LOGISTICS: A SYSTEMS ORIENTED APPROACH

W.M.P. van der Aalst
Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
e-mail: wsinwa@win.tue.nl
phone: -31 40 473637
fax: -31 40 442150

KEYWORDS: Logistics, Dynamic modelling, Petri nets, Formal specification.

ABSTRACT

A framework for the modelling and specification of logistic systems is presented. This framework is based on a hierarchical high-level Petri net model. Within this framework we developed a language, called ExSpect, to specify systems in terms of this Petri net model. This specification language is supported by a software package, also called ExSpect, which facilitates the modelling and analysis of complex dynamic systems.
We have modelled and analysed many logistic systems with ExSpect ([5]). Most of these applications resulted from the TASTE project ([3]). These practical experiences show that there is a need for a method to facilitate the modelling of complex logistic systems. We propose a systematic method, which is presented in this paper. This method is based on a complete taxonomy of the goods and information flows inside a logistic system. Based on this taxonomy we have developed a toolkit of standard logistical components that can be combined graphically, thus yielding an ExSpect specification which can be used to analyse the logistic system under consideration.

1 INTRODUCTION

High-level Petri nets have been used in many application areas, e.g. flexible manufacturing, computer architecture, distributed information systems, protocols, etc. In [12] there are a number of papers describing applications of high-level nets. ExSpect is a specification language based on a timed high-level Petri net model. ExSpect also has a hierarchical construct called *system*. This construct can be used to structure large and complex specifications, e.g. we describe a complex system in terms of its subsystems.

Although our specification language is very powerful, it is difficult to model large and complex logistic systems. In this paper we propose a method to facilitate the modelling of logistic systems. This method is based on principles of systems analysis and experiences gained in the *TASTE* project.

The TASTE (The Advanced Studies of Transport in Europe) project started in 1986, when the Dutch organisation for Applied Scientific Research (TNO) established a fund for a systematic approach in the field of transport and logistics. The goal of the TASTE project is to develop a tool to enable non-programmers to model and analyse problems in the field of interindustrial logistics. The project emphasizes the modelling and analysis of the flow of goods at an aggregated level in and between, production, assembly, distribution and transport. The aim of the tool is to support strategic decision making. A number of pilot projects showed that simulation is the most likely analysis method because of the complexity of a real logistic system. Furthermore, the pilot projects showed that existing simulation tools are either to specific (special purpose simulation packages) or to general (general purpose simulation languages). Since ExSpect combines the advantages of special purpose simulation packages and general purpose simulation languages, TASTE started to use ExSpect in 1988. See [3] for more information.

The TASTE project faced the fact that research in the field of logistics developed along two separate lines.
The first line concentrates on solving mathematical problems related to logistics. Often the problem statement is simplified to allow for analytical solutions. This is the reason that many results in this area cannot be applied to real problems encountered in practice. Examples of this line are the application of queueing networks to scheduling problems and the application of linear programming to transport planning. Although these analysis methods help us gain insight in the problem, they can only be applied in a rather specific situations.
The second line of research concentrates on practical logistic problems. The results are often qualitative and informal. The approaches used in this area are mainly discipline oriented, i.e. they focus on a specific aspect of logistics. Examples are the research on customer service, storage equipment, communication facilities (EDI), personnel requirements, etc.

Both of these lines did not lead to a complete and comprehensive model of logistics. Recent literature in the field of production control stresses the need for a systematic approach to production planning and control (e.g. [6],[7]). In Biemans [7] an attempt was made to structure manufacturing planning and control using a 'reference model', i.e. a representation of an idealized production organization, defining the tasks of the components as well as the interactions between the components. In [6], Bertrand, Wortmann and Wijngaard provide a number of general concepts for the design of production control systems.
This paper presents a "systems view of logistics" to be able to deal with the growing complexity of the control problems in logistics. The growing complexity is a result of the "total cost concept", which forces us to consider the entire logistic chain. Another reason for the increased complexity is the progress in information technology allowing for more sophisticated management systems.
The systems view of logistics we have developed includes a complete taxonomy of the goods and information flows inside a logistic system. Based on this taxonomy we distinguish four types of systems: physical systems, information systems, control systems and mixtures of the previous three types. Other topics are: decomposition, aggregation and typical control structures. Furthermore, we show that for the modelling of large complex logistic systems we are in need of a hierarchical construct. ExSpect provides such a

construct.

Based on these observations, we have specified a number of generic *building blocks*. Building blocks are parameterized subsystems (subnets) representing a typical activity. There are about 20 of these building blocks including a production unit, a distribution centre and a transport system. It is our belief that many practical logistic systems can be modelled using these building blocks. This modelling process is supported by software (ExSpect) and results in a specification that can be analysed using simulation or one of the analysis methods described in [1] and [2].

2 A SYSTEMS VIEW OF LOGISTICS

A logistic process consists of the flow of goods and services and the monitoring and control of these flows. Typical activities include: transportation, inventory management, order processing, warehousing, distribution and production. *Logistic management* is concerned with the development of functions to support these activities. A simplified definition of logistics is: "The process of having the right quantity of the right item in the right place at the right time" (Hutchinson [10]).
In the area of logistics many books are available, nearly all of which deal with the control and design of production, inventory and transport systems.
Some of these books concentrate on solving mathematical problems related to logistics. Often the problem statement is simplified to allow for analytical solutions. This is the reason that many results in this area only yield partial solutions.
Nearly all other books concentrate on practical logistic problems. The results are often qualitative and informal.
To model and analyse large and complex logistic systems we need a complete and comprehensive model of logistics. Therefore we use a systems oriented approach to structure the field of logistics.
Another reason to present a systems view of logistics is the growing complexity of the control problems in logistics. The total cost concept, an approach to minimize the overall costs, forces us to consider the entire logistic chain. Another reason for the increased complexity is the progress in information technology allowing for more sophisticated management systems.

To clarify our approach we start we a number of concepts adopted from systems analysis. We define a *system* as a group of *elements* working in an interrelated fashion toward a set of objectives. These elements are the smallest parts to be considered, sometimes referred to as *entities* or *objects*. Each element can be characterized by the *relations* with its environment. Examples of elements are humans, machines, goods or information processing equipment. The systems boundary defines which "part of the (logistic) world" is considered and which part is out of scope. It is possible to compose a number of systems into a new system. It is also possible to decompose a system into a number of sub-systems. This process can be repeated until we reach the level of elements. A *closed system* is a system without any interactions with "some" environment. An *open system*, however, has a certain (external) interaction structure. Note that it is always possible to construct a closed system from an open system by explicitly modelling its environment. This is expressed in figure 1.

