Simulation Integrated Design

for

Logistics

Hans P.M. Veeke

March, 2003

Simulation Integrated Design

for

Logistics

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft
op gezag van de Rector Magnificus prof. dr. ir. J.T. Fokkema,
voorzitter van het College voor Promoties,

in het openbaar te verdedigen op dinsdag 24 juni 2003 om 10.30 uur
door Hermanus Petrus Maria VEEKE
wiskundig ingenieur
geboren te Breda

Dit proefschrift is goedgekeurd door de promotoren:
Prof. dr. ir. G. Lodewijks
Prof. ir. H. Bikker

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. ir. G. Lodewijks	Technische Universiteit Delft, promotor
Prof. ir. H. Bikker	Technische Universiteit Delft, promotor
Prof. dr. ir. J.A.E.E. van Nunen	Erasmus Universiteit Rotterdam
Prof. dr.-ing. H.J. Roos	Universität Stuttgart
Prof. dr. ir. R. van Landeghem	Universiteit Gent
Prof. ir. J. in 't Veld	Technische Universiteit Delft
Dr. ir. J.A. Ottjes	Technische Universiteit Delft

Published and distributed by: DUP Science

DUP Science is an imprint of
Delft University Press
P.O.Box 98
2600 MG Delft
The Netherlands
Telephone: +31 15 27 85 678
Telefax: + 31 15 27 85 706
E-mail: info@library.tudelft.nl

ISBN 90-407-2417-2

Keywords: systems approach, logistics, process interaction simulation

Printed in the Netherlands

Acknowledgements

Imagine you have an independent profession with a sixty-hour working week and a family with four adolescent kids, then the conditions for writing a thesis are certainly not optimal. The success heavily depends on the support of the environment. Therefore, I do not consider this thesis as a result of me alone, but as the result of the many people who have supported me.

First of all I thank Mies, Bart, Jeroen, Max and Maartje for accepting the fact that the back of my head became more familiar to them than my face, which was staring to the screen mostly. I will certainly turn around more often from now on.

At the professional side many people have helped me in writing this thesis.

Firstly I wish to thank Gabriel Lodewijks, because he made the conditions to let me work on this thesis without any real worries. I have experienced the discussions with him and my other supervisor, Henk Bikker, as very productive and creative. I hope to continue the cooperation with both of them in future.

Secondly my thanks go to my colleagues of the departments Production Technology and Organization, and Marine and Transport Technology. Some helped me by reviewing intermediate results, others by offering literature and challenging brainstorms.

Finally, there are three people that I wish to thank explicitly. I would like to express my gratitude to Jan in 't Veld. I am not exaggerating when I say that he has learned me to "think" and what's more, to be "practical". I will never forget the moment when I presented my first results of academic research under his supervision. Within two minutes I was outside again, because my conditions were unacceptable in practice though scientifically correct. From that moment on, I realized that academic research and practical applicability are two different things and I took a lot of advantage of this experience.

Secondly, I am very grateful to Joan Rijsenbrij. I met him during the challenging Delta Sea-Land project in the late eighties. It was a unique experience and I learned a lot of him in stimulating people to achieve a goal that seems impossible to be reached at some times. It was the first time of my life I was "locked up" in a room with a few others and we were only allowed to leave it when we found a working solution. We did find it!

And finally I thank Jaap Ottjes, who I consider more than a colleague. I extremely enjoy our cooperation in the field of simulation. To be able to correctly implement the things we invent, you always need someone who is able to define the correct things.

Summary

The design of an innovative logistic system is a complex problem in which many different disciplines are involved. Each discipline developed its own way of conceptual modeling for a logistic system based on a mono disciplinary perception. It usually leads to a communication problem between the different disciplines and consequently to expectations of the performance that don't correspond with reality.

In this thesis a basic systems approach is used to define a conceptual model of a logistic system that can be used by all disciplines involved as a common reference of the design. A combination of a soft and a hard systems approach leads to a conceptual model in which the problem is formulated in terms of required performances and process structures. The logistic system is modeled as a structure of functions around three flows: orders, products and resources. The model evolves during the design project and is an enduring supporting tool for decision making with a clear relation to the systems objectives. This PROcess-PERformance model (PROPER) model is formulated in interdisciplinary terms and thereby enables the communication between different disciplines.

The PROPER model only reflects the structure of a system; it does not model the time dependent behavior of the system. This behavior is essential for correct decision making and usually this behavior is "simulated" on a computer. In practice simulation is only used during the final stages of a design project and then a correction of objectives and/or decisions is impossible or very expensive. In this thesis the use of simulation is recommended for decision making from the very start. To achieve this the description of time dependent behavior is also defined at an interdisciplinary level. Natural language is used to describe the processes as defined in the PROPER model at each aggregation stratum. These descriptions enrich the problem formulation phase by expressing the parallel and stochastic aspects of the system.

Like the other disciplines, simulation evolved as a specialist discipline. In order to preserve a direct connection with the process descriptions of the PROPER model, these natural language process descriptions are translated into an object oriented Process Description Language PDL. This language can be implemented in any object oriented software environment. It is here implemented in the Borland Delphi platform that is based on the programming language Pascal. The implementation is called TOMAS: "Tool for Object

oriented Modeling And Simulation". TOMAS is completely object oriented and fully complies with the "Process Interaction" implementation of the Discrete Event System Specification method (DEVS). In order to support the growing level of detail of the PROPER model during a design project, TOMAS also supports distributed simulation by offering an open event scheduling mechanism and communication between models at different aggregation strata.

Finally the use of PROPER, PDL and TOMAS is illustrated with an already finished complex project: the design of the first automated container terminal in Rotterdam. It is shown that the use of this approach would have led to a clear and complete objective definition and would have warned the project participants in an early stage for a mismatch between expected and real operational performance.
This approach will therefore not automatically lead to improved logistic designs, but it does contribute to a better correspondence between expectations and reality.

Samenvatting

Het ontwerpen van een innovatief logistiek systeem is een complex probleem, waarin vele verschillende disciplines zijn betrokken. Elke discipline ontwikkelde op eigen wijze conceptuele modellen van een logistiek systeem, die gebaseerd zijn op een monodisciplinaire perceptie. In het algemeen is dit de aanleiding voor een communicatieprobleem tussen de verschillende disciplines met als gevolg dat de verwachtingen ten aanzien van de prestaties niet overeenkomen met de werkelijkheid.

In dit proefschrift wordt een fundamentele systeembenadering gebruikt om een conceptueel model voor een logistiek systeem te ontwikkelen; dit model kan door alle disciplines worden gebruikt als een gemeenschappelijke referentie naar het ontwerp. Een combinatie van een zachte en harde systeembenadering leidt tot een conceptueel model, waarin het probleem wordt geformuleerd in termen van vereiste prestaties en proces structuren. Het logistiek systeem wordt gemodelleerd als een structuur van functies rond drie stromen: orders, materie en mensen en middelen. Het model groeit gedurende het ontwerpproject en biedt voortdurend ondersteuning bij de besluitvorming, steeds in relatie tot de systeemdoelstellingen. Dit "PROces-PERformance" model (PROPER model) maakt gebruik van een interdisciplinaire terminologie en maakt daardoor de communicatie tussen de verschillende disciplines mogelijk.

Het PROPER model geeft alleen de structuur van een systeem weer; het tijdafhankelijke gedrag van het systeem wordt erdoor niet gemodelleerd. Dit gedrag is essentieel voor een correcte besluitvorming en in het algemeen wordt dit "gesimuleerd" met behulp van een computer. In de praktijk wordt simulatie vaak pas gebruikt aan het eind van een ontwerpproject; dan is echter een correctie van doelstellingen en/of besluiten onmogelijk of erg duur. In dit proefschrift wordt daarom aanbevolen om simulatie te gebruiken vanaf het begin van een project. Om dit te bereiken wordt het tijdafhankelijke gedrag ook op een interdisciplinaire wijze beschreven. Natuurlijke taal wordt gebruikt om processen te beschrijven zoals ze voorkomen in het PROPER model op elk aggregatiestratum. Deze beschrijvingen verrijken de probleemstellingfase door parallelle en stochastische aspecten van het systeem expliciet uit te drukken.

Evenals de andere disciplines ontwikkelde simulatie zich als een specialisme. Om een directe verbinding met de bovengenoemde procesbeschrijvingen, en daarmee het PROPER model,

te behouden, worden deze beschrijvingen in natuurlijke taal vertaald naar een object georiënteerde taal: de Process Description Language (PDL). Deze taal kan in elke object georiënteerde software omgeving worden geïmplementeerd. De taal is in dit proefschrift geïmplementeerd in het Borland Delphi platform, dat gebaseerd is op de programmeertaal Pascal. De implementatie TOMAS ("Tool for Object oriented Modeling And Simulation") is volledig object georiënteerd and voldoet volledig aan de "Process Interaction" implementatie van het Discrete Event System Specification concept (DEVS). Om de tijdens het ontwerpproject toenemende graad van detaillering van het PROPER model te ondersteunen, biedt TOMAS faciliteiten voor gedistribueerde simulatie door middel van een "open event scheduling" mechanisme en communicatie tussen modellen op verschillende aggregatiestrata.

Tenslotte wordt het gebruik van PROPER, PDL en TOMAS geïllustreerd aan de hand van een reeds voltooid complex project: het ontwerp van de eerste geautomatiseerde containerterminal in Rotterdam. Er wordt aangetoond, dat het gebruik van de benadering uit dit proefschrift zou hebben geleid tot een duidelijker en volledige definitie van doelstellingen en dat de projectdeelnemers vroegtijdig zouden zijn gewaarschuwd voor een verschil tussen verwachte en werkelijke operationele prestaties.
Deze benadering leidt daarom niet automatisch tot betere logistieke ontwerpen, maar draagt wel bij tot een betere overeenstemming tussen verwachtingen en werkelijkheid.

Contents

List of Figures

Chapter 1

Introduction

"An expert is someone who knows more and more about less and less,
until eventually he knows everything about nothing."
- Anonymous -

One of the recurring problems in any large scale design project is the relation between the multidisciplinary design and the predominantly monodisciplinary participants. For example, the complaints about information systems that do not live up to expectations keep pace with the growth of automation. Machines (whether automated or not) often fail to deliver the performance that was specified during the design phase. The performance of a real-time system differs significantly from simulation results obtained during the design project.

The main subject of this thesis is the design of a logistic system and the use of simulation therein. Simulation theory has largely evolved as part of Operations Research during the last few decades. Many software tools resulted from this development and today it is hard to imagine a logistic system being developed without the use of simulation. Simulation is generally considered a powerful decision support tool. Nevertheless, it is a guess work as to whether the operational logistic system performs exactly as was predicted by the simulated system, or not. Some of the problems may be caused by purely technical issues such as an inadequate simulation platform, misjudged system loads, or a drastically changed environment in which the system works. But in many cases, the design of the simulation model is simply not appropriate nor complete for the environment in which it has to operate. Such a mismatch between the intentions of the decision makers and the perceptions of the simulation experts can only be explained by assuming an initial communication problem between these two parties.

1.1. Aim of this thesis

Currently all disciplines use some kind of a 'system' concept to deal with complex problems. For example, organization science considers organizations as combined *social, technical and*

economical systems; Logistics emphasizes an integrated approach to dealing with an *operational system*; Information technology developed several approaches for the design of *information systems*.

They all construct conceptual models to formulate problems and find solutions. However, significant differences are evident between the conceptual models of each discipline. The conceptual information model of a system is quite different compared to a conceptual logistic model of the same system. Apparently, conceptual modeling is part of the discipline itself. Simulation theory is no exception in this respect. To assure the validity of a conceptual model, most disciplines develop verification and validation activities. Figure 1.1 shows these activities as defined in the field of simulation (based on [van Gheluwe, 2000]).

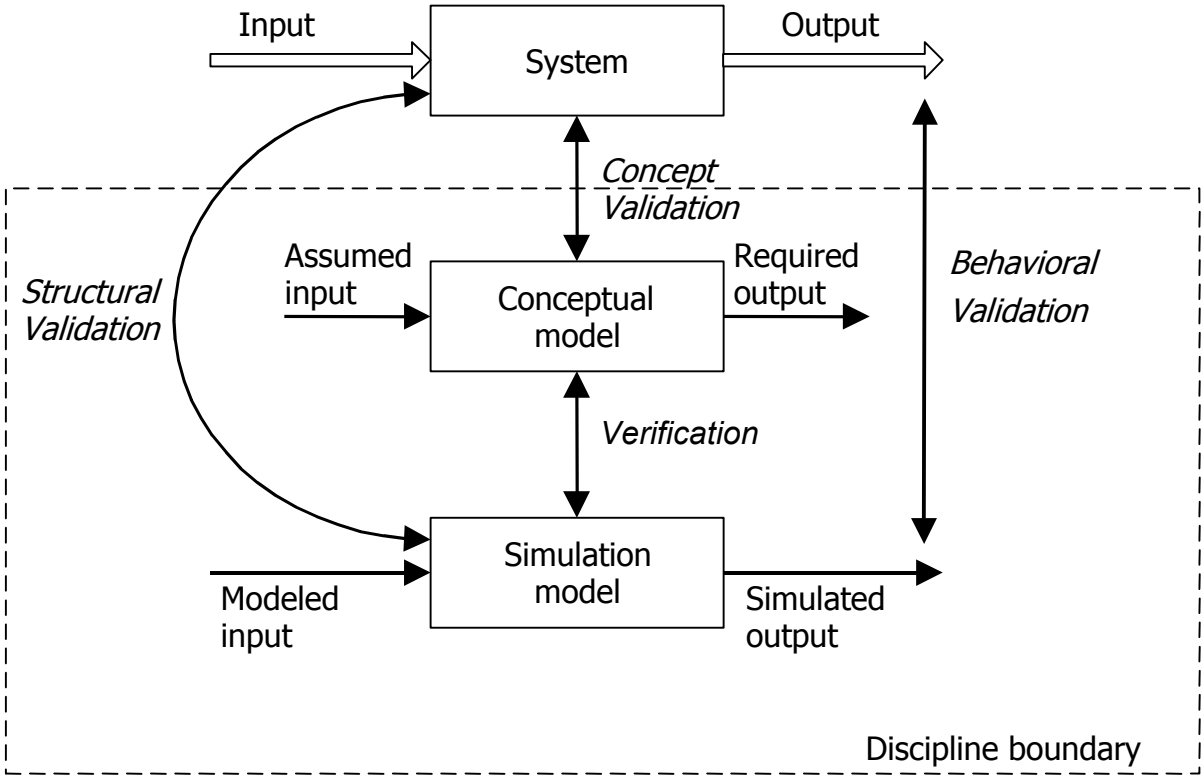


Figure 1.1: *Verification and validation activities*

There are three "systems": the real system with its (measured) inputs and outputs, the conceptual model of the system with assumed inputs and required outputs and finally the simulation model with modeled inputs and simulated outputs.

Validation checks the consistency of measurements in the real system with the simulated system. There are various kinds of validation. *Concept validation* between reality and the conceptual model is primarily the evaluation of realism of the model with respect to the goals of the study. *Structural validation* concerns the evaluation of the structure of the

simulation model with respect to the perception of the system structure. *Behavioral validation* evaluates the behavior of the simulation model. All kinds of validation depend on the presence of a real system and offer the possibilities to judge the quality of a simulation. Validation is answering the question: "is this the correct simulation model?".

Verification checks the consistency of a simulation model with respect to the conceptual model. It is answering the question: "is the simulation model working correctly?" Verification is considered an activity that exclusively belongs to the field of simulation.

Validation is in fact the interface between the simulation discipline and the other disciplines, in order to assure confidence in the simulation results. In figure 1.1. the boundaries of the simulation discipline are denoted by the dotted rectangle. The way in which the simulation experts make their conceptual and simulation models, is controlled by the validation activities.
In practice this leads to three types of questionable situations:

- Simulation support is considered a 'black box': just pop in the questions and correct answers will be returned. Validation has become the responsibility of the simulation experts.
- Usually, the global conceptual modeling of the system during the first stages of a design project does not include the use of simulation. Only during the last stages of a design project is simulation used, because the system is then concrete enough to enable validation. As a result the (possible) contribution of conceptual simulation modeling to generic problem solving, is lost. Contrary to conceptual modeling in other disciplines, conceptual simulation modeling includes the description of time dependent behavior, which provides increased insight and understanding of the design.
- In the case of innovation projects the possibility for validation does not exist, because there is, as yet, no real system. The real system itself is a subjective perception of each participant involved.

The situations described above, can be improved by considering conceptual modeling a generic interdisciplinary activity rather than a multidisciplinary activity. The term "interdisciplinary" denotes cooperation with a common goal (and by this a common perception of the system). This goal is and stays the starting point for all activities during the

design project and project management serves this goal. A multidisciplinary activity on the other hand is a form of cooperation combining different perceptions and is directed by project management towards the system's goal. An interdisciplinary approach generates discipline specific concepts starting from a single system concept. A multidisciplinary approach generates a system concept starting from discipline specific concepts.
During the last half of the 20th century the systems approach emerged as an interdisciplinary approach to study "systems". It opens the way to a generic conceptual mode of modeling logistic systems, thereby avoiding the jargon and specifics of separate disciplines. As previously mentioned, each discipline uses its own systems concept that may be based on systems approach but starts from the monodisciplinary viewpoint. In this thesis, the systems concept will be elaborated from the other way around. Starting with a general concept, a conceptual model for logistic systems will be derived down to the level where single disciplines have to become specific.
This approach enables each discipline to add generic knowledge and experience to the conceptual model. After agreement is reached on the conceptual model each discipline can proceed with the design of specific aspects of the system.

