines of the group
  IdleSet :TomasQueue; //set of idle machines
  JobQ :TomasQueue; //contains Jobs for the group
  Published
  procedure writeToLate;//register difference of due date and real
  completion time
end;
```

```
TMachine = class(TomasElement)
  MyGroup:TMachineGroup; // refers to its group
  MyJob :TJob; // selected Job
  MyTask :TTask; // the task in execution
  Published
  Function SelectJob:Tjob; // selects Job from MyGroup.JobQ
  Procedure Process; override;
end;
```

```
TJob = class(TomasElement)
  TaskList: TomasQueue;// contains all tasks of the job
  DueDate :double; // due date of job
end;
```

```
TTask = class(TomasElement)
  OnGroup:TMachineGroup; // machinegroup of the task
  ExecTime:double; // execution time of a task
end;
```

//Machine process in Tomas

```
Procedure TMachine.Process;
begin
  while true do
  begin
    while(MyGroup.JobQ.Length>0) do
    begin
```

```
MyJob :=SelectJob;
MyTask:=MyJob.TaskList.FirstElement;
MyJob.TaskList.Remove (MyTask);
MyGroup.JobQ.Remove (MyJob);
Hold(MyTask.Exectime);
if MyJob.TaskList.Length>0 then
begin
  MyJob.EnterQueue
  (MyJob.TaskList.FirstElement.OnGroup.JobQ);
end else
begin //MyJob is ready
  MyGroup.WriteToLate(Tnow-MyJob.DueDate);
  MyJob.Free;
end;
end;
EnterQueue(MyGroup.IdleSet);
while (MyGroup.JobQ.Length=0) do
begin
  Standby;
end;
LeaveQueue(MyGroup.IdleSet);
end;
end;

//Select job method: First in First out:
Function TMachine.SelectJob:Tjob;
begin
  SelectJob:=(MyGroup.JobQ.FirstElement);
end;
```

CONCLUSIONS

An informal object oriented Process Description Language has been elaborated to be used to design and adapt process-oriented discrete simulation models in a fast and flexible way. In the modeling, the concepts of time management and concurrency of processes are taken into account. In order to increase modeling flexibility the process-interaction method has been interpreted in a way that the active elements are taken to be entities that do the processing. The advantage of this prototype-modeling step is that the modeler can concentrate fully on the content of the system to be modeled and that the model remains accessible to systems experts. The approach is especially beneficial if in the final coding stage simulation software is used that supports process interaction modeling. The method is now being applied in modeling large-scale logistic systems and is part of logistic simulation courses.

REFERENCES

- Birtwistle, G. M., O. J. Dahl, B. Myhrhang, K. Mygrard (1973), *Simula Begin*, Van Nostrand Reinhold, New York.
- Duinkerken, M.B, J.J.M. Evers; J.A. Ottjes. 2001." A simulation model integrating quay transport and stacking policies on automated container terminals". *Proceedings of the 15th European Simulation Multiconference*. Prague (SCS) pp 909-916.
- Fishman, G.S. 2001. *Discrete Event Simulation. Modeling, Programming, and Analysis*. @001 Springer-Verlag New York, Inc. ISBN: 0-387-95160-1, 52-59
- Healy, J. R.A. Kilgore, 1997. "Silk,: A Java-Based Process Simulation Language". *Proceedings of the 1997 Winter Simulation Conference, IEEE*,
- Hooper, J. W, 1986. Strategy related characteristics of discrete-event languages and models. *Simulation* 46(4).

Ottjes J.A., Veeke H. P.M, and Buizer A.A, 2001: Experimenting with distributed modeling and simulation using the internet. *Proceedings of 15th European Simulation Multiconference (ESM2001)* pp. 909-916. June 6-9, 2001 Prague, Czech Republic. ISBN:1-56555-225-3

B. J. Overeinder (2000), Distributed Event-driven Simulation – Scheduling Strategies and Resource Management, ASCI dissertation series no. 58, Amsterdam.

Robert, C.A., Dessouky, M, 1998. "An Overview of Object-Oriented Simulation". *Simulation* vol:70:6, pp. 359-368. 1998.

Veeke, Hans P.M., Jaap A. Ottjes, 2000. Tomas: Tool for Object-oriented Modeling And Simulation. *In proceedings of Advanced Simulation Technology Conference (ASTC2000)*. April 16-20, 2000, Washington, D.C. pp. 76-81. The Society for Computer Simulation International (SCS), ISBN: 1-56555-199-0

Veeke, Hans P.M., , Jaap A. Ottjes, 2002 "A generic simulation model for systems of container terminals ", Submitted to the *16th European Simulation Multiconference ESM 2002*, Darmstadt.

Veeke, Hans P.M., Jaap A. Ottjes, 2002. TomasWeb: web site: www.tomasweb.com

Wortmann, A.M. 1991. *Modeling and Simulation of Industrial Systems*. Dissertation Fac. Of Mechanical Engineering, Eindhoven University of Technology (1991). ISBN 90-72015-91-6

Zeigler B.P., Praehofer H and Kim T.G., 2000. "*Theory of Modeling and Simulation* 2nd Ed. Academic Press, San Diego