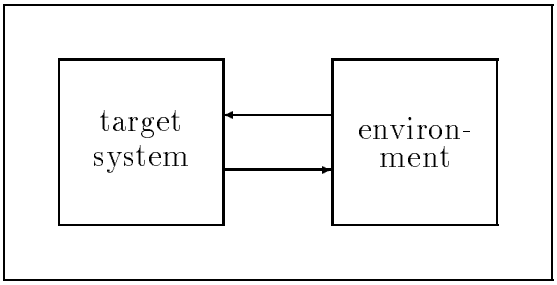

Figure 1: *A closed system composed of the target system and its environment*

Systems are represented by rectangles. We use arrows to denote relations between systems. Nearly all "real-life" systems are open. Consider for example a human-machine system, i.e. a person interacting with a machine. From a modelling point of view we can consider such a system a closed system. This is often useful for analysis. Yet, the human needs food and beverage and the machine needs electricity and maintenance. Note that the environment of a system can only be defined after the system boundary has been defined.

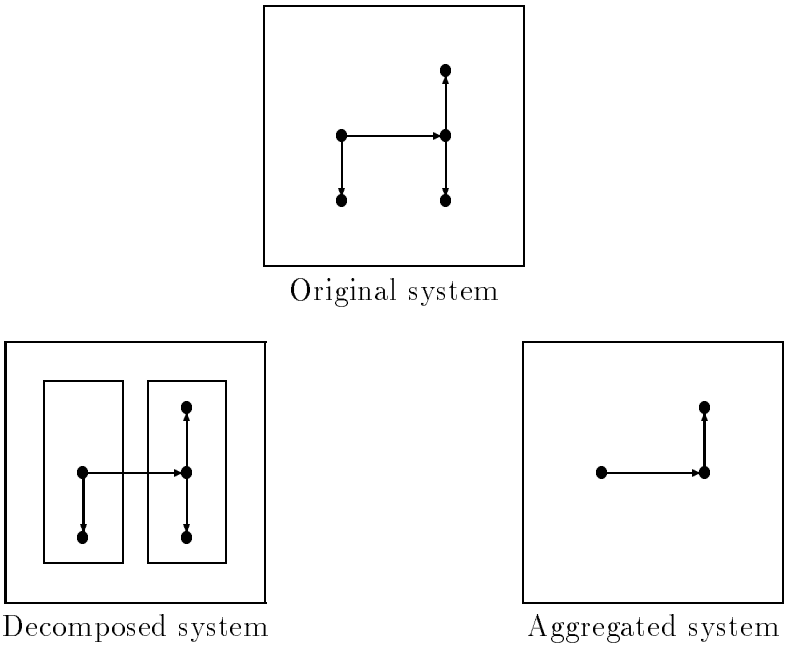

Figure 2: *The difference between composition and aggregation*

There are a lot of ways to *decompose* (*compose*) a system into (from) a number of smaller subsystems without changing the set of elements (entities). Decomposition is a way to deal with the complexity of system, because it allows for the consideration of only a small part of the system at the same time. The level of abstraction remains the same, because the set of entities is not changed. If a system X is decomposed into a number of subsystems $X_1, X_2, ..X_n$, then the proper composition of $X_1, X_2, ..X_n$ yields the original system. If we use another set of elementary objects (elements) to model a system we speak about *aggregation* (*disaggregation*) rather than composition (decomposition). An

alternative term for aggregation is abstraction, i.e. an aggregation step decreases the level of detail. Figure 2 shows the difference between composition and aggregation. Note that the decomposed system is equal to the original system. However, the aggregated system is different from the original system, because some of the details are omitted. There are a number of ways to decompose a system:

Functional decomposition : a system is split up in a number of subsystems each representing a set of related activities.

Spatial decomposition : a system is decomposed into a number of subsystems at different locations.

There are also a number of ways to disaggregate a system:

Functional disaggregation : some functional part of the system is modelled in more detail.

Spatial disaggregation : some geographical part of the system is modelled in more detail.

Disaggregation of an aspect : some aspect of the system is modelled in more detail. For a logistic system we can disaggregate the physical aspect of the goods flow. In this case we add, for instance, weight and volume to the description of the product.

Disaggregation of time : the dynamical behaviour of a system is modelled more precisely. For example, we model the state of a system every hour instead of every day. Note that the timescale has changed.

We can use the same classification for composition and aggregation respectively.

Several methods to develop a model of a system have been proposed. *Top down* development starts with a model at a high abstraction level, this model is refined by a number of disaggregation steps until the desired level of detail has been reached. To deal with the increasing size and complexity of the model a disaggregation step often coincides with a decomposition step. *Bottom up* development starts with a model for each of the subsystems. These models are detailed descriptions of some aspect or part of the systems, i.e. they have a low abstraction level. These submodels are composed into a model of the entire system. If the overall model becomes to complex, an aggregation step is applied to abstract from some of the details. Pure top down development is often impractical. Pure bottom up development would be a mess. In our opinion, a mixture a top down and bottom up development is the most sensible way to build a model.

The elements in a system interact with each other via relations. Such a relation is directed or undirected. If a relation is directed one element influences the other but the reverse does not happen. An undirected relationship can be represented by two directed relations. In this paper we use the terms *flow* and *channel* instead of relation because these terms are more suitable in the field of logistics. Note that there is a flow between two subsystems A and B if and only if there is a flow between some element of A and some element of B.

To structure the field of logistics we start with a classification of the flows inside a logistic