Therefore the first central research question of this thesis is:

"How can the concept of systems thinking be successfully used to construct a generic conceptual model for the design of logistic systems?"

Returning to the field of simulation, the absence of a real system prevents the use of validation activities. The only way to guarantee the correctness of simulation results now is fully dependent on verification. For this reason, it is necessary to involve other disciplines in the verification activities.
The simulation model must reflect both the structure of the conceptual model and the expected behavior of the elements in the model, which results in the behavior of the system as a whole. Therefore, verification is divided into structural verification and behavioral verification (see figure 1.2).
The structure of the conceptual model must be implemented in a simulation model in such a way that it can be recognized (and thus verified) by all parties involved. This is even more important for the behavior of the modeled system. Most conceptual models are characterized by a static structure. The time dependent behavior is not explicitly represented. 'Modeling behavior' is in fact the kernel activity of simulation. Adding behavioral descriptions to the

conceptual model enriches the model and the understanding of the system of all participants. For this purpose, however, all parties involved must be able to understand and discuss descriptions of behavior.
The challenge is to define a conceptual way of representing behavior, which can be unambiguously used for implementation in a simulation program.

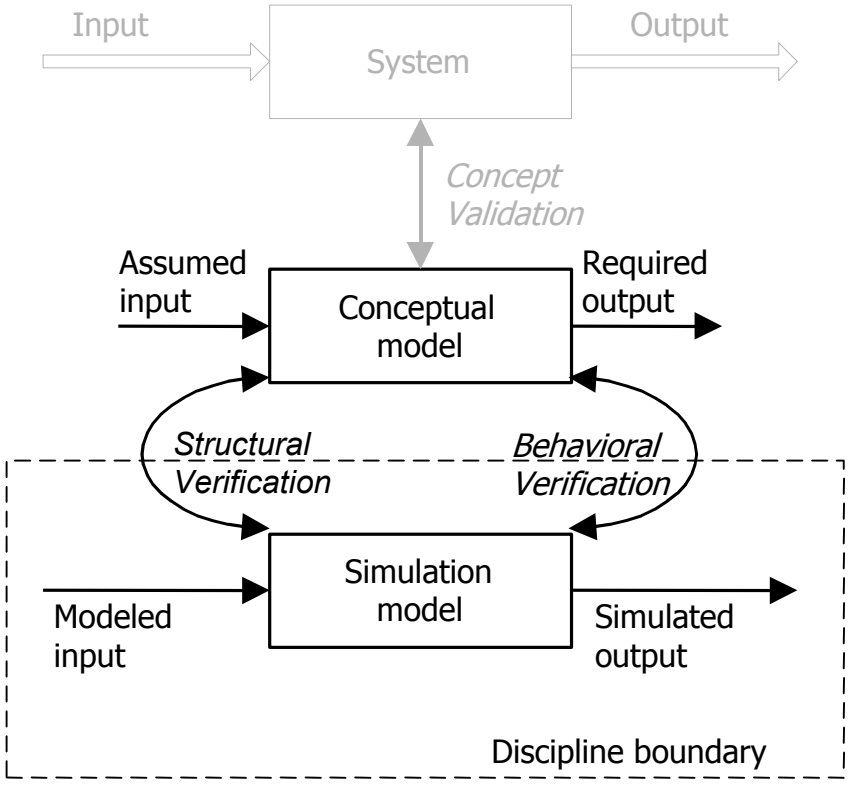


Figure 1.2: *Verification in the simulation of innovative systems*

This leads to the second central question of this thesis:

"How can the behavior of a logistic system be described in a conceptual interdisciplinary manner and how can it be implemented in a software environment?"

It will be shown that the research as described in this thesis, contributes to the field of the design of logistic systems as encountered in industry, transportation and common services. Firstly, it improves the quality of problem formulation by extending the range of interdisciplinary activities to include conceptual modeling. As such this the perceptions of the system under investigation converge to a common perception for all parties involved. What's more, all disciplines will be able to contribute to the modeling in an early phase of the design project.

Secondly, it improves the quality of the process of simulation modeling, especially in cases where validation activities are difficult or incomplete. Behavioral descriptions are added to the level of generic conceptual modeling. Implicit decision making on behavior will be excluded in this way.

The concrete advantages of this approach will be:

- an improved correspondence between design expectations and operational reality. The operational system itself will not be "better", but the results of the system will better fit to the expected results. Indirectly this may lead to better systems, if the expectations are not satisfactory and urge the creation of a better design.
- The construction of "shared memory" for succeeding (re)design projects by creating a "shared meaning" for system related terminology. The conceptual model represents decision making in defining goals, restrictions and alternatives. Future projects can use this information to decide upon the level at which a design revision effects the original design.

1.2. Overview of this thesis

The thesis consists of three parts.

Part I discusses conceptual modeling according to a general systems concept. Chapter 2 investigates terminology and modeling in systems approach and applies it to a logistic system. The result will be a conceptual model called the PROPER model (PROPER stands for Process-Performance) and in chapter 3 it will be positioned in the design process. The connection of the general conceptual model to specific organization, technological and information modeling, will be explained.

Part II introduces 'behavior' as the key concept to connect generic modeling to simulation modeling. In chapter 4 behavior will be described in a structured linguistic way.

Part III focuses on the simulation discipline. In chapter 5 the implementation of behavior in a software environment is prepared by using the object oriented approach of information system design to translate the behavior descriptions of chapter 4 into an object oriented but

still informal Process Description Language (PDL). In chapter 6 the PDL is implemented in the simulation platform TOMAS (Tool for Object oriented Modeling And Simulation). Chapter 7 shows the use of the PROPER model and TOMAS in the design project of the first automated container terminal DSL (Delta Sea-Land) of ECT (Europe Combined Terminals), a project that was performed in the period 1985 – 1995.
Finally chapter 8 summarizes conclusions and formulates ideas for future research.

Chapter 2

A systems approach to logistics

"The real voyage of discovery consists not in seeking new landscapes but in having new eyes"
- Marcel Proust -

2.1 Introduction

In this thesis, logistics is defined as "the coordination and control of the operational functions in their mutual connection" [de Vaan, 1991]. In this context a logistic system is the "whole" of people and means for executing the control and operation. A logistics system involves departments such as Finance, Marketing, Operations, Maintenance and Information Technology. With this multidisciplinary involvement, a monodisciplinary approach is insufficient to adequately describe logistic systems. The consequence is that the requirements specification for the design of a logistic system is liable to be incomplete. To prevent this shortcoming an approach is required that supports a complete description of the system, that can be applied by all disciplines involved and that can serve as a point of reference during the life time of a design process.

During the last decades, the systems approach evolved as a generic interdisciplinary approach to investigate and describe "systems", not only by studying the elements but by emphasizing the relations between the elements. This approach will be used here to determine the requirements for a conceptual description of a logistic system. Coordination and control, as mentioned by de Vaan, aim at some objective. In this thesis, the research is restricted to systems with an objective. The accepted conceptual models, as developed by the systems approach and applicable to these systems, will be described. They will be evaluated for common characteristics. On the basis of these characteristics, a conceptual model of a logistic system will be constructed. The model acts as a point of reference for all disciplines involved and creates a well defined framework for interdisciplinary communication.

2.2. The systems approach

The systems approach supports decision making by formulating problems 'in terms of systems'. A system is defined as a: "set of elements that can be distinguished from the entire reality, dependent on the objective of the researcher. These elements are mutually related and (eventually) related with other elements in the entire reality" [in 't Veld, 2002]. The term "element" is not to be interpreted as indivisible, but as "is part of a whole: the system".
The definition of a system leads to the next general characteristics of a system:

1. *It can be distinguished from a total reality or "environment".*
   The border line or zone is called "system boundary". The importance of choosing a proper system boundary is generally acknowledged. It is necessary to keep the environment involved during the entire research in order to prevent suboptimalization. The interaction with the environment must be described clearly [Holweg, 2001]. This is why the systems approach is also denoted as "thinking in context" [Whitmore, 1998].
2. *It contains elements.*
   Often the term 'component' is used, but 'element' is preferred here, because the term component already has a concrete meaning in several disciplines. An element may again be composed of elements, where the original system then acts as environment (elements may be considered subsystems). No rules or criteria on how to select elements in a system are found in literature. Paragraph 2.4. will elaborate this further and a number of rules will be formulated there.
3. *There are relations between the elements.*
   The relations reflect the structure of the system and by the relations the system's "emergent" properties arise. Emergent properties are only found at the stratum of the system as a whole, and not in any of its elements ("a car (system) drives, but none of its elements can drive"). Usually there are different relations between two elements for example a social relation, an economical relation, an organizational relation etc.

The definition of a system by in 't Veld refers to an interaction with the environment. In these cases a system is called an 'open' system and applies:

4. *There are relations between system and environment*.

Most systems contain elements that cooperate to achieve some objective; for these systems applies:

5. *The system has an objective*

Activities must occur to achieve the objective, so we must finally add:

6. *Systems contain processes: "something happens in time".*
   The state of a system continuously changes due to the mutual interaction between elements and the interaction between system and environment.

Logistic systems are systems that interact with their environment and try to achieve an objective. Therefore all characteristics described above apply to logistic systems.

The systems approach is also called the "systems movement". Checkland [1981] characterizes it by: "The systems movement is the set of attempts in all areas of study to explore the consequence of holistic rather than reductive thinking". Two things attract attention in this phrase:

- The systems movement appears in all research areas. The study of systems can be applied to all areas of science, rather in an interdisciplinary than in a multidisciplinary way as already explained in chapter 1.
- It's rather a way of holistic thinking than of reductive thinking. Holistic thinking emphasizes the study of the whole instead of the elements and is based on the idea that "the whole is more than the sum of the parts": this principle is also called "synergy".

It is the main subject of the systems approach to express the meaning of synergy and how this influences the structure and behavior of a system. The elements of a system can be studied neither isolated from each other nor isolated from the whole. For this reason the systems approach is complementary to the traditional way of scientific analysis, where complex problems are solved by reduction (simplification) and isolation (study an element separated from its environment). The systems approach preserves the context of the

elements. The traditional way of analyzing is but one of many ways to study systems: maybe necessary, but certainly not sufficient.

"Systems approach" is only one of the many terms used to denote the systems movement. Systems thinking, systems theory, systems analysis, systems engineering are widely used terms for partly overlapping, partly differing areas of knowledge [Kramer, 1991]. In this thesis the term "systems approach" is preferred, because it does not point to a discipline in particular and it denotes a way of research, which includes more than analysis alone. Research in a 'systemic' way aims to identify the general, structural and functional principles that can be applied to one system as well as to another (see example 2.1). "With these principles it becomes possible to organize knowledge in models, that are easily transferred…" [de Rosnay, 1988].

Example 2.1. The activities at a container terminal are performed by multiple types of equipment; there are, for example, quay cranes, stacking cranes, straddle carriers, multi trailer vehicles, automatic guided vehicles. Yet, regardless of the equipment used, all activities are traced back to three types of functionality: to transfer, to transport and to store. These types of functionality form the functional principles of all activities in any container handling system.

In literature, systems approaches are classified in different ways (see a/o Whitmore [1998], [Wigal, 2001], [Daellenbach, 2002] and [Williams, 2002]). The classifications range from dividing systems approaches according to the researcher's subjective or objective system perception to dividing systems approaches according the system's complexity level. All these classifications show that the theoretical development of the systems approach mainly took place in what is called the 'General Systems Theory' (GST) and in Cybernetics.

GST has been developed to describe the entirety of natural systems in a mathematical formalism [Bertalanffy,1973]. In this sense it is a reaction to the long period of specialization in science. Bertalanffy defines a system as an arrangement of elements, related in such a way that they form a unity or organic whole. The way to or reasons why to arrange are not explained further. This generic definition includes both open and closed systems.

Cybernetics introduced concepts like 'feedback' and 'homeostasis' (i.e. the system maintains its structure and functions under environmental change) [Wiener, 1948]. Ackoff [1969]

widened the area of the systems approach by stating that it can not only be applied to living organisms, but to anything that can be described as a system in a conceptual way, especially to organizations. In 1971 he introduced a 'system of systems concepts', consisting of 31 points. The main items of this system are [Ackoff, 1971]:

- a system functions in some larger system (so it is a subsystem)
- systems possess a structure, which shows how elements are connected
- systems possess processes.

In 't Veld [2002] defines a process as a series of transformations, by which one or more characteristics (position, form, size etc.) of the input element change.

Applications of the systems approach are divided into three categories: 'hard' systems approach, 'soft' systems approach and 'critical' systems approach ([Flood & Jackson, 1992]).

Hard systems approaches consider a system logically based and capable of unbiased description. They are characterized by the modeling of systems in order to optimize a performance or required objective. Their application is therefore restricted to purposive systems. All elements of the system are assumed to cooperate in concert to achieve the objective. The basic assumption, whether or not implicitly, is that the problem is stated right and unambiguous. Typically hard systems approaches are Operations research, systems analysis, software development, database design and systems engineering.

The soft systems approaches consider a system a subjective perception: dependent on the observer the same system is presented in different ways. The observer himself may also be part of the system and may have his own objectives besides the system's objective. For this reason these systems are called purposeful systems. Soft systems approaches therefore are mainly aimed at understanding and formulation of these so-called ill-defined problems and address the "what" question instead of the "how" question. Examples of the soft systems approaches are Soft Systems Methodology (SSM) en Theory of Constraints (TOC).
The hard systems approach is in fact part of the soft systems approach. Once the stakeholders reach agreement on the problem statement (a consensus on subjective perceptions), methods of the hard systems approach can be used to solve the problem. But validation of solutions again takes place in a subjective framework.

The critical systems approach emerged in the 1980's and "sits somewhat uncomfortably in the overlap between sociology, organization theory, systems thinking and by extension management science" [Daellenbach, 2002]. This approach looks at the methods developed by hard and soft systems approaches from the perspective of existing social structures and aims to define the conditions for their applicability. Contrary to the hard and soft systems approaches, this approach is still at the stage of scientific discussion. The contribution will result in a better definition of preconditions for problem statements and the period of validity of solutions. However, in this thesis the process from problem formulation to finding solutions is investigated, given the preconditions and an intended life time for the logistic system, so the critical systems approach will not be further considered.

Briefly recapitulating, a soft systems approach aims to state the right problem and a hard systems approach aims to solve the problem correctly.

Returning to the definition of a system as used in this thesis, the subjective perception is included: 'distinguishing the system is dependent on the researcher's objective'. There will be many system perceptions during the design process of a logistic system, because many disciplines are involved, each discipline having its own objectives. It is a matter of competing and sometimes even conflicting objectives and the first problem of the design process is to reach and maintain consensus on these objectives. Perception (and by this the objective) is determined by the environment in which people work.

Example 2.2. Efficiency generally forms part of the objective of a logistic system as a whole. But an operations manager interprets efficiency as a need for a high and constant degree of capacity utilization (an operational objective), which can be achieved by high stock levels; However a financial manager interprets efficiency as a need for low capital costs (a financial objective), which can be achieved by low stock levels in contrast with operations.

Both operations and the finance department in example 2.2. are elements of the logistic system. The only way to reach and maintain consensus on the objectives is to explicitly state the objectives of all elements and to aline these with the overall system objective.
The conclusion is that the design process of a logistic system requires a soft systems approach to deal with different perceptions. The design process starts with a so-called ill-defined problem. The first steps of the process must lead to an agreement on the objectives

and conditions. By then it is called a well-defined problem. Starting directly with a hard systems approach would bypass the proper objective definition and would lead to:

- accepting system boundaries as given.
  For example, looking at the effect of economic lot sizes, if one does not take into account the environment of the total supply chain, the savings may be smaller than the extra costs [Christopher, 1998].
- Considering elements as being naturally defined.
  If one regards an organization as a system, often the existing departments are regarded as the elements. But the departments are the result of design processes in the past. By doing so, the assumptions and starting points of these earlier design processes are implicitly imported into the new design process despite the new objectives and a changed environment.

A soft systems approach is required to deal with ill-defined problems (Ackoff [1971] calls them "messes") and to formulate well-defined problems ("difficulties"). A hard systems approach is required then to solve these well-defined problems.
As part of a hard systems approach many (more or less) short term theories or ideologies emerged, such as Manufacturing Requirements Planning (MRP), Business Process Redesign (BPR) and Supply Chain Management (SCM). According to Jonas [1997] they are the reason why fundamental generic theories are easily ignored. Even despite the development of good theoretical models McGinnis [2001] states: "Yet experts who design industrial logistics facilities rarely use these kinds of models; if at all, the models are used very late in the design process, to 'fine tune' the detailed design". A soft systems approach does not define system models, and the system models of a hard systems approach are either temporarily used or are too specific.

Now the problem is to find a system concept in a hard systems approach, which can be generally applied within the design process of a logistic system, taking the soft systems approach into account, and which can form a more or less lasting framework for specification and review of logistic systems.

Such a system concept will be called a *conceptual system model*. Only a small number of such conceptual models has been defined. Checkland [1981] positions the use of these

models in his Soft Systems Methodology (SSM), the most widely used and accepted soft systems approach.

First SSM will be described to position well known conceptual models. After that the conceptual models will be described shortly and finally a general concept for modeling open (logistic) systems will be defined.

2.3. The Soft Systems Methodology

The Soft Systems Methodology is shown schematically in figure 2.1. The methodology consists of 7 steps. The first step is the recognition of an unstructured problem situation. In the second step 'rich pictures' are made to describe and discuss the problem. Rich pictures are combinations of text and pictures expressing the situation in terms of the researcher and problem owner. Rich pictures use draughts of elements, structures, processes and environment.

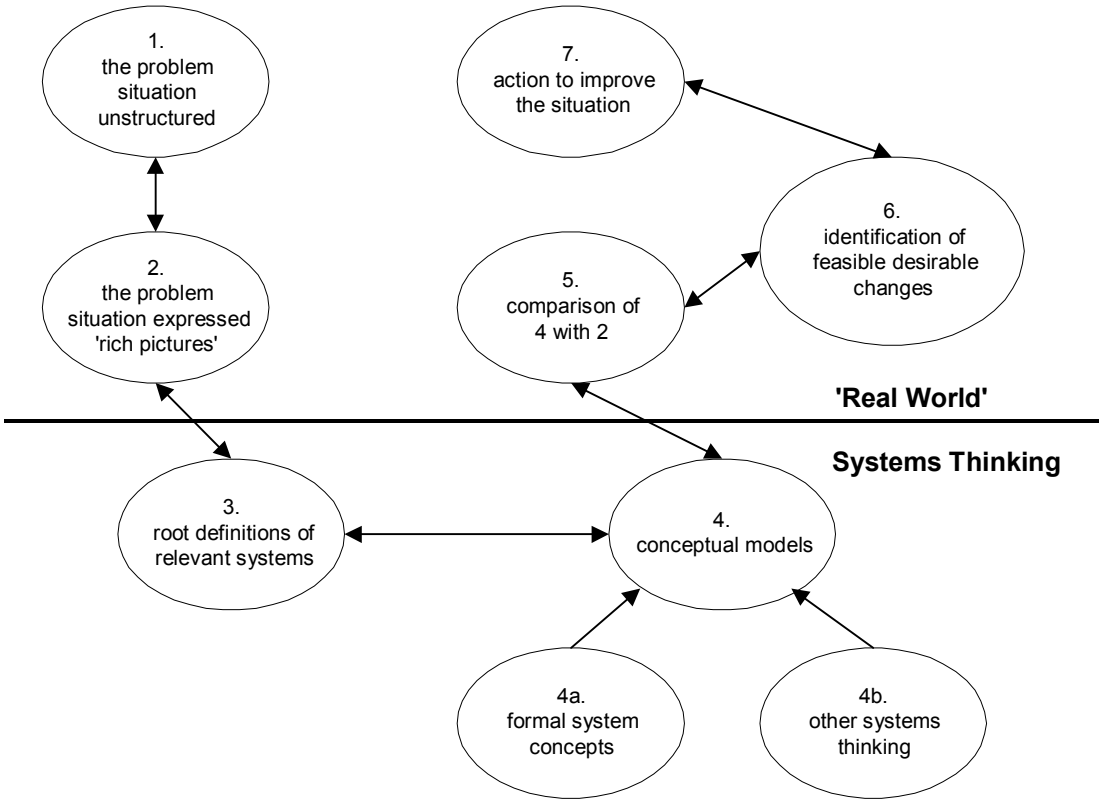


Figure 2.1: *Soft Systems Methodology*

In step 3 the rich pictures are analyzed and 'root definitions' are defined by abstraction. Relevant systems are distinguished in which activities are formulated. A number of activities is declared absolutely necessary for the system and these are the root definitions. A correct root definition satisfies the so-called CATWOE principle. It states explicitly the Customers, the Actors of the activity, the Transformation performed by the activity, the World view (Weltanschauung) of the activity, the Owners and the accepted preconditions from the Environment. The root definitions are used to construct conceptual models in step 4. In the next step these models are compared with reality as described by the rich pictures. The comparison leads to the identification of feasible and realizable changes. These changes determine the actions required to improve or solve the problem situation.
All steps in the methodology are executed in an iterative way.
Step 3 and 4 are classified as 'systems thinking' about reality. The other steps are rather informal and use natural terms and means to communicate about the problem in order to reach agreement about the problem statement and the possible solutions. The only rule Checkland formulates to construct a conceptual model is that the model must be based on root definitions. However his root definitions still are concrete activities. A general root definition embodying CATWOE might take the following form: "*A (..O..) owned system, which, under the following environmental constraints (..E..), transforms (..T..) this input (..input..) into this output (..output..) by means of the following major activities (...), the transformation being carried out by these actors (..A..) and directly affecting the following beneficiaries and/or victims (..C..). The world view, which makes this transformation meaningful contains at least the following elements (..W..)".* [Laws, 1996]
The objective of the activities however is missing in the CATWOE principle; the very same objective, which was found to be the expression of subjective 'perception' in paragraph 2.2.

As a result, in practice, many systems approaches use the term 'process' for an activity, construct models with it, but lack the connection with the objective of the process and consequently the objective of the system containing the process. For example, Business Process Reengineering (BPR) addresses objectives and missions indeed, but doesn't connect them to the modeling of business processes [Grover et al., 1995].

In this thesis, conceptual models are required to preserve the objectives as formulated in the soft systems approach. The root definitions (or modeling elements) in step 3 of figure 2.1. should therefore express these objectives. All modeling elements should contribute to the objective of the system they belong to. The conceptual models represent the problem

formulation part of the hard systems approach (step 4 in figure 2.1.) and the elements should be formulated in terms of their contribution to the system's objective. This type of elements are called "functions" from now on. Only four conceptual models are defined where elements represent functions: the Formal System Model (FSM), formulated for systems with human activities, the Viable System Model (VSM) developed by Stafford Beer [1985], two function models of in 't Veld [2002] and the 'control paradigm' of de Leeuw [1982].

2.4. Conceptual system models

2.4.1. The Formal System Model

Checkland calls a conceptual model, being part of SSM, a Formal System Model (FSM). Macauley [1996] defined such a FSM for a system with human activities. The model is also called a HAS (Human Activity System) model. According to Macauley the system S is a formal system if and only if it meets the following criteria:

- S must have some mission.
- S must have a measure of performance.
- S must have a decision-making process.
- S must have elements, which mutually interact in such a way that the effects and actions are transmitted through the system.
- S must be part of a wider system with which it interacts.
- S must be bounded from the wider system, based on the area where its decision making process has power to enforce an action.
- S must have resources at the disposal of its decision making process.
- S must either have long term stability, or the ability to recover in event of a disturbance.
- Elements of S must be systems having all the properties of S (except the emergent property of S).

Such a formal system is presented in figure 2.2.

Although a system makes expectations on what the subsystems should do, the subsystems do have a certain degree of autonomy about how these expectations will be realized. This

"what-how" relation is a common principle in all systems approaches and is called "hierarchy'. Hierarchy should not be confused with authority, it should rather be considered a way of "nesting".

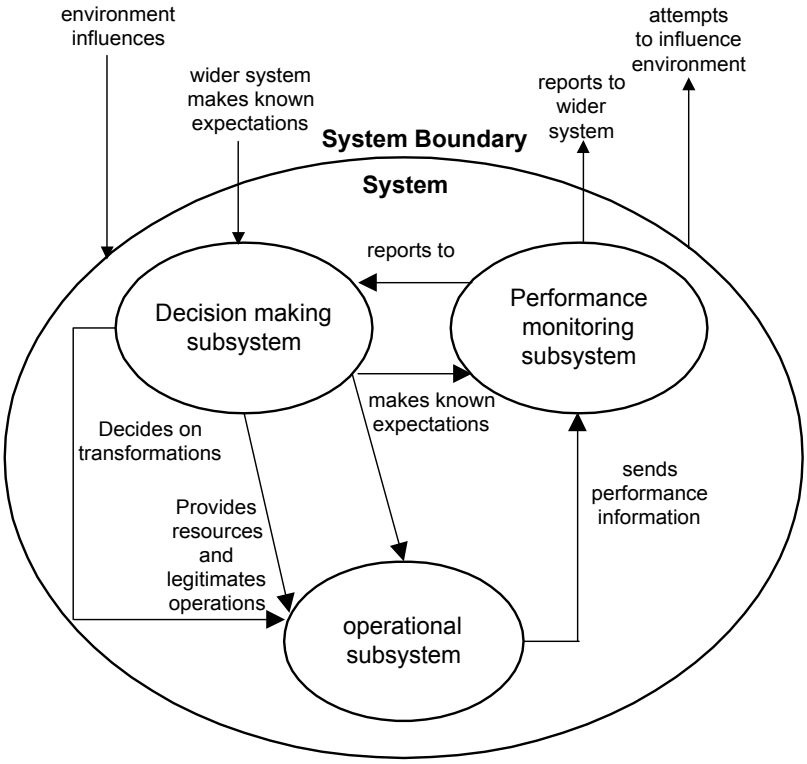


Figure 2.2: *A Formal System Model of a human activity system [Macauley, 1996]*

The elements of a system in this conceptual model are systems in themselves or (better still) subsystems. Three types of subsystems are distinguished: decision-making, performance monitoring and operational subsystems.
The model does not contain the "products" of the system.

2.4.2. The Viable System Model

The Viable System Model (VSM) of Stafford Beer [1985] consists of functions, which have to be present in any viable system; these systems all have the same pattern of functions. But this pattern should not be considered an organization structure. This function pattern forces the researcher to leave the existing structure, which is usually represented by an organization chart, and it enables the researcher to view the organization from a different perspective.

VSM's basic idea is that in order to be viable, organizations must maximize the freedom of the participants within the practical restrictions of the requirements to let the organization realize its objectives.

Beer distinguishes five functional groups. He calls them systems, because each functional group is a viable system on its own (see figure 2.3).

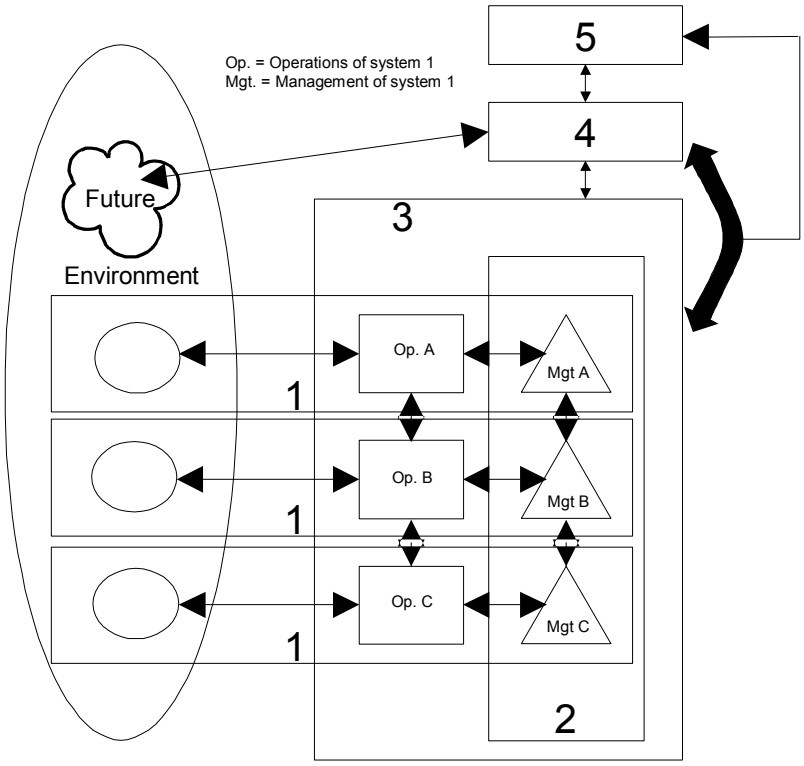


Figure 2.3: *The Viable System Model [Beer, 1985]*

The systems are:

1. System 1: Implementation consisting of execution, operational management and environment
2. System 2: Coordination of operational systems. Beer illustrates it with a school timetable. It is a service that ensures that a teacher has only one lecture at one time and that only one class uses a particular room for a given lesson. This service does not affect the content of the lecture - that is the concern of the teacher - but facilitates the operation of the school. A production schedule for a manufacturing utility is also one such system.

3. System 3: Internal control. This system preserves the purpose of the organization, "here-and-now".
4. System 4: Intelligence deals with the future environment: "there- and-then". It makes propositions to adapt the purpose. At the organization level research departments and marketing typically belong to this system.
5. System 5: Strategy and Policy. This system is responsible for the direction of the whole system. "System 5 must ensure that the organization adapts to the external environment while maintaining an appropriate degree of internal stability." [Jackson,1991].

Again, the model is recursive, each system being a complete VSM in it's own right.
The systems do not necessarily correspond to task descriptions, and several systems can be performed by just one person or department. Just as a FSM, the VSM does not contain the products of the system.

2.4.3. The function models of in 't Veld

In 't Veld [2002] developed two models to analyze organization systems:

- the "steady state model", which describes the *functions* needed to permanently realize a fixed objective in the proper way by a repetitive process .
- The innovation model, which describes the functions needed to determine the right objective(s) and to correctly implement the processes to achieve it.

The steady state model (figure 2.4) consists of a structured set of functions, expressing that contribution, which is repeatedly delivered to the environment in a controlled way but not how this contribution is achieved in a concrete way. The set of functions is itself a function again, which makes the model recursive. As mentioned before, the term "function" should not be confused with the common notion of "task description".

Example 2.3. A dynamo of a bicycle transfers mechanical energy into electricity by using the rotations of the wheel; this is a task description for the function "to produce electricity". The function can be fulfilled in many other ways (for example by a battery). A task description is device specific.

The model was primarily meant for analyzing organizational problems, but turned out to fit the construction of new organizations as well. The complete structure of functions is quite complicated. The model shown in figure 2.4. is a simplified version, where functions are grouped according to their relation with control [Veeke, Ottjes, 2000]. The detailed original model is shown in appendix A.
The steady state model contains:

1. A transformation function, by which input is transformed into a desired output.
2. An input zone to identify, qualify and quantify the input flow. If the functions in this zone are correctly fulfilled the transformation function only receives the proper input at the proper time.
3. The output zone analogously qualifies, quantifies and identifies the output to be delivered to the environment. If the functions in this zone are correctly fulfilled, the function as a whole only delivers the proper output at the proper time.
4. The intervention zone corrects disturbances in input, throughput and output by means of feedback and feed forward and by repairing deficiencies. Disturbances can only be experienced if standard values are defined. The actions of the intervention zone are directed towards these standard values.

Example 2.4. 600 Containers must be unloaded from a container ship; the standard time taken to unload the ship is 10 hours (standard value 1). Each quay crane unloads 30 container per hour on average (standard value 2). So 2 quay cranes are assigned to this operation. If after 6 hours of operation the number of containers unloaded is less than expected, an extra quay crane is assigned. This is an example of feed forward based on a disturbance in throughput.

5. The initiation - evaluation zone delivers the standard values for the intervention zone with respect to the facets quality, quantity and lead time; it is the translation of the requirements entering the function from a higher echelon, into manageable rules and standard values for these facets. Performance monitoring data are evaluated and reported to the higher echelon. In case of structural deviations rules and standard values are corrected.

Example 2.5. From repeated monitoring of unloading ships in example 2.4, it appears that an extra quay crane had to be assigned regularly. Now two possible decisions

can be made: decrease the standard value for the quay cranes to 25 containers per hour or increase the berth time of ships. The latter will probably not be accepted by the environment.

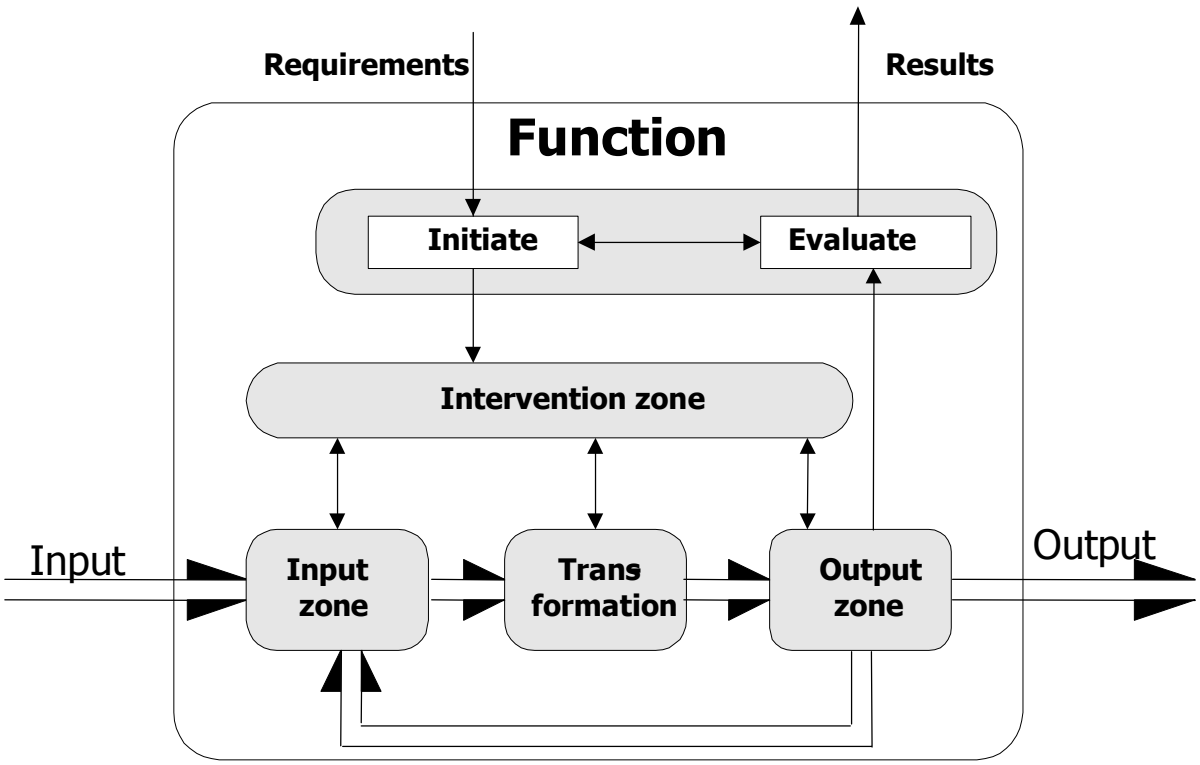


Figure 2.4: *A schematic steady state model*

The steady state model is recursive: each element in the structure is a function in it's own right. By repeatedly zooming in on a function, a hierarchy of functions will arise.