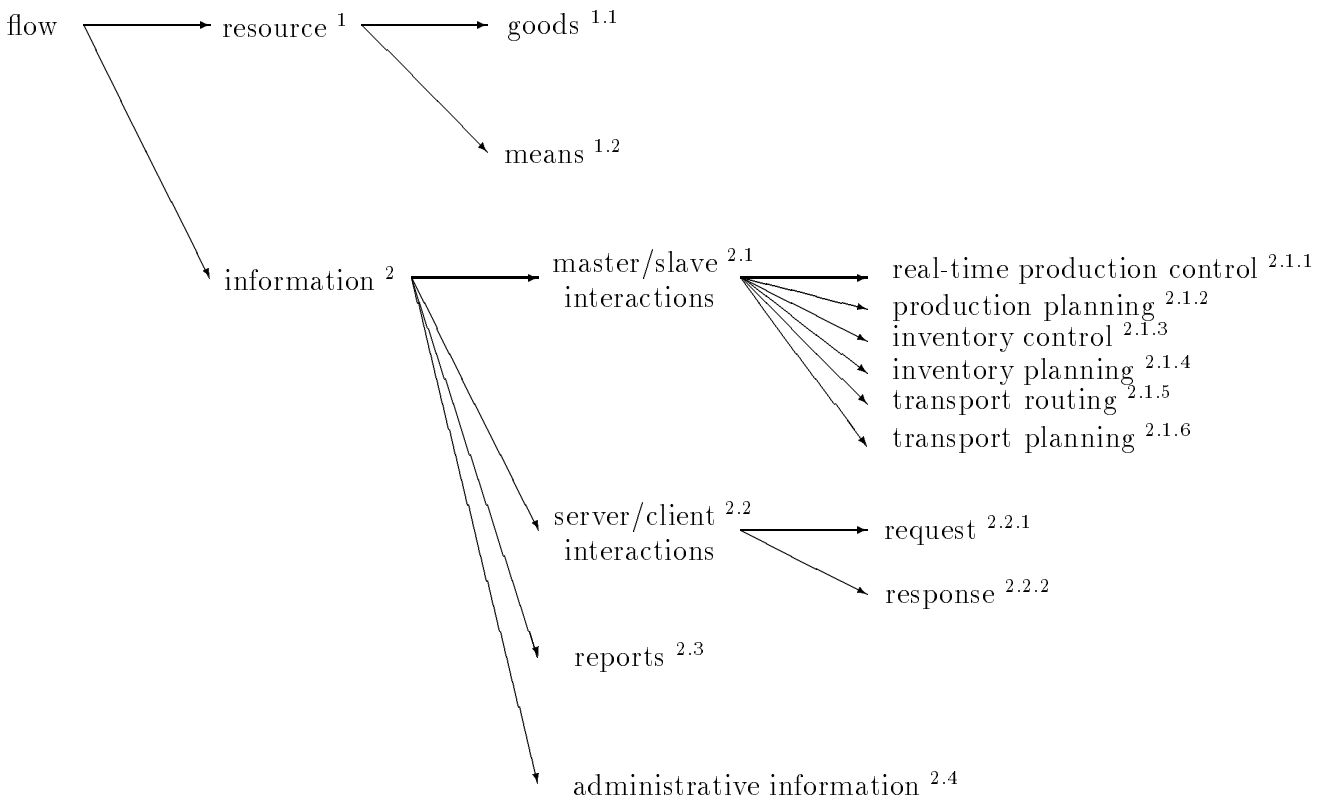

Figure 3: *A taxonomy of the flows inside a logistic system*

system. Figure 3 shows our taxonomy, the arrows should be interpreted as "is subtype of". For example, the flow of goods is a subtype of the flow of resources.
Resources (1) are the physical or abstract objects in a system. We distinguish between *goods* (1.1) and *means* (1.2). Goods are materials, components and products in a logistic chain. In general these goods are physical objects. Examples of non-physical goods are bank accounts or reservations, we call these objects 'abstract objects'. The resources needed to create, maintain or distribute these goods are called means, e.g. machines, tools, trucks, manpower, etc. Means are employed , but not consumed like materials. Sometimes we use the term *capacity resources* to refer to these means. It is hard to draw a strict dividing line between goods and means, think for example of a tool in a machine that wears off significantly when it is used.
We use the term *information* (2) for all other kinds of interaction. Information can be characterized by: "all the messages needed to get the right quantity of goods at the right time at the right place". Information itself is not an object to pursue. In most cases information is kept to a minimum. We divide the class of information flows into four subclasses: *master/slave interactions* (2.1), *server/client interactions* (2.2), *reports* (2.3) and *administrative information* (2.4).
Master/slave interactions are the messages exchanged between a control system (master) and a subordinate system (slave). The master sends commands to the slave and the slave sends some status information to the master. Essential is the fact that their relationship is not based on equality. Examples of such interactions are: *real-time production control* (2.1.1), *production planning* (2.1.2), *inventory control* (2.1.3), *inventory planning* (2.1.4), *transport routing* (2.1.5) and *transport planning* (2.1.6). For the moment this classification

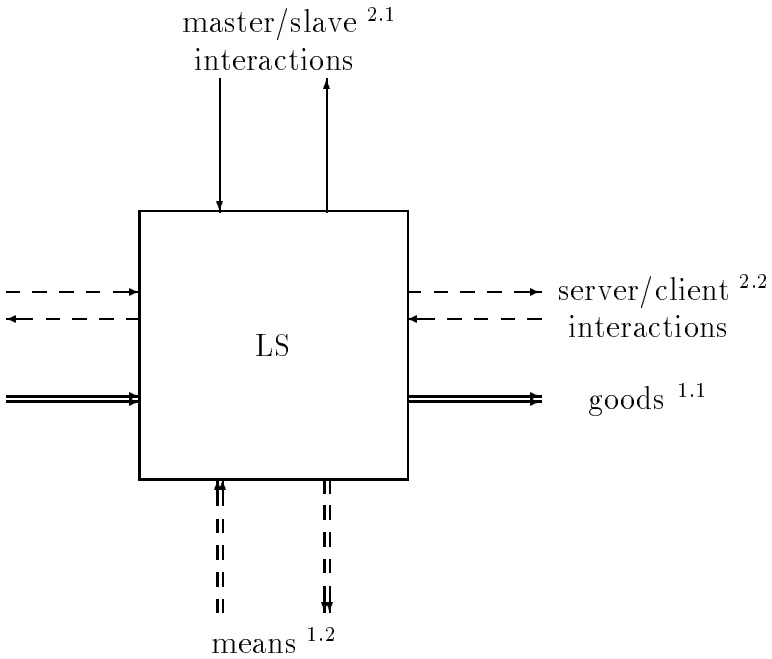

Figure 4: *A logistic system*

is self-explanatory. We will return to this subject in section 4.
Client/server interactions are based on the equality of both parties involved. An alternative term for client/server interactions is coordination. We have requests and responses instead of commands and status information. The client sends a *request* (2.2.1) to a server. This is always followed by a *response* (2.2.2) from the server to the client. There are two kinds of requests and responses, with and without a "commit". A request without a commit means that the client only inquires about some service or goods. Otherwise (with commit), the request is satisfied by the server if possible. In this case there is response with a commit indicating that the server will deliver the requested service or goods. In all other cases there is a response without a commit. Note that this classification conforms with the ideas emerging from the field of Electronic Data Interchange (EDI).
Finally, we have the flows of reports and administrative information. These are the information flows not covered by the flows (2.1) and (2.2). A detailed description is beyond the scope of this paper.

We introduce a graphical convention to denote these flows, flows of resources are represented by a double arrow and flows of information are represented by single arrows. To distinguish flows of means from flows of goods we represent flows of means by dashed double arrows. Client/server interactions are also represented by dashed arrows. All other kinds of information flow are represented by an ordinary arrow. Figure 4 shows these graphical notations. This concludes our taxonomy of the flows inside a logistic system. In section 4 we will show how to model these flows in terms of ExSpect types.