A "function" is an abstraction of what happens by consequently describing why it happens. Opportunities for reorganization become noticeable by describing an organization in terms of functions. Fulfilling a function can be assigned to several jobs and one job may fulfill several functions.

In 't Veld distinguishes subsystems and aspect systems: a subsystem includes a subset of the elements, but all relations; an aspect system includes all elements, but only a subset of the relations. An aspect system covers a particular flow of elements. The steady state model is a model for one single aspect where the elements are functions. Examples of an aspect system are the product flow, the job flow, the resource flow and the data flow.

Several aspect models are required to completely model a system. Up until now there is no conceptual model available for the relationship between aspects.

The use of the steady state model is restricted to the parts of an organization, which maintain a stated objective: the operational system. There's no function in the model to change the purpose or innovate the transformation. For this purpose in 't Veld developed the innovation model (figure 2.5). This conceptual model shows what functions are needed to define a new operational system.

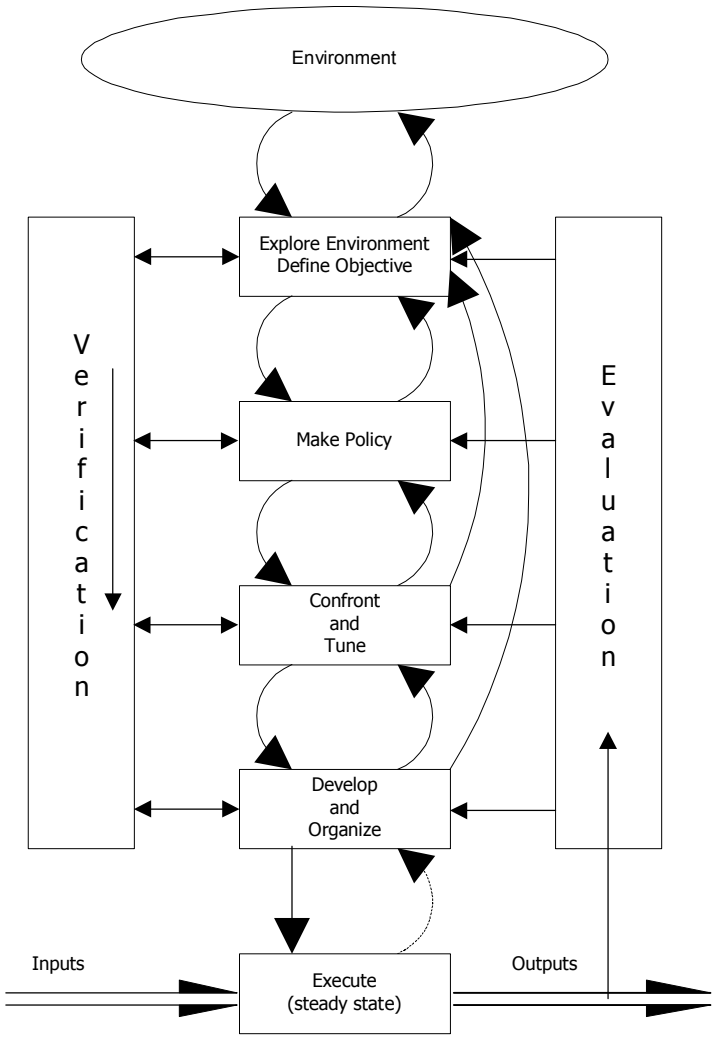


Figure 2.5: *The innovation model [in 't Veld, 2002]*

Innovation starts with the exploration of the needs of the environment and with the determination of the stakeholders. This will result in the definition of objectives and preconditions. During the next step, a policy is formulated stating the priorities among the objectives and the ways and means to achieve the objectives. These "wishes" must be compared and tuned with the prospects of the organization. Finally the new system must be

developed and implemented, resulting in a new operational system, being a "steady state" again.

The functions of the innovation model are executed in an iterative way. The innovation as a whole must be controlled. The downstream control is called verification (consisting of a master plan and programming). The main question is: "are we getting the innovation right?". The upstream control (starting after implementation) is called evaluation with the main question being: "did we get the correct innovation?".

2.4.4. The control paradigm of de Leeuw

The control paradigm (figure 2.6) of de Leeuw [1982] consists of a controlled system and a controller. Both the controlled system and the controller interact with the environment.

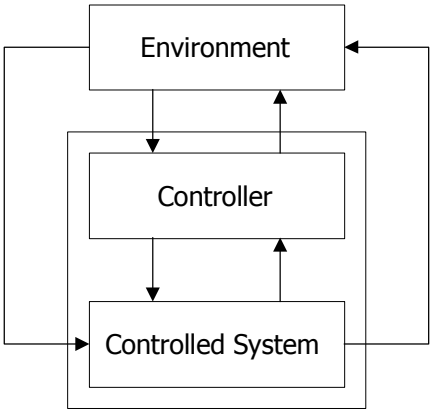


Figure 2.6: *The control paradigm [de Leeuw, 1982]*

Control is defined by de Leeuw as any type of directed influence. So the controller should have an objective. The controller controls certain aspects of the controlled system. It has to be capable of observing these aspects and of comparing these observations with standards or reference values for these aspects. The reference values are derived from the implicit or explicit objective of the controller. The controller may decide to specific control actions as a result of the comparison. De Leeuw describes a number of conditions for the control to be effective:

1. There must be an objective.

2. The controller must have a model of the controlled system to predict the effect of control actions. During the process of control this model can be refined.
3. The controller needs information about the environment and the state of the controlled system.
4. The controller needs sufficient possibilities to control.
5. The controller must have sufficient capacity for information processing at its disposal.

Control actions can be executed both directly (internal control) and indirectly through the environment (external control).
For both types of control de Leeuw distinguishes:

1. Routine control: actions within the scope of the current model of the controlled system.
2. Adaptive control: actions, which change the structure of the controlled system.
3. Strategic control: actions, which result in a change of the objective of the controller.

The control paradigm restricts itself to control and is not concerned with input to output transformations.

2.5. Conceptual modeling for logistics

2.5.1. Introduction

All conceptual models described in paragraph 2.4. are related to open purposive systems and can therefore be applied to logistic systems. All models (except FSM) distinguish explicitly:

- functionality to maintain a stated objective (objective-keeping) and
- functionality to adapt to a new objective (objective-adaptation).

In this thesis these different functionalities are reflected by considering:

- a logistic system as an open system with objective-keeping facilities

- a design process as an open system that creates a logistic system on behalf of a desired objective-adaptation.

Using these definitions it is clear that the FSM, VSM and the control paradigm contain elements for both functionalities. Making this distinction between the elements results in table 2.1., which classifies the elements to belong to a logistic system itself or the design process of a logistic system. The steady state model of in 't Veld covers the objective-keeping facilities, while his innovation model covers the objective-adaptation facilities.

	Logistic system	**Logistic design**
FSM	All subsystems	N.A.
VSM	Systems 1, 2 en 3	Systems 4 en 5
In 't Veld	Steady state model	Innovation model
Control paradigm	Controlled system Routine control	Adaptive control Strategic control

Table 2.1: *Conceptual system models related to "logistic systems" and "logistic design"*

A hard systems approach concentrates on the logistic system with a stated objective, a soft systems approach emphasizes the design process where concurrent (and sometimes even conflicting) objectives arise as a consequence of different perceptions. Therefore the columns of table 2.1., also correspond to the boundary between hard and soft systems approaches.

In this thesis, the emphasis lays on the definition of a conceptual model for a logistic system that can be used during the design process. Such a model will be the desired result of the first steps of a design process and will serve as a common communication tool during the rest of the design process. It will be a common platform for developing a solution for the disciplines involved and it evolves together with the design of the system.

The characteristics of objective keeping functions in logistic systems will be described in paragraph 2.5.2., from which a conceptual model will be derived. Chapter 3 will focus on the design process and the role of the conceptual model therein.

2.5.2. Common characteristics of the conceptual models

The conceptual models of paragraph 2.4. have the following characteristics in common:

1. They are empty with respect to resources and tools. The models specify 'what' and 'why', but offer complete freedom with respect to physical interpretation. This physical interpretation is exactly the area of each of the disciplines involved (organization, technology and information).
2. All elements are systems and are called subsystems. This similarity of elements is a condition for correct modeling, because there will be no confusion of thought and zooming in and out is unambiguously defined (a system is composed of subsystems, and a set of subsystems is a system). This is the basic principle for being recursive, which is a characteristic of all models. Zooming in can be repeated until the appropriate level of detail for the objective of the research is reached.
3. Every system fulfills a function in its environment by satisfying a need. If the elements of a system are subsystems, then consequently, the elements also fulfill functions. This "functional approach" is applied to all models.
4. The border line between system and environment is not further defined in FSM, VSM and the control paradigm. In 't Veld however, considers the border line as a boundary zone, containing functions to fit the flow elements for transformation or delivery by the system.
5. All models clearly distinguish control (decision making, intervention) functions and operational functions, it is a "paradigm" indeed. This paradigm will be used from now on for all system models.
6. Only in 't Veld distinguishes "aspects" explicitly, where an aspect is defined as a subset of all relations. This is expressed by the choice of (horizontally) flowing elements, which must be transformed to fulfill the systems function. Simultaneously, there is a controlling (vertical) information flow within each aspect. This leads to the next statements:

    - The design of a multidisciplinary product requires an approach taking multiple aspects into account. Several models are required, one for each aspect. In 't Veld shows no strict rules for modeling the coordination and communication between aspects. Each aspect represents a repetitive process and therefore

this coordination and communication will also be a repetitive process with its own standards and interventions. In terms of the VSM, this process is positioned at the level of system 3.

- Each steady state model of in 't Veld transforms a different flow, one for each aspect, and is being controlled by an information flow. The transformation function of this information is part of the system, which transforms the need of the environment (input) to the functional performance (output).

7. All models with the exception of the steady state model contain functions only, they do not "produce" anything. The steady state model combines function and transformation. The transformation function (input and output zone included) delivers the output (product) by which the function is fulfilled in a physical way. This function is represented in VSM by the operational system (but without flowing elements). The input-throughput-output representation with elements flowing through a transformation function can be applied universally, even to control functions. The physical product there is the control action. In addition, both services and data (information) can be considered physical products. Transformation, input and output zone together will, from now on, be referred to as the *"process"*. The reports from controls to the environment (and higher echelons) will be called the information on functioning or simply the "*performance*". Therefore a system will be modeled by a "PROcess-PERformance"-model ("PROPER" model). Both the control flow and the product interact with the environment (incoming and outgoing arrows).

Evaluating the above characteristics, a conceptual model of an objective-keeping system has to meet the following criteria:

1. the system fulfills a function to satisfy a need of the environment and is purposive. A system will, from now on, be referred to by its function.
2. The elements of a system are subsystems, which inherit all the characteristics of a system again.
3. Each system consists of an operational subsystem and a control subsystem; the control subsystem translates a need to the specification of a required product and monitors the operational subsystem; the operational subsystem transforms input to the required output by means of a process.

4. The model distinguishes aspects, being subsets of the relations. Each aspect represents a product flow, so the model includes more than one product flow. The relations between aspects are maintained by a coordinating control system.

Conditions 2 and 3 are quite different. Condition 2 formulates the principle of "zooming", condition 3 can best be described by "opening" a system at a particular aggregation stratum[1]. A next lower aggregation stratum can only be reached by first opening a system, and then selecting a part of the functions to zoom in to. In this way, the principle of control and operation is preserved. For example the steady state model of in 't Veld is an "open" view of a system (function). One is now able to zoom in on each of the functions of the steady state model, thereby considering this function as a system again, but now at a lower aggregation stratum.

The resulting conceptual model for an objective-keeping system (the process-performance or PROPER model) is shown in figure 2.7.

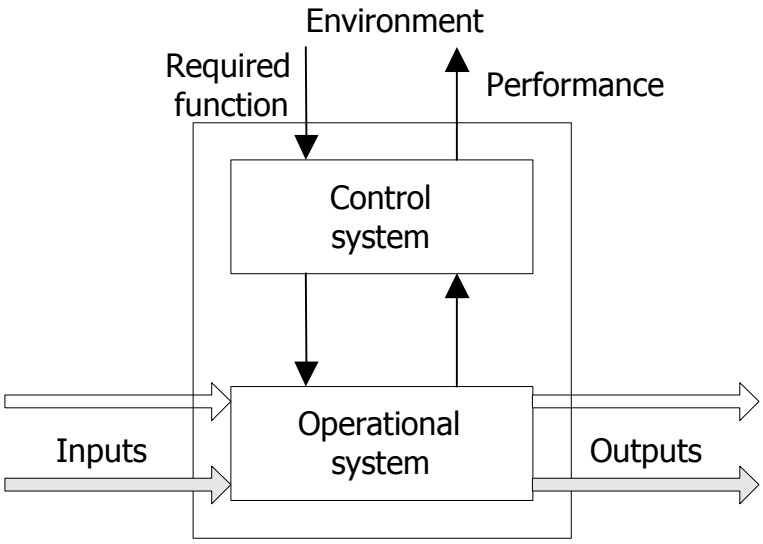


Figure 2.7: *The PROPER model of a system*

A function description must be determined by means of abstraction from the physical reality in order to construct a conceptual model. It is not important 'how' something is done, but 'what' and 'why'. Abstraction to functions offers two advantages:

a. it stimulates to be creative and to radically change the way of realization, either by different technological tools and equipment, or by combining functions in a different

[1] The term "aggregation stratum" is preferred here to express the "level of detail".

way. It is a structured pathway to reach innovation, next to the other path of 'accidental creation'. Example 2.6 illustrates this.

Example 2.6. Damen [2001] reports on a number of attempts at KPN Research that failed to make the mail process more flexible. Finally a research project started, which concluded that only 6 functions are needed to be able to describe any manner of mail processing: to transport, to store, to pack, to unpack, to merge and to split. Starting from this viewpoint, a completely new and flexible system has been implemented.

b. If the design is recorded in terms of functions, the basic assumptions and choices made during the design process remains clear and accessible for future design projects. This construction of 'memory' prevents the "invention of the wheel" once again and excludes the implicit assumption of superseded conditions (see example 2.7).

Example 2.7. During the design process of the highly automated Delta Sea-land Terminal (as a part of ECT in Rotterdam) the organization of the container "stack" (storage) appeared to be one of the most important factors for the performance of the terminal as a whole. It was shown that reshuffling containers during their stay in storage would significantly the effectiveness of the final transfer process. However, based on experience with manual container handling the following principle was implicitly applied: "a container once stored will not be moved until the moment of final export". The most important reason was the risk of container loss or damage. This risk was no longer under discussion with full automation and therefore this principle was no longer valid.

2.5.3. The performance of a function

The degree to which a system (and each subsystem) functions, should be measurable. The function description alone is inadequate for this purpose. Emergent properties that may, at least, be quantified or are qualified (by comparison), must be defined. They have to be derived from the objectives of the system. The system looses these emergent properties if functions are missing (a car won't drive without power function), or experiences a value

change of the properties if some functions are not properly fulfilled (worn down brake blocks decrease the braking power).

Bikker [2000] defines the next properties (he calls them "main process criteria"):

- *Effectiveness*: a measure of the result expressed as the ratio between expected and real results.
- *Productivity*: a measure of the efforts made to obtain the result. It is the ratio between results and efforts. Again expected and real efforts are compared. Productivity includes both results and efforts. The ratio between expected and real efforts alone is called efficiency.
- *Flexibility* includes product flexibility, production flexibility and innovation flexibility. The first two kinds of flexibility are usually expressed in terms of effectiveness and productivity: how large is the product mix, how fast can a product switch be established and how long (or rather short) is the lead time; these are design requirements for the logistic system. During operation in steady state, they are monitored and eventually cause a redesign. Innovation flexibility indicates the ability of the organization to decide on a (re)design and implementation.
- *Control* can be interpreted in two ways: controllability or control burden. Controllability is expressed in terms of effectiveness and productivity, for example by means of (the distribution of) delivery reliability. Control burden is considered a cost factor and is therefore part of productivity.
- *Quality of work* cannot be easily quantified. A common criterion used is the degree of autonomy for each job or department. This property can be assigned exclusively to the organizational aspect of a logistic system, but can be expressed in terms of the PROPER model. It concerns the combination of control and operational functions.
- *Innovation capacity* can only be measured after repeated execution of design processes and is therefore not considered further in this thesis.

From the above list of criteria three main properties result: effectiveness (E), productivity (P) and efficiency (C). The other properties can be expressed in terms of these properties or can not be quantified at all. Control, quality of work and innovation capacity rather belong to the specific discipline of organization.

The criteria effectiveness, productivity and efficiency are used during real operation of a system to evaluate the performance, but also during the design of a system to evaluate design alternatives. The properties are used for three different purposes [in 't Veld, 2002]:

1. strategic: to select between alternative resources[1]
2. tactical: to determine the best way to use selected resources
3. operational: to determine the operational performance

Ad 1. During the first steps of design resources must be selected. Only resources that are able to deliver the intended result are considered further. This is expressed by the "theoretical effectiveness" $E_{theoretical}$ in terms of results R by:

$$E_{theoretical} = \frac{R_{expected\_with\_the\_resource}}{R_{intended}} \qquad (2.1)$$

To make a selection between resources that all have a sufficient theoretical effectiveness, the theoretical productivity $P_{theoretical}$ is defined by:

$$P_{theoretical} = \frac{R_{intended}}{I_{expected\_with\_the\_resource}} \qquad (2.2)$$

where I represents the efforts (or investments).
The resource with the highest theoretical productivity will be selected.

Ad 2. A selected resource can be used in different ways. The best way is the one where the minimum feasible efforts lead to maximum feasible results with the resource. This "maximum feasible productivity" $P_{standard}$ will be the standard for the operation:

$$P_{standard} = \frac{R_{standard}}{I_{standard}} \qquad (2.3)$$

[1] By "resource" not only a physical piece of equipment is meant, but it can also represent an organizational structure.

Ad 3. During operation the standard productivity will not always be realised, due to disturbances. Both results and efforts may differ from the standards. This real productivity is defined by:

$$P_{real} = \frac{R_{real}}{I_{real}} \tag{2.4}$$

To evaluate the real productivity it is regularly compared with the standard productivity, leading to the definition of "Performance". This is shown by the following formula:

$$\text{Performance} = \frac{P_{real}}{P_{standard}} = \frac{\frac{R_{real}}{I_{real}}}{\frac{R_{s\,tan\,dard}}{I_{s\,tan\,dard}}} = \frac{R_{real}}{R_{s\,tan\,dard}} \times \frac{I_{s\,tan\,dard}}{I_{real}} = E_{real} \times C_{real} \tag{2.5}$$

Productivity, effectiveness and efficiency are therefore related by the formula:

$$P_{real} = P_{s\,tan\,dard} \times E_{real} \times C_{real} \tag{2.6}$$

Effectiveness and productivity can be quantified for each separate function at each aggregation stratum and can be defined separately for each aspect. For example response time is part of the effectiveness of an information system, reliability belongs to the effectiveness of a technical system etc. From now on the degree of functioning or "performance" will be expressed in terms of effectiveness, productivity an efficiency.

2.5.4. The "PROPER" model of logistic systems

The logistic system is a subsystem of the organization as a whole; it contains a subset of the elements, but includes all the relations. In this thesis, logistics is approached from the view point of the primary function and three aspects are included in the conceptual model:

1. the "product" as a result of a transformation.

2. The flow of orders; without customer orders no products will flow. In this flow orders are transformed into handled orders.
3. The "resources" (people and means) to make the product. To use them, they must enter the system and they will leave the system as used resources.

The results of the transformations are delivered products, handled orders and used resources.

The PROPER model of a logistic system is represented in figure 2.8. The aspects orders, products and resources will be represented by block arrows, single lined arrows will represent data flows. Usually the flow rate of resources is slower than the flow rates of orders and products. People can be employed long-term and means and equipment will usually be used during their economical life time.

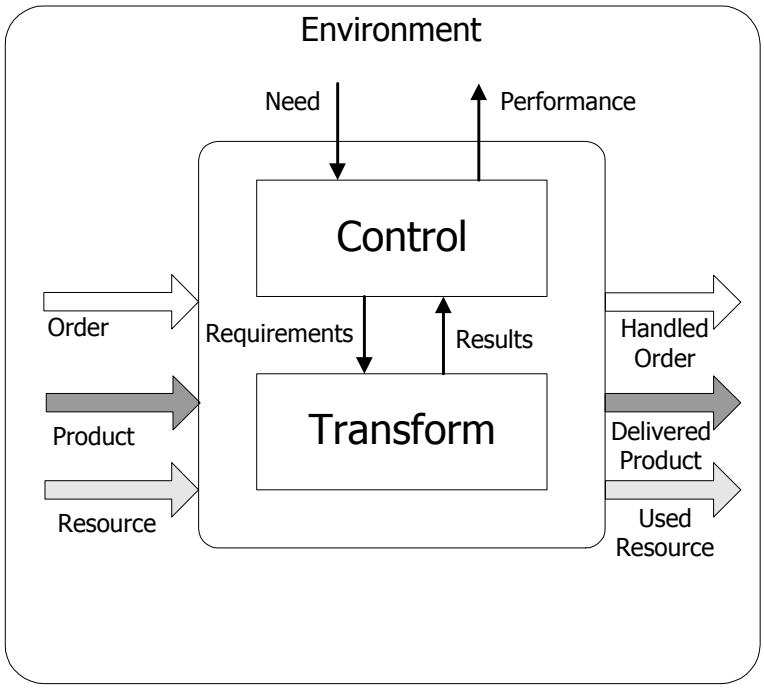


Figure 2.8: *The PROPER model of a logistic system*

The term "product" does not only imply a physical product. Especially in service industries or in the tertiary sector, a product can also be immaterial. For example, when applying the model to an "insurance" company the product can be a need for insurance to be transformed into an insurance.
Information flows both horizontally and vertically. Horizontally, the flow contains (mostly technical) data for the operation itself, vertically the flow contains control data.

Within the black box "transform" three parallel transformations are distinguished:

- The transformation of (customer) orders into handled orders ("perform")
- The transformation of products (raw materials) into delivered products ("operate").
- The transformation of resources into used resources ("use").

The control function coordinates these transformations by generating executable tasks derived from the orders and by assigning usable resources.

Opening the 'Transform' black box of figure 2.8 now results in figure 2.9.

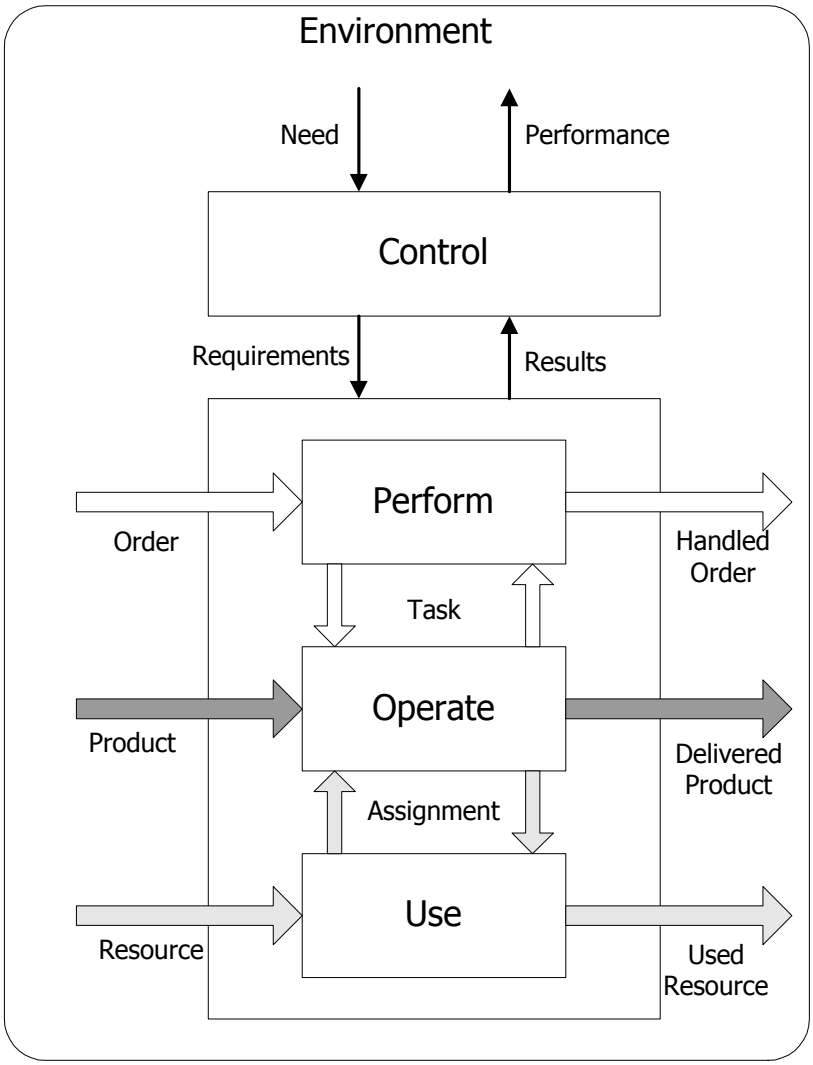


Figure 2.9: *The aspects in the PROPER model of a logistic system.*

Elements of several disciplines come together in the exchange of tasks and assignments, which is the main subject of communication during the design process. The communication

concerns requirements, derived from the function's objectives, and feasibility, determined by technological or information practicability.

2.5.5. The "PROPER" model and logistic practice

Up until now the modeling has been generic but quite abstract. It should however be possible to classify the common functions of logistics appearing in practice, into one or more of the chosen aspects.

Logistics is usually divided in material management and physical distribution ('outbound logistics') (see for example [Visser and Van Goor,1996]). Within material management they distinguish purchase logistics ('inbound logistics') and production logistics. Finally they add reverse logistics because of the increased attention to the flow of returning products. They argue that a logistic concept requires a coherent and iterative way of decision making about: the physical layout (the basic pattern), the corresponding control system, the required information and the matching organization. The basic pattern refers to the division of basic functions into subsystems. The control system complies with the control function of figure 2.9, while the information need points to both the horizontal and vertical information flows. The matching organization refers to the organ structure and to the structure of authority and responsibility.

During the last decade material management and physical distribution are integrated in "supply chain management". In terms of the systems approach integration means extending the systems boundary and considering the whole as a system to be controlled again. Towards the end of the nineties, the Supply Chain Council (SCC) developed a so-called reference model for this integrated approach to logistics: the Supply Chain Operations Reference (SCOR) Model. This model supports the evaluation and improvement of the performance of the supply chain, organization wide [SCC, 2002]. It emerged from the combination of Business Process Reengineering (BPR), benchmarking and process measurement. SCOR contains:

- All customer interactions starting from order entry up to paid invoice (see order flow in figure 2.9.)
- All material transactions (see product flow in figure 2.9.)

- All market interactions, starting with the determination of aggregated need up to the execution of each separate order (see control in figure 2.9).

The model describes four levels of supply-chain management:

- Level 1 contains five elementary management processes: Plan, Source, Make, Deliver and Return; The objectives of the supply-chain are formulated at this level.
- At level 2 the five processes are described more precisely by means of three process categories: 'Planning', 'Execution' and 'Enable'. The basic idea is that each of these three categories can be distinguished in each of the processes. The execution category of Source, Make and Deliver is further divided into 'Make-To-Stock', 'Make-To-Order' and 'Engineer-To-Order' types. In this way a complete scheme of 26 possible process categories is created. Any company is able to configure its existing and desired supply chain with this scheme.
- Level 3 shows, which information (and software) is needed to determine feasible objectives for the improved supply-chain.
- Finally level 4 addresses the implementation. Level 4 changes are unique, so specific elements are not defined; only guidelines and best-practices are described.

SCOR is a reference model: contrary to a conceptual model it classifies all logistics activities; it aims to improve rather than to innovate. All existing configurations can be modeled, current non-existing solutions cannot, as it turns out from the most recent addition of 'return'.

Plan, source, make, deliver and return are management processes and are part of a control function. To determine, which control function they belong, a short explanation of each is given below.

Plan: is demand and supply planning and management. It balances resources with requirements and establishes/communicates plans for the whole supply chain. It manages business rules and supply chain performance.

Source: takes care of the supply of stocked, make-to-order, and engineer-to-order products. It schedules deliveries, receives, verifies and transfers products, it manages inventories,

capital assets, incoming products, supplier networks, import/export requirements, and supplier agreements.

Make: concerns the execution of make-to-stock, make-to-order, and engineer-to-order production. It schedules production activities, manages in-process products (WIP) , performance, equipment and facilities.

Deliver: covers order, warehouse, transportation, and installation management for stocked, make-to-order, and engineer-to-order products. It includes all order management steps from processing customer inquiries and quotes to routing shipments and selecting carriers. It also includes all warehouse management from receiving and picking products to load and ship products.

Return: is the return of raw materials (to supplier) and receipt of returns of finished goods (from customer), including defective products, and excess products.

Comparing these descriptions with the functions and aspects of the PROPER model of figure 2.9 shows that:

- There is no strict distinction between aspects in SCOR. Source, make and deliver in particular contain parts of each aspect; to put it differently, each aspect contains a source, a make and a deliver process.
- The control of the product flow is split between make and deliver. Make takes care of stocks-in-process, while deliver emphasizes warehousing at receipt and shipping.
- Plan contains both the long-term planning and balancing, and the daily coordination of the flows.
- Return represents a complete product flow. In terms of the PROPER model it is a subaspect within the product aspect and it can be studied separately using the PROPER model.

As argued before the distinction between aspects is important, because they reflect disciplinary backgrounds and perceptions. It must be clear whether decision-making concerns the order flow, the product flow or the resource flow, in order to enable correct objective settings. Each flow is controlled by its own control system coordinating the source,

make and deliver control functions. Each flow oriented control system again must be coordinated with the other aspects by a control function at the next higher echelon.

Including the basic processes source, make and deliver finally leads to the PROPER model for each aspect of a logistic system as shown in figure 2.10.

The definitions of Visser and van Goor [1996] for purchase logistics, production logistics and physical distribution can be mapped one-to-one to source, make and deliver respectively. It is remarkable to see that the field of logistics is divided into functional areas instead of flow oriented areas. Terms like order logistics, product logistics or resource logistics are not encountered in logistic concepts.

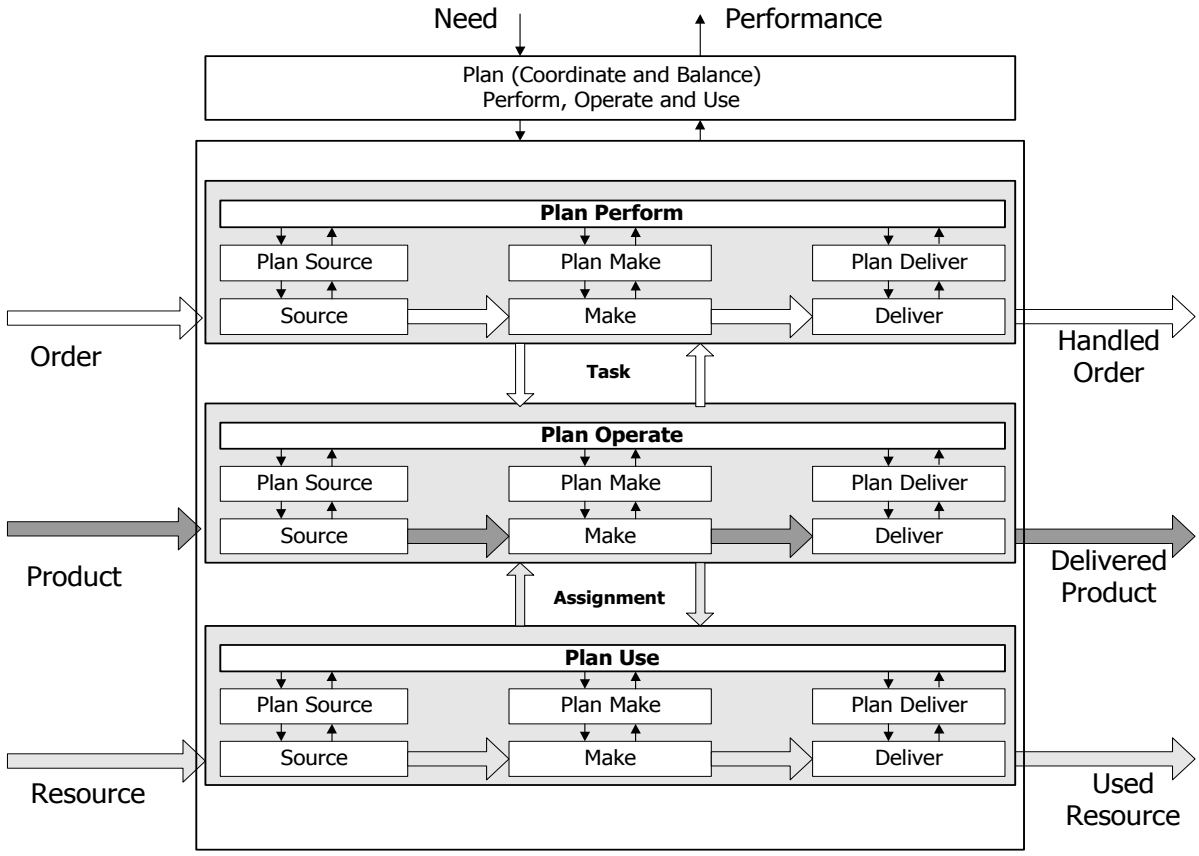


Figure 2.10: *Functions for the aspects in the PROPER model of a logistic system*

Well known concepts like Just-In-Time and KANBAN can now be positioned in the make process of the product aspect. A control concept like Manufacturing Requirements Planning can be positioned at the level of plan Operate etc.
The position and use of this model in the design process can now be investigated, which is the subject of the next chapter.

Chapter 3

The functional design of logistic systems

"To the man who only has a hammer in the toolkit, every problem looks like a nail."
-Abraham Maslow-

3.1. Introduction

In the previous chapter, the process – performance model (PROPER) for a logistic system has been defined. This model represents the aspects orders, products and resources that arise interrelated for discussion during the design process, not only in operational respect but also in control respect. In this chapter the use of the PROPER model during the design process will be examined.