Figure 4 shows the general form of a logistic system. Such a system is composed of a number of subsystems. It is possible to repeat this process until the lowest level is reached.

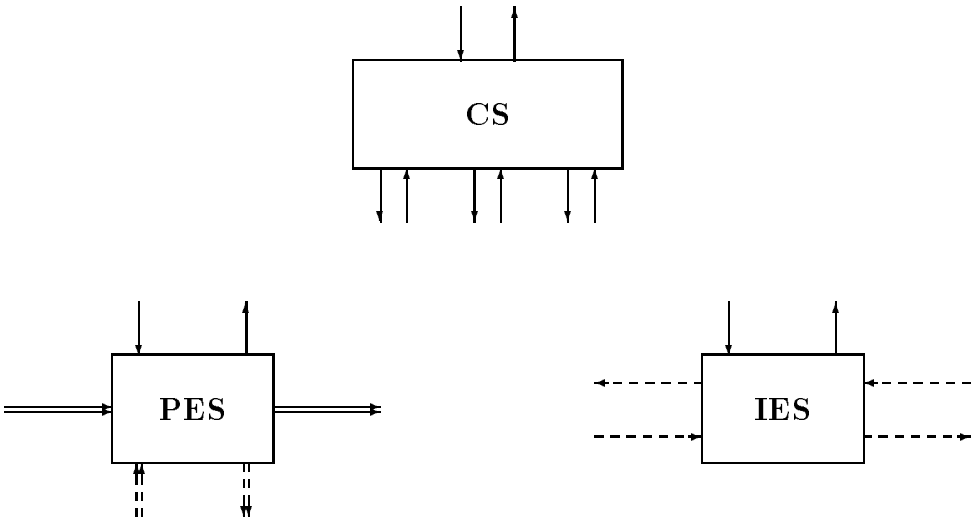

Figure 5: *The elementary systems*

At the lowest level there are three kinds of systems:

- *physical elementary systems* (PES)
- *information elementary systems* (IES)
- *control systems* (CS)

Physical elementary systems (PES) are systems dealing with resources and are controlled by master/slave interactions. Examples of PES are machines, automated guided vehicles and people doing manual work.
Information elementary systems (IES) are systems dealing only with information. An IES is also controlled by master/slave interactions.
Examples of IES are demand forecast and order entry systems. An IES is controlled by some higher authority and communicates with other (information) systems via requests and responses (client/server interactions).
Elementary systems (PES and IES) are controlled by a control system (CS). A control system controls subordinate systems via master/slave interactions and is controlled by master/slave interactions. Examples of CS are: real-time controllers, MRP-modules and managers. In general an incoming command is translated in a number of commands for the subordinate systems.
Figure 5 shows a graphical description of these three kinds of elementary systems, i.e. PES, IES and CS.

Now we can give a recursive definition of a *logistic system* (LS): a logistic system is an elementary system (PES or IES) or a set of logistic systems controlled by a control system (CS). Figure 6 shows an example of a logistic system.
Note that there is a *hierarchy* of systems. This approach allows us to decompose a logistic system into a control system and a number of logistic sub-systems. This decomposition process is repeated until the logistic sub-systems are considered elementary.

Our definition of a logistic system (LS) is summarized in the box. Physical elementary

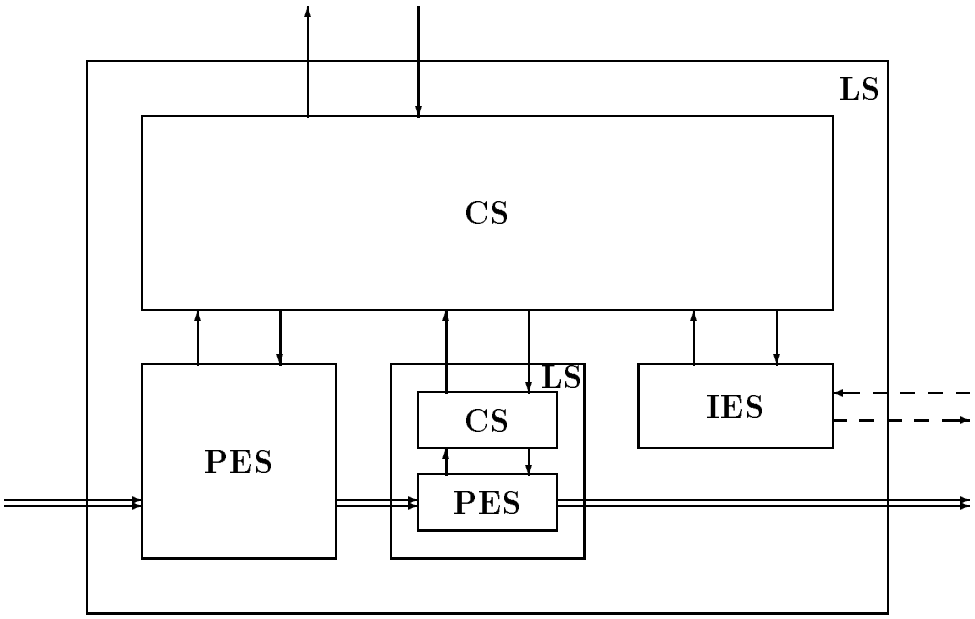

Figure 6: *A logistic system*

system and information elementary system are logistic systems. A group of logistic systems is a logistic system. An aggregate of one or more logistic systems controlled by some control system is also a logistic system.

```
PES =  physical elementary system
IES =  information elementary system
CS =  control system
LS = PES | IES | LS-list | CS,LS-list
```

Using this framework we have modelled typical control structures encountered in the field of logistics. With our approach it is possible to express the recent developments in logistics (JIT, MRP, BSC, Kanban), see [2].

3 EXSPECT

At Eindhoven University a specification language, called ExSpect, has been developed (see van Hee et al. [8], [9] and van der Aalst and Waltmans [5], [4]). This specification language has been developed to describe *discrete dynamic systems* ([8]). We use this language to specify logistic systems. ExSpect is based on a *timed coloured Petri net* model, i.e. a Petri net model extended with time and "coloured" tokens.