In the course of the design process, the level of detail of the model increases continuously by succeeding cycles of problem formulation and solution. The elements of the model remain PROPER models on different aggregation strata during this trajectory. As such, the model is a frame of reference for all disciplines involved. The model plays an important role in problem formulation: each solution or each decision formulates new problems, in more detail or tapered to one or more subsystems or aspects.

The design process itself will be examined first. The nature of design has changed significantly not only as a result from a changing market, but also as a result from the developments in information technology. The accent of the process has shifted from "streamline the existing" to "start from scratch" in order to make use of the new opportunities.

In this chapter, the application of the PROPER model in the design process will be explained by means of the innovation model of in 't Veld (see paragraph 2.4.3). The elaboration of the PROPER model is represented then as a process composed of two clearly distinguished steps: function design and process design. Each of these design steps is composed of succeeding cycles, where each cycle determines the "context – function – structure – behavior" in this order. At the change from function design to process design, the phrasing of questions changes from "what?" and "why?" to "how?". This transition coincides with the

transition from generic PROPER modeling to specific modeling of each discipline. The transition will be illustrated for the fields of organization, technology and information.

The PROPER model will always be required for communication on and feedback of detailed designs. The use of the model supports the structured recording and evaluation of elaboration and changes. This satisfies the major condition to construct "shared memory", from which future design processes can draw again. Shared memory starts with "shared meaning" and the PROPER model plays a major part in this. [Konda et al., 1992]

At the end of this chapter, it will be explained in which steps of the design process simulation can contribute to decision making. It is the introduction to the modeling of time dependent aspects of the logistic system.

3.2. The design process

The pressure to cut down costs, to reduce lead times, to grade up quality and in general to enhance the effectiveness and productivity of logistic systems, has increased enormously during the last decades. As a result, the importance of "(re)engineering" grew more and more [Davenport, 1993]. The number and possibilities of tools for design expanded simultaneously by the development of information technology. For example, Computer Aided Design (CAD) and computer simulation are common property now. Above that, information technology also changed the contents and structure of logistic systems. The increased speed of communication influenced decision making processes in a logistic system, and the technological tools became more advanced by the use of automation. Therefore, currently designing starts more and more from scratch with the risk of errors being repeated or re-inventing the wheel. To rule out this risk, design decisions and principles should be recorded in terms of the PROPER model, since the model is conceptual and fundamental with respect to functions and processes. Therefore, this model has to be positioned in the design process. To determine this position, the characteristics of the design process of a logistic system have to be described by means of a model.

Several conceptual models for a design process have been developed. From a classification by Konda et al. [1992], it is concluded that these models mostly are developed from the viewpoint of a single discipline. There are models for the design of technical systems or

products, for information systems and for organizations. A logistic system however covers all these aspects.
Currently, a design process is not restricted to the original subject of design (the product) itself. In all disciplines, the 'making of' the product and its use is also included from the very start. In the design of products the (efficiency and effectiveness of the) manufacturing process is considered and in the design of a technical system its use in the production (e.g. for maintainability). The design process of information systems includes the implementation trajectory and the maintenance, and the design of an organization is not complete without its institution. Terms like 'concurrent engineering' and 'life-cycle engineering' all refer to this tendency of integration of different product aspects.
The common starting point for all design processes is usually the function to be fulfilled. After that the function is detailed and finally concretized for implementation.

In order to support an interdisciplinary approach of the design of a logistic system, a step preceding the different (parallel) design trajectories of each discipline is required. In this thesis, the design of a logistic system is considered a combination of the design of a product (the logistic system itself) and the design of a process (the way in which the logistic system works).
The product design concerns the determination of the objectives of the system. Central questions in this process are "what is feasible?", "what is required?", "what functions are to be fulfilled by the system?". It is a strategic decision process and it will be denoted by the "*function design*" from now on (see paragraph 3.3.).
The way in which the system will work, concerns the definition of structure, processes and resources and it will be denoted by "*process design*" (see paragraph 3.4). Process design deals with the optimal utilization of resources, technology and information and belongs therefore to the field of tactical decision-making.
Function design covers the performance part of the PROPER model, while process design covers the process part.

Function design makes the difference between innovation and improvement. Usually improvement only concerns the reorganization or reengineering of an existing system, which implies a rearrangement of functions or a different interpretation of functions. Innovation concerns the extension, reduction or change of functions as a consequence of the introduction of new technology, resources and/or organization.

Jonas [1997] even distinguishes 3 steps in the design process: analysis, projection and synthesis (see figure 3.1). Jonas states: " Transforming a vague feeling of discontent into a solution turns out to be a 3-step process of reducing uncertainty (contingency). The traditional concept of industrial design neglects the first two steps and acts at the very end of the process". The function design as mentioned before, corresponds to the first two steps according to Jonas. After these steps, a "problem" is formulated that can be used for process design. During the first step (analysis) a feeling of failure or discontent (e.g. "the market asks something different") is translated into a number of possible reasons (e.g "we are too expensive", "we don't make the right product", "we don't deliver in time").

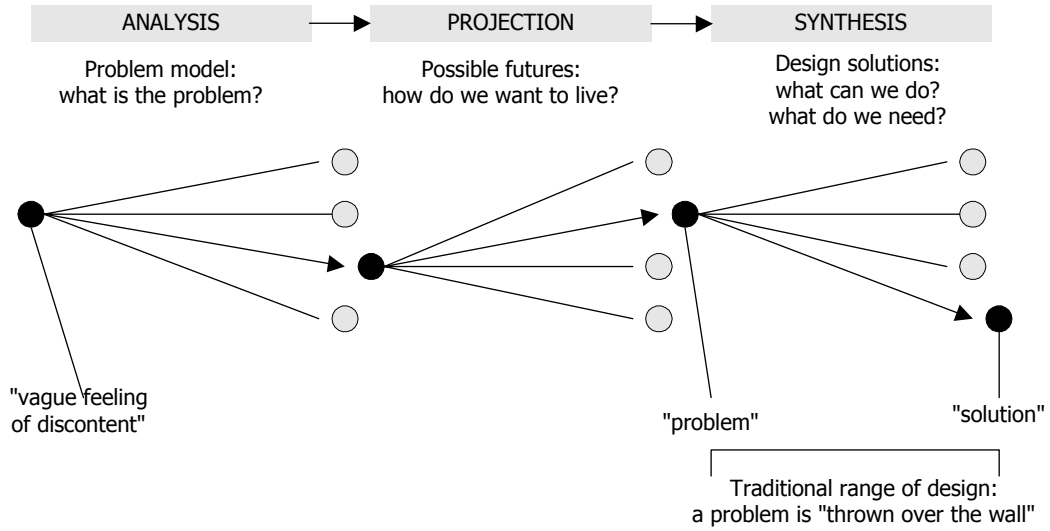


Figure 3.1: *Design as a 3-step process of "problem solving"*

In the step called "projection" the feasible and desired possibilities are investigated that may remove these reasons ("expand", "shrink", "innovate processes", "innovate products", "automate"). Choosing between these possibilities results in the definite problem formulation for the design trajectory.

Both the division of the design process in function design and process design, and the division in analysis, projection and synthesis are plotted in relation to the functional steps of the innovation model of in 't Veld as described in paragraph 2.4.3. For convenience of comparison, the environment and the functions 'verification' and 'evaluation' are omitted (see figure 3.2.). At the right hand side of the figure, the result of the design process is represented: an operational logistic system. This system is represented by a PROPER model. The second step of the innovation model is called "define alternatives" to emphasize the iterative character of the process. In 't Veld used "make policy" originally, but a policy (or

selected alternative) is achieved after several iterations via confront and tune and (provisional) developing.

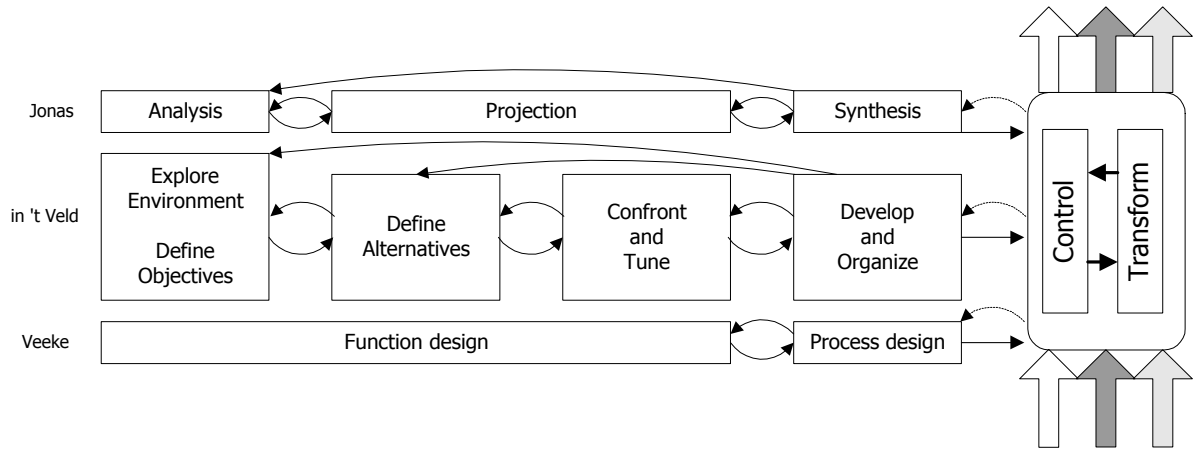


Figure 3.2: *The innovation model as a design process*

The figure shows that function design encloses the steps 'explore environment and define objectives', 'define alternatives' and finally 'confront and tune'. The process design takes place in the step 'develop and organize'.

Most literature on design concerns process design only. During function design however it is determined what can be required at all from the system and this is the starting point for the process design. This distinction fits well to the different performance indicators as described in paragraph 2.5.3. During function design the strategic indicators theoretical effectiveness (formula 2.1.) and theoretical productivity (formula 2.2.) should be determined. This cannot be achieved without iterating with parts of the process design (albeit provisional), because the efforts to be expected are to a large deal due to the resources. Having determined the intended results, the results to be expected and the efforts to be expected, a definite configuration can be selected, which will be elaborated in full detail during the process design.

Example 3.1. The Delta Sea-land Project (DSL) of ECT was a design trajectory to develop a largely automated container terminal. This project was preceded by several years of research to determine both the resources and technology that were most appropriate and feasible to reach the intended results. In those years of research for example several alternatives were investigated for the quay transportation system, such as rail-bound transportation, conveyors and free ranging automated vehicles. In the end a system with automated vehicles was selected based on requirements of flexibility, productivity and technical feasibility. How to

construct these vehicles (with respect to automation and technology) and how to control them – and even which part of the sea sided functionality they should fulfill- was not yet determined at that time. It was clear however, that this system could offer the best theoretical effectiveness and theoretical productivity.

3.3. Function design

Function design aims to determine a system configuration that is able to provide a required performance in an optimal way i.e. to provide the intended result with acceptable efforts. For a logistic system, this can generally be expressed by: "offer the required service with acceptable costs" [Lambert, 1997]. During function design, the system configuration should be tested for feasibility and desirability.
The steps of function design will be denoted by the functional terms of the innovation model of in 't Veld.

Step 1. Explore environment and determine objective

During function design the environment mainly consists of the customer market, the society, the company to which the system belongs and the chain to which the system belongs. By exploring the environment, the need of the environment (the required service) is determined.
This need is translated into objectives, preconditions and principles. Preconditions fall outside the range of influence of the system and are firm restrictions for the rest of the design trajectory. Examples are statutory regulations, environmental laws and the like. Principles are conditions drawn up by the company's culture. They can be influenced but changing a principle goes beyond the scope of the system to be designed and often involves high costs.

Objectives are expressed in terms of the PROPER model:

- regarding the order flow: what is the demand composed of, what is the required lead time and what is the required reliability of delivery?
- Regarding the product flow: what are the products required, what is the required quantity and quality, what are the costs and flow times?

- Regarding the resource flow: what is the required quality and quantity of resources with respect to the order and product flows?

For the relation between the flows this leads to questions as:

- between order and product flows: what is the ratio between order quantity and product quantity?
- Between resource and product flows: what is the required flexibility of utilization?

The objectives for the order and product flows strongly influence the resource flow. In cases of (a.o.) large seasonal influences (e.g. sugar industry), continuous operation (e.g. service departments) and dangerous work (e.g. chemical industry) social factors play a major role. In addition, the increasing degree of automation did enable new modes of operation and working methods, but was not always welcome and has led to considerable commotion with innovations.

At the end of this step the objectives are expressed in terms of intended results.

Stap 2. Define alternatives

During this step, a number of alternative configurations is being determined that may satisfy the requirements. The definition of alternative configurations is an alternation of thinking creatively and structuring. Creativity does result in new ideas, but the result of it can and should be reflected in terms of the following structured approach.

The definition of alternatives is an iterative process itself, which can globally be defined as a repetition of cycles consisting of context determination, function determination, structure determination and behavior determination [Ackoff, 1971]. Each succeeding cycle takes place on a next aggregation stratum. The following table shows the contents of each part of a cycle.

Context determination	In terms of the systems approach, this is the determination of the system boundary. The system is considered a part of a larger chain. Upstream the

	system can be expanded into the direction of suppliers, downstream into the direction of customers. Shrinking the system is also possible. *System concepts* will result, in which different configurations can be selected. Each alternative has a primary function, based on the objectives.
Function determination	The primary function is divided into subfunctions (zooming). For each subfunction the intended result is defined.
Structure determination	Subfunctions can be particularized into both horizontal and vertical direction. Horizontally there are two major ways: - specialization / parallellization: the flows are being split or combined - differentiation / integration: the functions are being split or combined. This step is of vital importance, because splitting and combining flows and functions influences both the results and the efforts that can be expected. Splitting functions or flows generally results in increased costs (extra transfer actions, increased space requirements and the like). Combining flows may also lead to increased costs (complex technology, turnaround costs, extra sorting etc.). Particularizing into vertical direction concerns the control structure. Control echelons are introduced and the degree of autonomy for each function group is determined. This type of structuring indicates the controllability and control burden (the need for control) As a result a number of *structure concepts* are defined. For each structure the results that can be expected are to be determined.
Behavior determination	Each structure causes a 'behavior'.

	This shows itself on the one hand in communication and consultation requirements, on the other hand in time dependent phenomena with respect to the contents of a structure (stocks, throughput times etc.). This step therefore determines the expected behavior: a *behavior concept*. Behavior also influences both the results and the efforts to be expected. Each structure shows its own specific behavior.

Table 3.1. The cycle of determining alternatives

Table 3.1. shows that a feeling of discontent is translated into configurations consisting of a system and structure concept with a corresponding behavior concept. They lead to a problem formulation for process design. In terms of Jonas's model the analysis yields system concepts, and the projection phase yields structure and behavior concepts.

The PROPER model is the basis for the definition of system and structure concepts.

During the step of defining alternatives each of the disciplines involved should evaluate the feasibility of a configuration from its own perspective. The model is not bound to a specific discipline and reflects clearly the environment, functions and structures. It is considered a "cognitive map" of the design. According to Dwyer [1998] such a map is "a system model from the perspective of how people involved with it will understand it". He further states that "systems incorporating human beings must be designed with the cognitive properties of humans in mind".

The determination of results to be expected is a part of process design already, albeit provisional. By means of draughts, prototyping, based on experience and by means of simulation each discipline contributes to this determination. This illustrates the iterative character of designing.

If a system or structure concept is considered infeasible (the results to be expected don't match up to the results to be expected), this concept will not be elaborated further. For the

remaining configurations the results in view and the results to be expected are specified now to the stratum of subfunctions.

The PROPER model does not reflect the behavior concept. It is a static model of the system. Step 3 will address this further.

Step 3. Confront and tune.

Having the intended results and results to be expected, this step aims to determine for each configuration the efforts to be expected. For this purpose, the process component of the PROPER model is examined now. A process takes time and capacity (costs). Confrontation means for each discipline involved (in this thesis they are technology, organization and information) the separate assessment of "can we do this?" and "do we want this?". Subsequently, all disciplines together try to tune to one another with any adjustments. For illustration purposes, common questions for each of the disciplines, which are to be answered for each alternative, are formulated in table 3.2.

Technology	Are the grouped functions technologically feasible? What kind of hardware (and developing time and capacity) is required? What are the consequences with respect to operations, maintenance and environment.
Organization	What will be the departments? Are we able to and do we want to realize these within the existing organization? What are the demands on competencies of people and other means? Can they be obtained here or elsewhere? What education efforts are required?
Information	What are the demands on architecture, software and hardware? What administrative systems, control systems and production support systems are required? Are we able to develop the systems required in house or should we hire capacity for this?

Table 3.2. Confront and tune by discipline

The results of 'confront and tune' include a specification of results and efforts to be expected for each alternative. The alternative with the maximum theoretical productivity is selected first. The other alternatives are kept for the event that during process design this alternative still does not match the expectations.

For the determination of the theoretical effectiveness and productivity, an understanding of the behavior of the system is required. For this purpose, simulation is an outstanding tool. Even without concrete resources defined, simulation enables the evaluation of expected behavior of the processes in the PROPER model. The behavior of functions can also be represented. In chapter 4 the notion of "behavior" will be defined exactly.

3.4. Process design

3.4.1. Introduction

Process design starts at the point where a configuration is selected. The selection may be provisional as an iteration with the confront and tune function, where different configurations are being compared. The system has to be developed and organized now. By function design the structure of functions to be fulfilled, is reflected including intended results and (a first estimate of) results and efforts to be expected. These values are the target figures for the optimal process design.