The classic (or basic) Petri net is a directed labelled bipartite graph with two node types called *places* and *transitions* (see Murata [14]). The nodes are connected via labelled *arcs*. Connections between two nodes of the same type are not allowed. Places are represented by circles, transitions by bars. A place p is called an *input place* of a transition t if there exists a directed arc from p to t. A place p is called an *output place* of a transition t if there exists a directed arc from t to p. Places may contain zero or more *tokens*, drawn as black dots. The number of tokens may change during the execution of the net. A

transition is called *enabled* if there are enough tokens on each of its input places. In other words, a transition is enabled if all input places contain (at least) the specified number of tokens. An enabled transition can *fire*. Firing a transition means consuming tokens from the input places and producing tokens for the output places. The state of a Petri net is the distribution of tokens over the places. Many authors use the term *marking* to denote the state of a basic Petri net. A *firing sequence* is a sequence of states, such that any state s_i is followed by a state s_{i+1}, resulting from the firing of some enabled transition in state s_i.

Many authors have extended the basic Petri net model with *coloured* or *typed tokens* (e.g. Jensen [11], van Hee et al. [8]). In these models tokens have a value, often referred to as colour. There are several reasons for such an extension. One reason is the fact that (uncoloured) Petri nets tend to become too large to handle. Another reason is the fact that tokens often represent objects or resources in the modelled system. As such, these objects may have attributes, which are not easily represented by a simple Petri net token. These "coloured" Petri nets allow the modeller to make much more succinct and manageable descriptions, therefore they are called "high-level" nets.
To specify complex logistic systems it is useful to extend the Petri net model with time. We use a rather new timing concept, where "time" is in tokens. See van Hee et al. [8] and van der Aalst [1], [2] for more information. An important advantage of using a formalism based on Petri nets, is the fact that we can use all kinds of analysis methods developed for Petri nets.
Existing techniques which can be used to analyse timed coloured Petri nets, may be subdivided into four classes:

Simulation is a technique to analyse a system by conducting controlled experiments (see Shannon [15]). These experiments are used to verify the correctness of the model and to predict the behaviour of the system under consideration. Because simulation does not require difficult mathematical techniques, it is easy to understand for people with a non-technical background. Simulation is also a very powerful analysis technique, since it does not set additional restraints.

Recent developments in computer technology stimulate the use of simulation for the analysis of timed coloured Petri nets. The increased processing power allows for the simulation of large nets. Modern graphical screens are fast and have a high resolution. Therefore, it is possible to visualize a simulation graphically (i.e. animation).

Markovian analysis For timed coloured Petri nets with certain types of stochastic delays it is possible to translate the net into a *continuous time Markov chain*. This Markov chain can be used to calculate performance measures like the average number of tokens in a place and the average firing rate of a transition (see Ajmone Marsan [13]).

Structural analysis There are several kinds of structural analysis, by which it is possible to prove that a given Petri net has a set of desired properties (e.g. absence of traps and deadlocks, boundedness, liveness, invariance, etc.).

An interesting way to analyse a coloured Petri net is to calculate (or verify) *place and transition invariants* (P and T-invariants). Place and transition invariants can be used to prove properties of the modelled system (see Jensen [11]). Intuitively, a place

invariant assigns a weight to each token such that the weighted sum of all tokens in the net remains constant during the execution of any firing sequence. By calculating these place invariants we find a set of equations which characterizes all reachable states. Transition invariants are the duals of place invariants and the main objective of calculating transition invariants is to find firing sequences with no 'effects'.

Reachability analysis is a technique which constructs a reachability graph, sometimes referred to as reachability tree or occurrence graph (Jensen [11], Murata [14]). Such a reachability graph contains a node for each possible state and an arc for each possible state change. Reachability analysis is a very powerful method in the sense that it can be used to prove all kinds of properties. Another advantage is the fact that it does not set additional restraints.

Obviously, the reachability graph needed to prove these properties may, even for small nets, become very large (and often infinite). If we want to inspect the reachability graph by means of a computer, we have to solve this problem. This is the reason we have developed several reduction techniques, see [1] and [2].

The language ExSpect consists of two parts: a functional part and a dynamic part. The functional part is used to define types and functions needed to describe operations on the value of a token. 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.
The dynamic part of ExSpect is used to specify a network of transitions and places and therefore the interaction structure of a system. The behaviour of a transition, i.e. the number of tokens produced and their values, is described by functions. The language also has a hierarchical construct called *system*. A system is a subnet, i.e. an aggregate of places and transitions. A system can also contain other (sub) systems. The system concept supports both top-down and bottom-up design. A system can have a number of parameters. As a result, a system can be customized or fine-tuned for a specific situation. This way it is possible to specify generic system specifications, that are easy to reuse.

The language ExSpect is supported by a software package, also called ExSpect. This software package consists of a number of tools. These tools are integrated in a *shell*, from which the different tools can be started. The *design interface* is a graphical mouse driven editor, which is used to construct or to modify an ExSpect specification. Such a specification is stored in a source file (module). This source file is checked by the *type checker* for type correctness. If the specification is correct the type checker generates an object file, otherwise the errors are reported to the design interface. The *interpreter* uses the object file to simulate the specification. This interpreter is connected to one or more *run-time interfaces*. These interfaces allow one or more users to interact with the running simulation. It is also possible to interact with some external applications, for example presentation software. Recently we added the *ITPN Analysis Tool* (IAT) to ExSpect. This tool translates a specification into an interval timed Petri net which is analysed using the methods described in [1]. The tool also allows for more traditional kinds of analysis such as the generation of place and transition invariants.

4 A LOGISTIC LIBRARY

We have modelled (specified) many logistic systems. These practical experiences show that these logistic systems have subsystems which have a lot in common. For example, a distribution centre and a production unit have transportation subsystems for internal transport, from a modelling point of view these subsystems are (often) similar. Especially when we structure the logistic system as described in section 2, we encounter typical subsystems. To support the modelling process it is useful to reuse these subsystems, often called *components* or *building blocks*. Reusing these components reduces the modelling effort.

In this section we describe a small logistic library containing a number of powerful predefined components. These components are inspired by work done in the TASTE-project. To structure the library and to support the modelling process we use the approach presented in section 2. We start by identifying a number of useful data-types using the classification given in section 2. Then we describe a number of logistic building blocks. It is our belief that a library like this one is useful when modelling "real world" logistic systems. A building block (component) is considered to be useful if it is:

- easy to use
- powerful
- flexible
- robust

A component is easy to use if it is easy to understand its semantics and there is a straightforward relation with the world we want to model. The modelling power of a library depends on the expressive power of the building blocks (is it possible to model something?) and the average size of a model in terms of the building blocks. Note that it is possible to have building blocks allowing for the modelling of a large class of systems but in a roundabout way. Compare this to programming in assembler, i.e. it is possible to program anything but it takes a lot of effort. The flexibility of a component also depends on two aspects: (1) is it easy to adapt the component and (2) are the important characteristics of a component parameterized. Parameterized building blocks are useful, because they are tailored for a specific situation. Finally, a building block has to be robust in the sense that it can handle various inputs, i.e. the number of assumptions about the environment of the component is as small as possible.

The logistic library we have developed, is an attempt to maximize the four objectives: easy to use, powerful, flexible and robust. Note that some of these objectives may be contradictory. Our goal is not to present an exhaustive list of logistic components covering all situations encountered in logistics, but to show that it is possible to create a comprehensive set of generic logistic building blocks. Our aim is to capture logistic knowledge in this library and to validate the "80/20-situation", where 80 percent of the components needed are already available in standard libraries 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.

The library we propose is hierarchical, i.e. some of the building blocks are composed

of other building blocks. ExSpect supports the user of this library to make his own building blocks from already existing ones. This way the user is enabled to make complex hierarchical models with a lot of levels. Therefore we provide some guidelines: (1) the number of levels in the hierarchy (visible to the user) should be smaller than 5, (2) the number of different building blocks at the same level (in a subsystem) should be smaller than 10. Note that these figures a only guidelines, they depend on the system to be modelled.

4.1 The type definitions

In this section 2 we presented a classification of the flows inside a logistic system. We will use this to classify the type definitions used by the logistic building blocks. A list of basic type definitions is given in table 1. The type `material` is a mapping from products

```
type id from num;
type location from str;
type prod from str;
type operation from str;
type capacity from real;
type timewindow from real >< real;
type commit from bool;
type conditions from real;
type age from real;
type material from prod -> real;
type task from operation >< capacity;
type route from (num -> (location >< $task)) >< num;
```

Table 1: *Some basic type definitions*

(`prod`) to reals representing the quantity of each product. The type `timewindow` is used to denote an interval of time. Another interesting type is the type `route`. A route is a list of pairs and a pointer pointing to a pair in the list. Each pair is formed of a location and a set of tasks. The pointer is used to identify the current location and the tasks to be executed at this location. Note that the list is implemented as a mapping from `num` to `location >< $task`. Table 2 shows a value of type `route`. All other types definition in table 1 are self-explanatory.

route				
num	location	$ task		num
		operation	capacity	
1	'EindhovenDC'			
2	'ParisPU8'	'drillingFA8'	2.55	
		'grindingDR7'	1.08	
		'grindingRT6'	1.29	2
3	'LyonPU9'	'paintHG9'	4.93	
		'polishIR7'	0.08	
4	'MadridDC'			

Table 2: *A value of type* `route`

```
– 1.1
 type goods from id >< route >< material;
– 1.2
 type means from id >< (operation -> capacity) >< age;
– 2.1.1
 type realtimeprodcommand from $goods >< means >< task >< $goods;
 type realtimeprodsignal from $goods >< $means;
– 2.1.2
 type aggprodcommand from (prod >< timewindow) -> real;
 type aggprodsignal from (prod >< timewindow) -> real;
– 2.1.3
 type delivercommand from goods;
 type receivesignal from goods;
 type stocklevel from material;
 type acceptedorder from goods;
 type replenishcommand from (prod >< timewindow) -> real;
 type replenishsignal from material;
 type ordervolume from ((prod >< timewindow) -> real) >< (material);
 type orderlimit from (prod >< timewindow) -> (real >< conditions);
– 2.1.4
 type replenishmentstrategy from prod -> (str >< real >< real >< real);
 type inventorylevels from prod -> (real >< real >< real);
– 2.1.5
 type routecommand from (num -> (location >< $goods >< $goods)) >< means;
 type routesignal from means >< location;
 type availabletranscap from timewindow -> (operation -> (capacity >< conditions));
 type acceptedtransorder from goods;
– 2.1.6
 type transportstrategy from str >< real >< real >< real;
 type transportperformance from real >< real >< real;
– 2.2
 type request from id >< route >< material >< timewindow >< conditions >< commit;
 type response from id >< route >< material >< timewindow >< conditions >< commit;
– 2.3
 type report from str;
– 2.4
 type admin from str;
```

Table 3: *Some logistic type definitions*

Table 3 shows some type definitions, each corresponding to a specific kind of flow in a logistic system. The flow of goods is represented by the type `goods`. Goods flowing through the network have an identification, some routing information and some materials associated to it. Examples of objects of type `goods` are, a pallet, a parcel or a single product. Table 4 shows a value of type `goods` representing a set of parts, needed to produce a car, with identification 897654. Note that currently the parts are located in Paris, where they have to be assembled.

Objects of type `means` have an identification, an age and a capacity for each kind of

goods							
id	route					material	
	num	location	$ task		num	prod	real
			operation	capacity			
897654	1	'EindhovenDC'			2	'chassisX19'	1
	2	'ParisPU8'	'drillingFA8'	2.55		'wheelT45'	4
			'grindingDR7'	1.08		'engineFM11'	1
			'assembleRT6'	1.29			
	3	'LyonPU9'	'paintHG9'	4.93			
			'polishIR7'	0.08			
	4	'MadridDC'					

Table 4: *A value of type* goods

operation the object can perform. This type is used to specify capacity resources, such as machines, trucks, etc.

Client/server interactions are represented by objects of the type `request` and `response`. A request has an identification, a route, a contents (material), a time window, a condition and a commit field. The usual interpretation of a request is: "can you deliver me some materials within a time window, given some conditions". If the commit field is "true" the request is automatically satisfied if possible. The `conditions` field is used to specify the requested conditions, for example maximal price or minimal quality. In all cases a request signal is followed by a response signal having the same identification.
The other types (mainly master/slave interactions) will be discussed when we describe the corresponding building blocks. Note that we chose "the easy way out" to model reports and administrative information.

4.2 The stock point

In this section we describe a building block: the `sp` system, where `sp` stands for stock point. This building block is composed of a number of other building blocks. All of these building blocks can be used to model inventories. Examples of stock points (i.e. `sp` systems) are a regional warehouse, a distribution centre or a storage area containing supplies and raw materials. The main characteristic of our stock point is that it has a more or less autonomous behaviour.
ExSpect system definitions are split in a header and a contents part. The header part (sometimes called signature) contains the system name, its interaction structure and its parameters. The interaction structure is given by (possibly empty) lists of input places and output places. The contents part describes the internal structure of the system. For more information, see [5] or [8].
The header of the `sp` system is:

```
sys sp[in incommand:replenishmentstrategy, responsein:response,
          ingoods:goods, requestin:request,
       out outstatus:inventorylevels, requestout:request,
           outgoods:goods, responseout:response,
       val reporttime:real,
```

```
        location:location,
        suppliertable:(prod->((location->num)><conditions)),
        expectedorderleadtime:real,
        expectedhandlingtime:real,
     fun replenish[s:replenishmentstrategy,physicalstock:material,
                   demand:((prod><timewindow)->real),
                   ordered:((prod><timewindow)->real)
                  ]:replenishcommand,
        orderlimit[s:replenishmentstrategy,physicalstock:material,
                   demand:((prod><timewindow)->real),
                   ordered:((prod><timewindow)->real)
                   ]:orderlimit,
        handleintime[x:material]:real,
        handleouttime[x:material]:real
   ]
```