Jonas [1997] notices that this is the territory of the traditional design approach. Process design is rather a multidisciplinary / monodisciplinary than an interdisciplinary trajectory. The methodologies of all disciplines currently contain a stage called 'conceptual design', probably based on the multidisciplinary requirements of the environment to which the methodology is applied. The conceptual design shows a large overlap with the function design of the preceding paragraph. The functions defined however usually cover only one single aspect or subsystem of the PROPER model and use a terminology that originates from the discipline itself. Representing this by the innovation model structure of figure 3.2 the multidisciplinary approach results in figure 3.3.

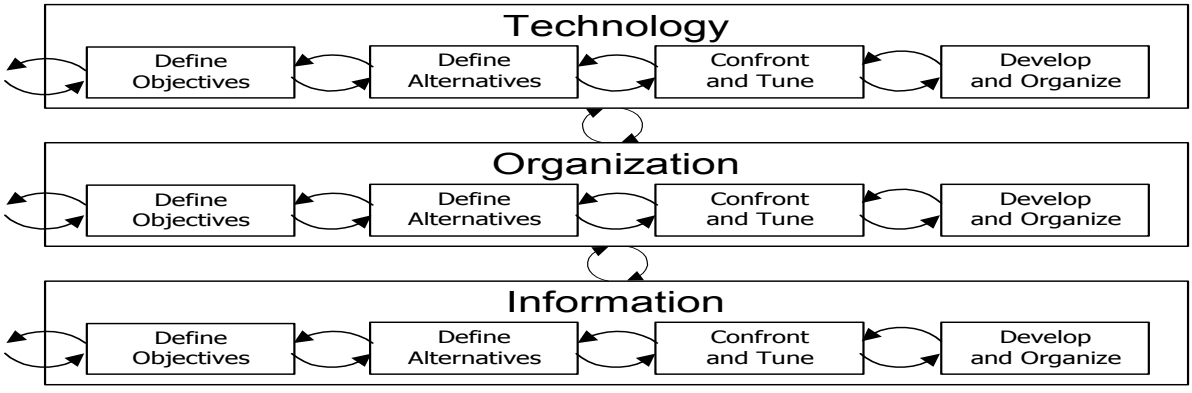


Figure 3.3: *A multidisciplinary design approach*

The interdisciplinary approach used so far here takes all aspects and a consciously chosen system boundary into account during the function design. The result is shown in figure 3.4.

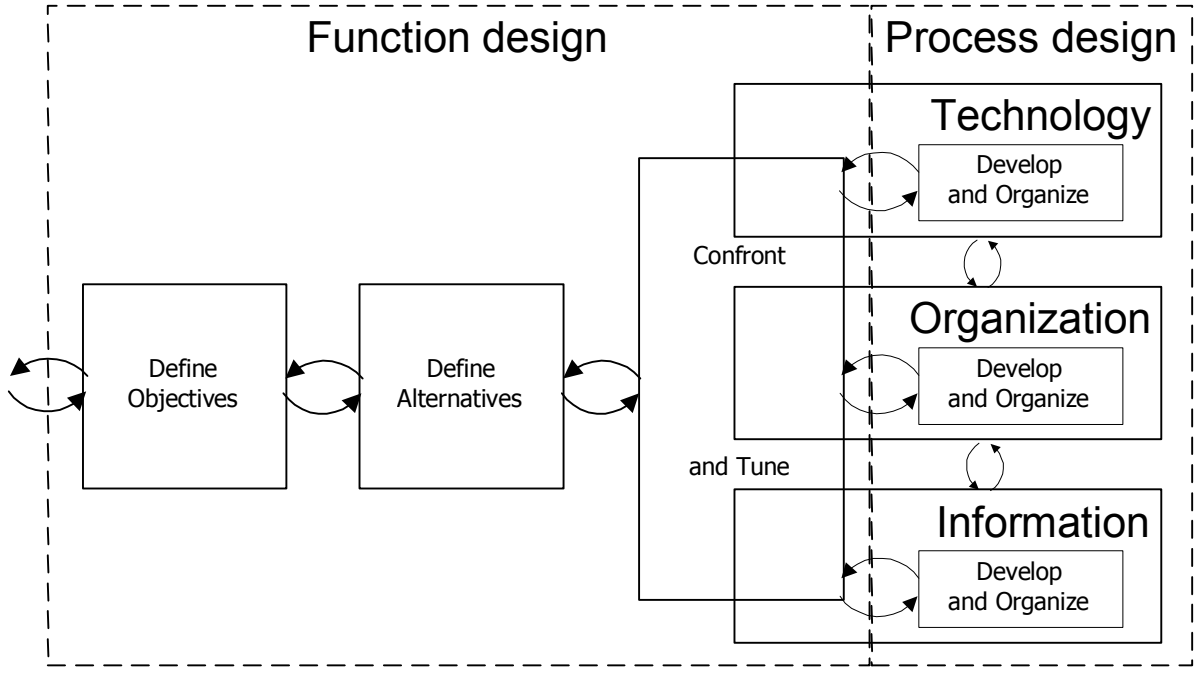


Figure 3.4: *The interdisciplinary design approach.*

For each of the disciplines technology, organization and information the characteristic design steps will be described shortly and the connections between function design and process design will be illustrated in the following paragraphs.

3.4.2. The design of technical systems

The technological process design mainly focuses on the resource flow of the PROPER model. It concerns the design of machines, tools and transportation equipment. During function design the function grouping was established and they should be physically filled in here. Doepker [2001] distinguishes the following steps:

1. *Establishing needs and generating ideas* resulting in requirement specifications (the functional and design requirements and design criteria). Functional and design requirements have already been formulated with the PROPER model. The design criteria are described by Pugh [1990] and they consist of a product design

specification (PDS), the organisation and the personnel. The PROPER model is an appropriate way to reflect these criteria.

2. *Conceptual design* is the decision making to obtain a concept. Doepker calls it a very important step, because design changes are still "cheap" to implement. He already refers here to the changes of one single design only.
3. *Preliminary or detailed design*. Pahl and Beitz [1996] call it "embodiment design"; layout and form are being quantified during this step. Materials and geometry are defined.
4. *Final design*. Detailed analyses (stress analysis, shaft design, heat transfer etc.) are performed in this step.
5. *Implementation.*

During the steps 1 and 2, specific requirements are added by the technological discipline for example concerning materials, safety, size, weight, ergonomics etc. The steps 3 and 4 detail the design, eventually resulting in extra restrictions. These restrictions are given feedback to the other disciplines with reference to the commonly defined functionality in the PROPER model. The more detailed insight with respect to "behavior" is added to the model.

Example 3.2. illustrates the combination of product flow and resource flow from the PROPER model in order to achieve a complete functional specification of requirements for the technical resources. They form the basis for confrontation and tuning and the starting point of technical design.

Example 3.2. The import process of containers at a deep sea terminal covers the transfer process between ship and stack area. For the resources to be used it is decided to use quay cranes to unload, automated vehicles for the transport function and stacking cranes to stack the containers.
Transfer functions are required to transfer containers between the resources.

Suppose this product flow for this part of the Operate function is modeled as in figure 3.5. (from [Veeke & Ottjes, 1999]).
One of the alternatives to be investigated during confront and tune step of function design is the one where the automated vehicles don't have a lifting installation; the transfer functions are to be performed by the other equipment.

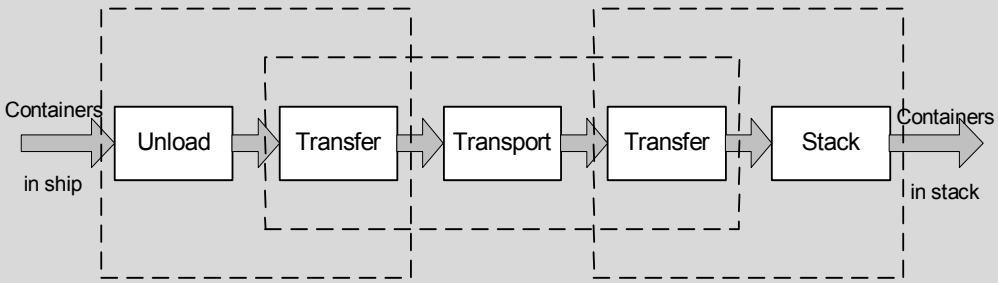


Figure 3.5: *Functions in the container import process*

By adding the resource flow to the product flow, figure 3.6 results.

Now the technologists are asked to provisionally work out the vehicle design. The required functions are defined by isolating the vehicle part from figure 3.6 and zoom in on the assignment part. For illustration purposes all control functions are not shown.

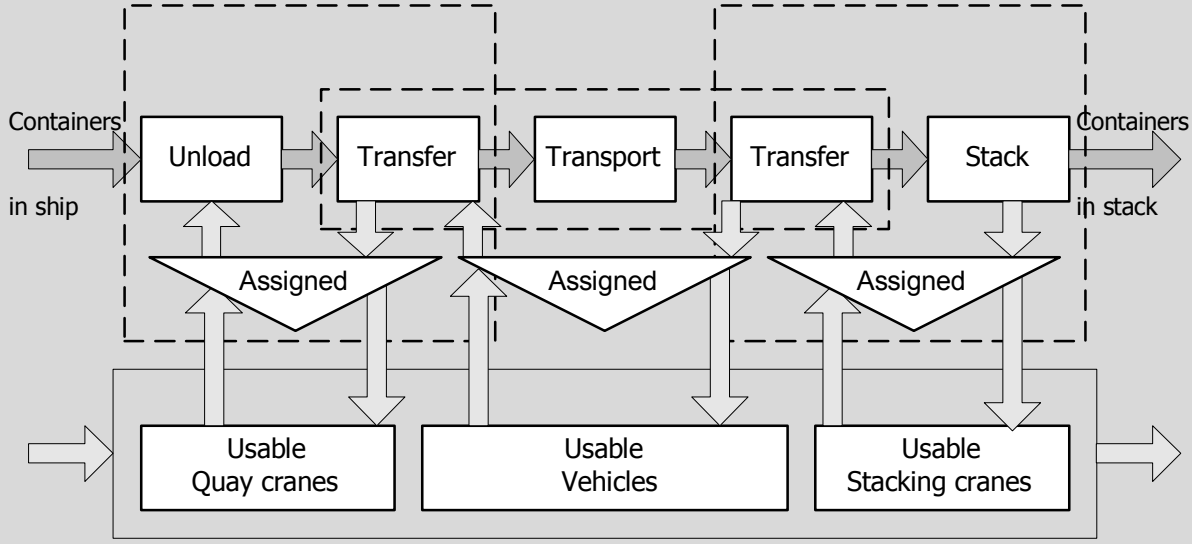


Figure 3.6: *Product and resource flows in the container import process*

From the technological viewpoint, the size, speed etc. of the vehicles are major design criteria. Vehicles always have a physical position, use space and they should be "stored" in all cases where waiting for a job or a transfer by quay crane and stacking crane is encountered. This becomes clear by zooming in to the buffer of assigned vehicles (figure 3.7).

The Receive, Wait for SC and Deliver functions are parts of the transfer functions of figure 3.6. The results and efforts to be expected with this type of vehicles in this configuration are

to be estimated from these functions. Expected results will be the number of transports per time unit, while efforts will be the number of vehicles, their occupation and space requirements.

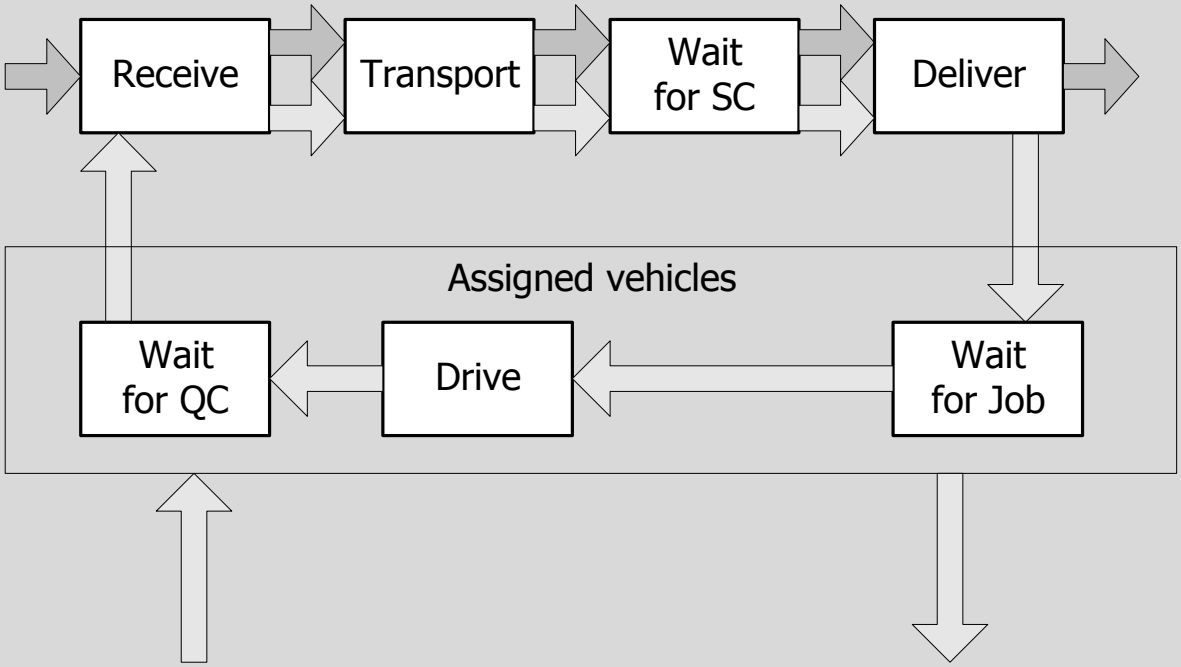


Figure 3.7: *Vehicle functions in the container import process (QC = quay crane, SC = stacking crane)*

The Wait functions can be given feedback to the overall design to take into account for comparison with other alternatives.
The pure technological functions 'Drive' and 'Transport' will be worked out after the alternative is selected. In this case the results and efforts to be expected are accepted and have become standards for the technological design.

3.4.3. The design of organization systems

Bikker [1995] defines a strategy of organization design (figure 3.8). He distinguishes two major pathways for the design:

1. the analysis of objectives and policy
2. the analysis of the organization of processes.

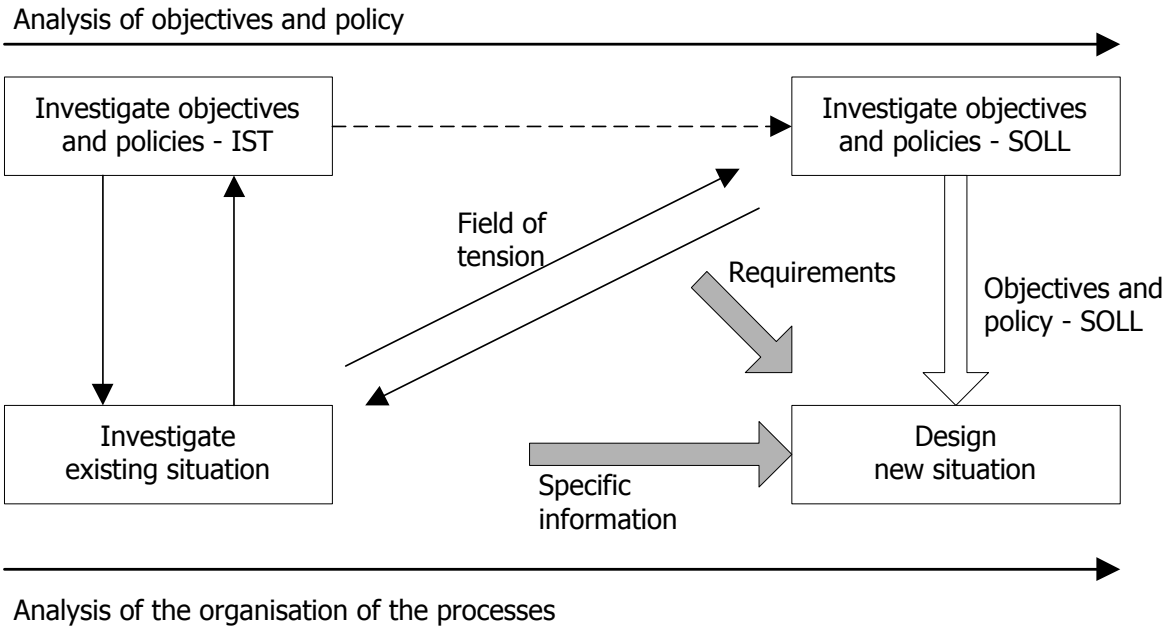


Figure 3.8: *Strategy of organization design (Bikker [1995])*

The first path, analysis of objectives and policy, fully coincides with the function design of the design process in this thesis. The use of the PROPER model fits very well to investigate the existing situation and to design a new situation, which is the second path. Above that the model is able to visualize policy decision problems in a compact and conveniently arranged way. Example 3.3 illustrates this.

Example 3.3. The Carrier refrigeration company considered a major reorganization to facilitate the combination of different sales companies operating on a partly overlapping market. The character (culture) of the companies varied from purely sales oriented to heavily installation oriented. Products range from simple refrigeration equipment to complex installations. The installations are maintained by the company during their life time according to a service contract with the customer.

A PROPER model was created at the highest level of abstraction, visualizing the order and product flow (figure 3.9).
There are four main functions distinguished. A sales function to sell new installations and to maintain customer relations, a service function to respond in time to disturbances, an installation function to perform the installation process and a maintenance function to maintain installations.