There are four input pins and four output pins. The pins `ingoods` and `outgoods` represent the flow of goods in and out of the stock point. If some external party needs some products it can send a request to the stock point via the place connected to `requestin`. The stock point responds via `responseout`. The main objective of a stock point is to keep inventories of certain products, if the inventory level of a product falls below a certain level or we want to anticipate on future developments a replenishment is needed. To order the products necessary for such a replenishment we have the pins `requestout` and `responsein`. The replenishment strategy can be altered by some "higher" authority via the `incommand` and `outstatus` pins. The meaning of the value and function parameters will be discussed when we describe the subsystems of `sp` shown in figure 7.

The system `stockcontrol` controls the other two logistic subsystems `replenish` and `distribute`:

```
sys stockcontrol[in incommand:replenishmentstrategy, rs:replenishsignal,
                    ov:ordervolume,
                 out outstatus:inventorylevels, rc:replenishcommand,
                     ol:orderlimit,
                 val reporttime:real,
                 fun replenish[s:replenishmentstrategy,physicalstock:material,
                               demand:((prod><timewindow)->real),
                               ordered:((prod><timewindow)->real)
                              ]:replenishcommand,
                     orderlimit[s:replenishmentstrategy,physicalstock:material,
                                demand:((prod><timewindow)->real),
                                ordered:((prod><timewindow)->real)
                               ]:orderlimit
                ];
```

This system has an interface with some higher authority which tells the system to change

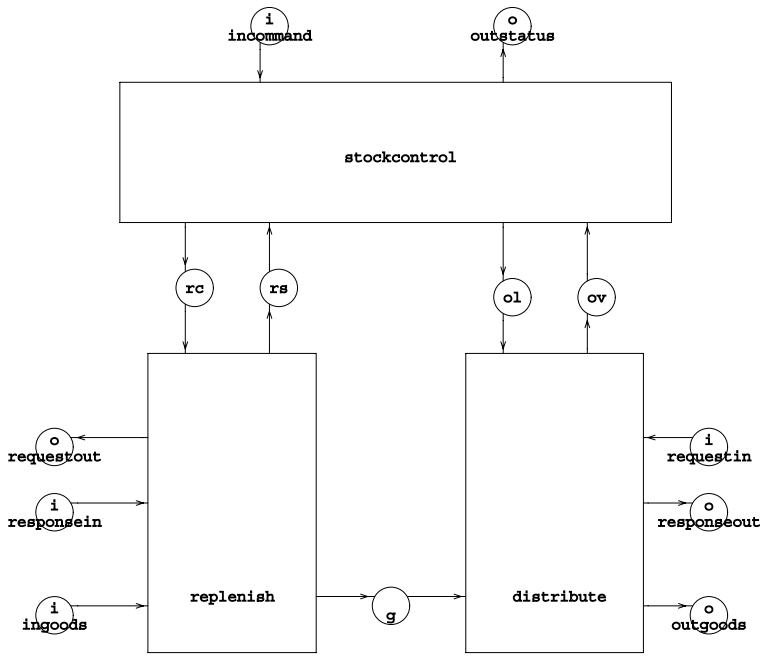

Figure 7: *The stock point*

its replenishment strategy. This strategy is defined for each product, see table 3. A strategy has a name and a number of parameters. Based on this strategy and the function parameter `replenish` the system issues replenishment commands via output pin `rc`. The parameters of the function `replenish` are the strategy (`s`), the current stock (`physicalstock`), the backorders and expected demand (`demand`) and the products already ordered (`ordered`). The input pin `rs` keeps the `stockcontrol` system informed about the (physical) replenishments. The output pin `ol` of type `orderlimit` is used to pass the upper bounds for the quantity of distributed goods in each period to the `distribute` system. The input pin `ov` keeps the `stockcontrol` system informed about the physical stock (`material`) and the actual demand for products (`(prod><timewindow)->real`). The upper bounds are determined using the function parameter `orderlimit`. The parameters of this function are identical to the parameters of the `replenish` function. From time to time the system reports the physical stock level, the demand level and the amount of ordered products using the output pin `outstatus`. The time between two successive reports is set using the `reporttime` parameter.

The system `replenish` takes care of the ordering of goods to replenish the stock:

```
sys replenish[in incommand:replenishcommand, response:response, ingoods:goods,
              out outsignal:replenishsignal, request:request, outgoods:goods,
              val reporttime:real,
                  location:location,
                  suppliertable:(prod->((location->num)><conditions)),
                  expectedorderleadtime:real
             ]
```

The meaning of the input and output pins follows directly from figure 7. The `replenish` system accepts all goods addressed to the `location` parameter and sends them to the place connected to `outgoods`. Periodically the total quantity of accepted goods is reported. The time between two successive reports is specified by the value parameter `reporttime`. The value parameters `suppliertable` and `expectedorderleadtime` are used to order the products.

The system `distribute` accepts orders, stores products and distributes them:

```
sys distribute[in incommand:orderlimit, request:request, ingoods:goods,
               out outstatus:ordervolume, response:response, outgoods:goods,
               val location:location,
                   reporttime:real,
                   expectedhandlingtime:real,
               fun handleintime[x:material]:real,
                   handleouttime[x:material]:real
              ]
```

The meaning of the pins is straightforward given figure 7. The `distribute` system reports the current inventory level and the accepted orders from time to time (as specified by `reporttime`) via the output pin `outstatus`. The value parameter `expectedhandlingtime` is used to determine whether it is possible to deliver within the requested time window. An upper bound for the number of products that can be supplied in each period is given via the input pin `incommand`. The two function parameters represent the time it takes to store and the time to pick some material.

Now it is time to take a closer look at the logistic subsystems `replenish` and `distribute`. Figure 8 shows the internal structure of the `replenish` system. It contains three subsystems: `replenishcontrol`, `procurement` and `acceptgoods`. The `replenishcontrol` system passes the replenishment commands to the `procurement` system and reports the total amount of received goods for each period. The `procurement` system orders goods requested by the `replenishcontrol` system. The `acceptgoods` system informs the `replenishcontrol` system about the actual replenishments. Note that `procurement` is an IES, `acceptgoods` is a PES and `replenishcontrol` is a CS.

The internal structure of the `distribute` system is shown in figure 9. The subsystem `acceptorders` handles the incoming requests for goods and reports all accepted orders to the `distributioncontrol` system. The control system `distributioncontrol` passes the maximum order quantity for each period to the `acceptorders` system. It also controls the `stockholding` system by issuing commands via the output pin `dc` of type `delivercommand`.

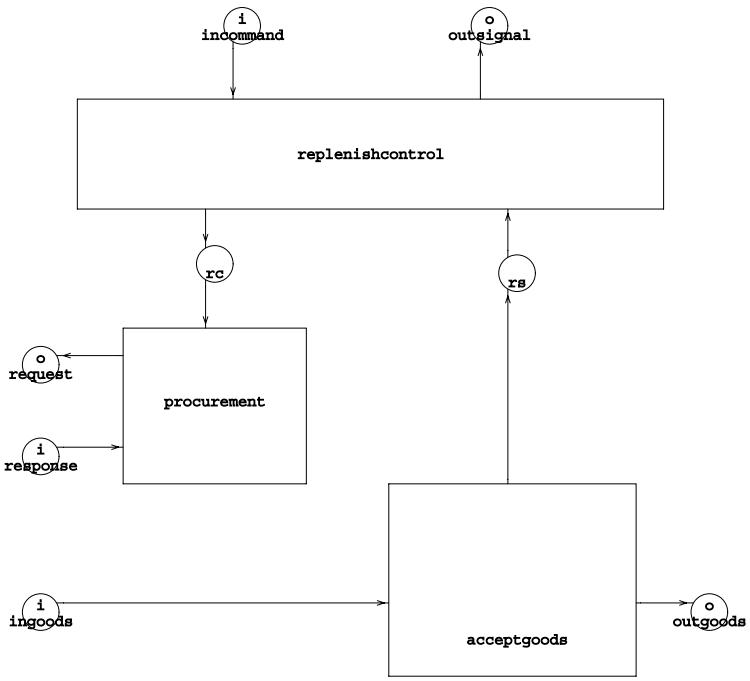

Figure 8: *The replenish subsystem*

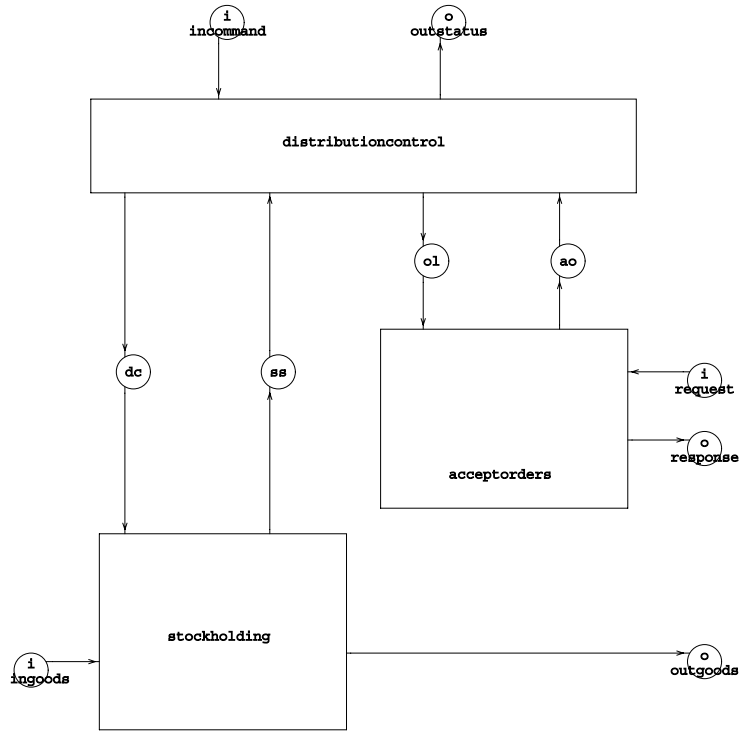

Figure 9: *The distribute subsystem*

This concludes our description of the building blocks related to stock holding. We realize that this description is far from complete, but is gives the reader a impression of the modelling capabilities of such a library. We also specified building blocks for other logistical processes, such as, transportation, demand, supply and production. The total number of building blocks available is about 20, see [2] for a more extensive description of this library.

5 CONCLUDING REMARKS

We described a framework for the modelling and specification of logistic systems. This framework consists of a language, a software package and the method presented in this paper. The method has been used to develop a number of standard logistical components. These components reduce the time needed to model a complex logistic system.
The modelling effort depends on the availability of these building blocks. We aim at a "80/20"-situation, where 80 percent of the components needed are already available in standard libraries 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.
Practical experiences give us the confidence that this approach is powerful and easy to apply.
In further research the following subjects are of our interest. First, we want to extend our modelling approach to a well-defined method. We also want to improve the logistic library and create new ones (e.g. a library for information systems). Finally, we want to improve the existing analysis tools.

Acknowledgement

This research has been supported by IPL-TUE/TNO.

References

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

[2] W.M.P. van der Aalst. *Timed coloured Petri nets and their application to logistics*. PhD thesis, Eindhoven University of Technology, Eindhoven, 1992.

[3] W.M.P. van der Aalst, M. Voorhoeve, and A.W. Waltmans. The TASTE project. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets*, pages 371–372, Bonn, June 1989.

[4] 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, Nürnberg, June 1990. Simulation Councils Inc.

[5] 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, Amsterdam, 1991.

[6] J.W.M. Bertrand, J.C. Wortmann, and J. Wijngaard. *Production control: a structural and design oriented approach*, volume 11 of *Manufacturing Research and Technology*. Elsevier Science Publishers, Amsterdam, 1990.

[7] F.P.M. Biemans. *Manufacturing Planning and Control: a reference model*, volume 10 of *Manufacturing Research and Technology*. Elsevier Science Publishers, Amsterdam, 1990.

[8] 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.

[9] K.M. van Hee, L.J. Somers, and M. Voorhoeve. A Formal Framework for Dynamic Modelling of Information Systems. In H.G. Sol and K.M. van Hee, editors, *Dynamic Modelling of Information Systems*, pages 227–236. Elsevier Science Publishers, Amsterdam, 1991.

[10] N.E. Hutchinson. *An integrated approach to logistics management*. Prentice-Hall, Englewood Cliffs, 1987.

[11] 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, Berlin, 1990.

[12] K. Jensen and G. Rozenberg, editors. *High-level Petri Nets: Theory and Application*. Springer-Verlag, Berlin, 1991.

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

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

[15] R.E. Shannon. *Systems simulation: the art and science*. Prentice-Hall, Englewood Cliffs, 1975.