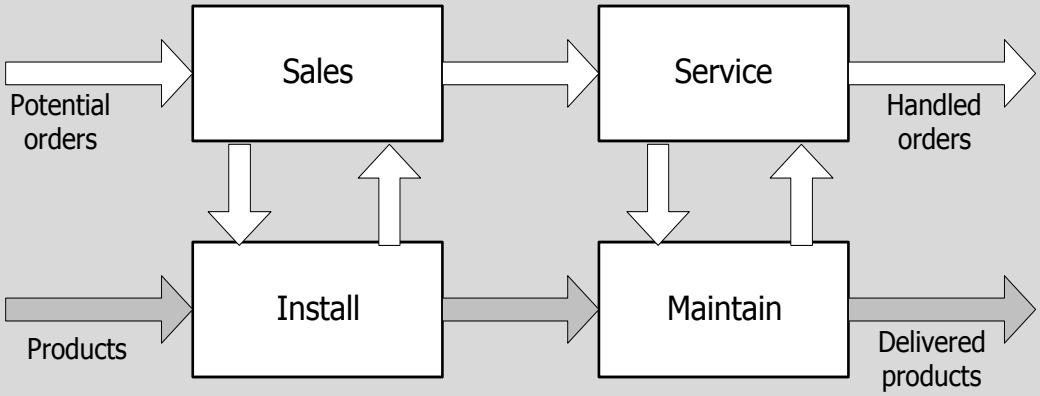


Figure 3.9: *Main functions of Carrier refrigeration company*

This model was used to illustrate and discuss two principal organizational alternatives.

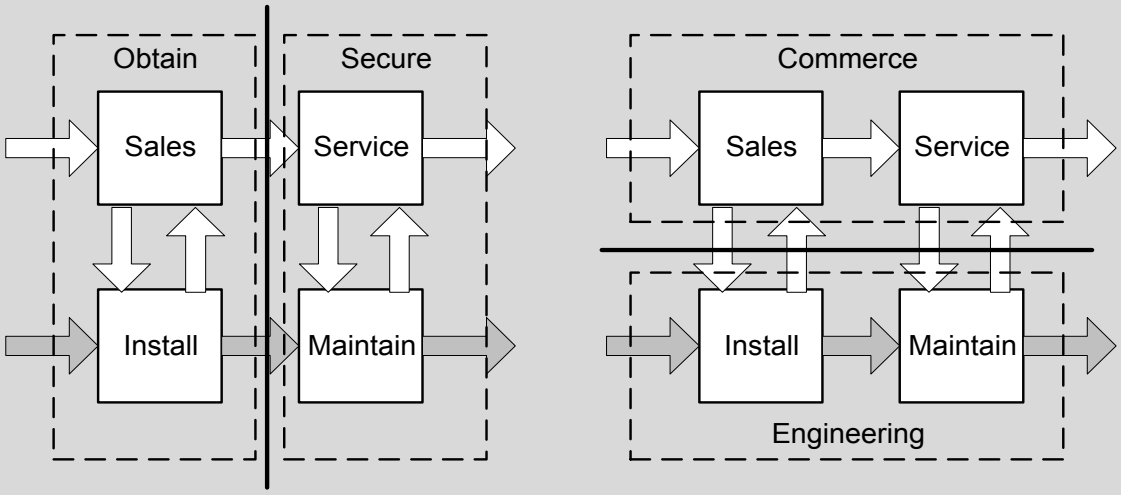


Figure 3.10: *Alternatives for the organizational structure*

Each alternative has consequences concerning the rest of the design process, the personnel and the flexibility with respect to in- and outsourcing. Especially the interfaces between obtain / secure and commerce / engineering were a major item of discussion.

The PROPER model represents a conceptual organization design as a starting point for process design. In process design the consequences of function particularization are investigated, first to compare different alternatives and later to detail the selected alternative. Not only departments should be decided on (the organ structure), but also on the competencies and the responsibilities (the personnel structure). They all determine the resources required.

3.4.4. The design of information systems

Burback [1998] distinguishes four fundamental phases that occur in all software engineering methodologies: analysis, design, implementation and testing. These phases address what is to be built, how it will be built, building it and making it of high quality. They all not only apply to the product delivery, but also to deployment, operations, management, legacy and the discontinuation of an information system.
The analysis phase defines the requirements of the system, independent of the way in which these requirements will be accomplished. The PROPER model can be used for this purpose. If one replaces the "order, product and resource" flows by "information on order, product and resource" flows, the model completely reflects data flows. In addition, the use of the PROPER model prevents the use of different element types during requirement analysis. "Documents" will be assigned to data flows (instead of functions), databases will usually be positioned at stock / buffer locations and within evaluation functions. One should be aware however of the fact, that the PROPER model is concerned with operational data only. Data used for innovation and modeling purposes are not contained in this single design approach.
The next design phase concerns the structuring of the information system: the architecture. It defines components, their interfaces and behaviors. They can be directly derived from the PROPER structure. In the implementation phase components are built either from scratch or by composition. Finally the testing phase assures the correct quality level of the system.
Currently most methodologies provide an object-oriented development in the analysis and design phase. According to Lodewijks [1991] object orientation increases the level of abstraction of programming languages and enables a better fit of the design process to the human perception. He shows a direct projection of a functional design of a physical process or system to object oriented programming [Lodewijks, 1996]. An extended object oriented analysis and design for the purpose of simulation modeling, based on the PROPER model, will be shown in chapter 5.

3.5. Simulation for the support of the design of logistic systems

In the last paragraphs, simulation has been mentioned several times as a supporting tool for decision making during the design process. In both function design and process design, simulation can be applied in conjunction with the PROPER model for "quantifying the structure". A logistic system is a complex system consisting of a large number of elements with multiple interrelations and often a stochastic behavior. It is these situations that a static

structure (such as the PROPER model) is not sufficient to decide for the most appropriate design of a system. Dimensions should be taken into account also and control algorithms must be tested thoroughly. The complexity of the system makes a mathematical description impossible. Simulation however is capable of describing the time dependent behavior of these systems and performing experiments as if the system is (virtually) operational. Simulation can even provide increased insight in seemingly simple situations, especially in case of stochastic phenomena. Global models during strategic decision making are often considered "simple" situations, but decisions in this phase can not be easily changed in later steps of the design process. Besides that the results of global simulation experiments provide the necessary data to verify complex simulation models.

Example 3.4. Veeke[1983] showed the use of simulation in a rather simple system, which was calculated manually (but systematically) by in 't Veld (see [in 't Veld, 2002] for the original case) . The system concerns a testing department for new airplanes delivered by a production line at a constant rate of 1 airplane each 1.5 day. The customer also expected this supply rate. Test flights are made with each airplane, and when a test flight shows defects, they are repaired in a covered shed. Repairing defects after a test flight takes 3 days on average, ranging from 1 to 8 days. If a test flight shows no defects, the airplane is delivered to the customer. Airplanes require an average of 10 test flights (distributed between 4 and 16); the last test flight shows no defects anymore. A test flight is assumed to take 1 day. The dimensions of the shed had to be determined and in 't Veld used the systems approach to calculate the average number of positions required in the shed. Using Little's formula [Little, 1961] :

$$N = \lambda * D \qquad (3.1)$$

where N = average number of elements in the systems
λ = arrival rate of elements
D = average throughput time

the average number of positions can easily be calculated. The average air plane passes the shed 9 times with each time an average of 3 days. This leads to a total average throughput time of 27 days. The arrival rate λ equals 2/3, so on average 2/3 * 27 = 18 positions will be required.

This number of positions was actually provided, but this passes over the addition "average".

By means of a simulation model, it was shown that the number of occupied positions ranges from 10 to 28. The availability of 28 positions is required to realize a theoretical average number of 18 occupied positions. With the simulation model, it was further analyzed that the availability of 20 positions would be sufficient to near the required output of 1 airplane each 1.5 working day.
Another conclusion can be drawn; the simulation model used an infinite capacity approach to determine the number of positions required. If at a later stage another number of positions is decided on, then it is immediately clear what the utilization of the positions will be. For example, if 20 positions would have been selected, then the occupation would be 18 / 20 = 95%. This offers the opportunity to verify more complicated models.
In reality this case took place in the 1960's and at that time, it was impossible to create a computer simulation for it. If computer simulation would have been used, the system would have been equipped differently.

From the examples above, it can be concluded that simulation is able to support decision making from the very first global modeling steps until the final detailed steps.

The PROPER model has been defined and the use of it during function and process design is illustrated. The time dependent behavior is not included in this model; an extension to this model is required to connect the interdisciplinary modeling so far to the "world of simulation".

Chapter 4

Cognitive mapping of function to process

"We are thus able to distinguish thinking as the function which is to a large extent linguistic."
- Benjamin Lee Whorf -

4.1. Introduction

In the previous chapters, the PROcess-PERformance model (PROPER) has been developed and it was shown that it serves as a common reference model for the disciplines involved. It thereby supports the designers to deal with complexity and to discuss a system design. The schematic representation by means of the PROPER model however is static, showing no time dependency at all. Besides an analysis of the static structure, there is a need to study the time dependent behavior of the system design. This behavior effects the required dimensions of the system and determines its theoretical, standard and real effectiveness and productivity. Therefore, the time dependent behavior requires a representation that can be understood and used by all disciplines involved and that is a natural extension to the PROPER model itself. In this chapter such a representation will be defined.

Currently it is common use to apply simulation models to study the time dependent behavior of systems. As with the other disciplines, simulation evolved in a monodisciplinary (i.e. predominantly mathematical) culture. In the world of management and technology, it is considered a job of specialists, and as a result design problems are usually "thrown over the wall" to the simulation experts. These experts are expected to understand the specialist requirements, to model the system correctly and to produce correct results regarding the problem formulation. In this situation, the quality of the simulation solely depends on the quality of problem interpretation by the simulation expert. The simulation expert, however, is not supposed to be an organization or technological expert. As shown in chapter 2 with the systems approach, a "system" is a subjective perception and therefore a conflict can arise between the problem owner's perception and the simulation expert's perception.

In this thesis, simulation is considered an approach to represent system behavior in a software environment. To represent behavior correctly, it must first be described in terms

that can be understood by the problem owners themselves. Secondly the description should be made unambiguous for the simulation expert.

The description of behavior of a system under study will be derived from the PROPER model. It will be based on natural language, because natural language facilitates the communication and discussion between the different disciplines. Galatescu [2002] states that natural language has the unifying abilities, natural extensibility and logic, which are unique properties to enable conceptual model integration. Descriptions of behavior can be expressed with natural language. In this way the specification of behavior becomes the responsibility of the problem owner again instead of the solution provider (i.e. the simulation expert). The natural extensibility of natural language also supports the use of conceptual models during the design process.

In this chapter, the use of natural language for describing behavior will be examined. When a problem owner explains the way in which a system "works" using natural language, he/she already describes behavior globally. The basic idea in this thesis is to structure these descriptions in such a way that they can be mapped to the processes of functions in the PROPER model and still can be recognized and understood by the problem owners. Otherwise stated, the behavior of a function is mapped to a "process description" in a cognitive way. The PROPER model is thereby extended with the time dependent behavior of the system, preserving the interdisciplinary character. Process descriptions can be defined at each aggregation stratum.
Subsequently a process description is communicated and discussed with the simulation expert and it is the basis for an unambiguous translation to a software environment, which results in a simulation specific description.

4.2. Behavior

Behavior is a notion, which is used by many disciplines with many different meanings. Therefore, in this thesis, a basic definition of behavior is used as the starting point.
At the very start of cybernetics, RosenBlueth, Wiener and Bigelow [1943] defined 'behavior' as "any change of an entity with respect to its surroundings". They declare this behavioristic approach as the examination of its output and the relation of this output to the input. They contrast this to the 'fundamental approach', in which the structure, properties, and intrinsic

organization of the object are studied, rather than the relationship between the object and its environment. This fundamental approach agrees to the use so far of the (static) PROPER model, which supports the design of structure, property and organization. As a result of this model, desired outputs are defined, and apparently studying behavior is needed to derive from this the necessary inputs and to verify the feasibility of the desired results.
The 'entity' in the definition of Rosenblueth et al. above is interpreted to be a function[1] in the PROPER model. Having this, the term 'any change of' must be made clear with respect to that 'function'.

To achieve this, the definition of In 't Veld [2002] of 'any change' is considered. He defines the behavior of a system as the way, in which the system reacts to internal and external circumstances, to input and changes in input. He relates behavior with the 'state' of the system. "The state at some point in time of a system with a given structure, is a set of values that together with the input signal at that point in time unambiguously determine the output signal". He considers the set of values the result of input signals from the past. A value must belong to some kind of a 'property' of the system.

The term "unambiguously" complicates the interpretation of this definition. Apparently the state of the system is not fully described by the values of its internal properties but must also include the values of external properties, which may influence the internal properties (by means of input signals). The conclusion is, that the state of a system must reflect both the internal and external circumstances. Consequently, there are two sources by which a property value may change: the system itself or an external source (another system or the environment).

The change of an entity as mentioned by RosenBlueth et al. can now be expressed as a change of the 'state of the function'. A change of the state is therefore a consequence of a change in the set of property values or a change in the input signals. The functions in the PROPER model react to these changes by delivering (wanted or unwanted) output signals.

In this thesis, behavior is now defined as:

[1] In this chapter the terms 'system' and 'function' are used interchangeably. The elements of a system are functions, the system as a whole is also a function.

"behavior is the property of a function that describes the way in which the state of the function together with its input result in output."

According to this definition, a function has two major properties: a state and a behavior. The behavior property cannot be expressed by a simple value, but must somehow express the way in which the state will change.

First the state of a function in a logistic system and its input will now be defined. Then a method will be defined to describe behavior, showing how the state and input together result in output.

4.3. The state and input of a logistic function

Input signals of a logistic function are derived from the PROPER model. Three types can be distinguished :

- Physical signals: the order, product and resource flows. These flows might be the output of preceding functions (internal environment) or enter the function directly from the external environment .
- Control signals: generated by the control part of a function to the operational part of the function (interventions) or entering the function from higher echelons (needs). The needs are translated to standards for the operational function.
- Disturbance signals: are part of the input to the control function (the feed forward and feed back control). They originate from the inside of the operation part (e.g. machine disturbance) or from the environment of the function (e.g. weather conditions) and influence the progress directly.

If the input signals are represented in the basic logistic function of figure 2.8, skipping all output signals, the resulting picture is figure 4.1.

Using the concept of effectiveness and productivity, performance indicators are assigned to the processes dealing with these input signals; the indicators are expressed in terms of results and efforts. Results are the output of a logistic function, while efforts are required to achieve the results. Efforts must be made to provide space, resources, products and time for operation and control. In general: results are the consequence of making efforts to

transform input into output. There are many ways in which efforts can be made to achieve the same results. They heavily depend on the function structure of the logistic system, but also on the relations between the aspects inside the system. Resource assignment and maintenance together with task specification and scheduling influence the efforts being made. By describing the way in which efforts are made one is in fact describing the relation between input and results: the behavior.

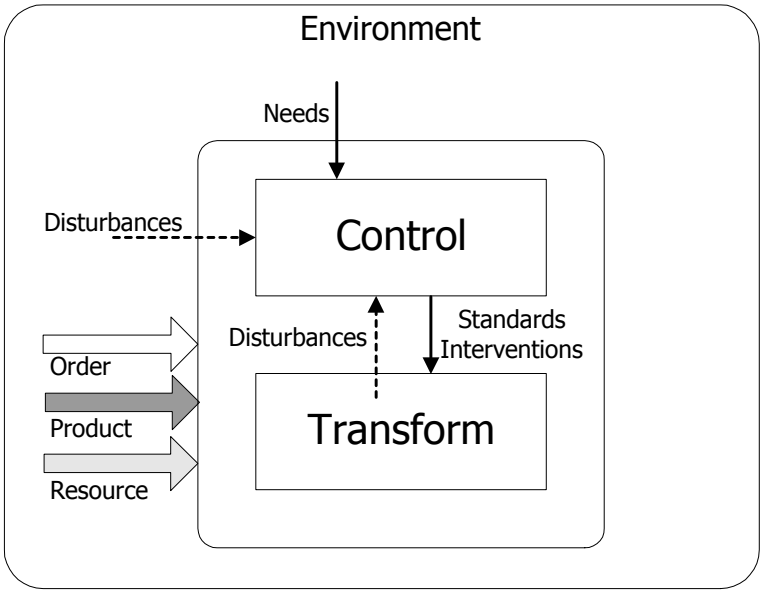


Figure 4.1: *Input signals of a logistic function*

Operations Research has developed many algorithmic methods for cases that can be formulated in a mathematical way and often 'time' is no part in these relations. A logistic system however, is a complex system with many relations, unexpected situations and each operational transformation takes (or "costs") time; for this type of problems, simulation has proven to be a suitable approach.

Efforts represent the number and duration of resource assignments, the use and number of products and resources, and finally the efforts of control. Not only the number of effort requiring elements but also the duration of use is important.
The state of a function is defined as a property at some moment in time, and therefore duration must be translated to a value expressing it at that particular moment in time. According to in 't Veld the state is the result of input signals and the behavior from the past, so the duration to be expressed at any moment is the remaining duration. It becomes an expression for the moment in time at which one (or more) value(s) in the set of values will change.

Example 4.1. Suppose a function is 'busy' with a task that takes a total of 3 hours. After one hour of work, a part of the state description of the function will be: "function is busy for the next two hours".

The example shows that the state of a function not only contains values of static "observable" properties (e.g. busy / idle), but also the period of time during which these properties will keep their current value (e.g. busy for 2 hours). The properties are derived from the static structure of the PROPER model, the time periods are derived from the properties of the input signals (e.g. execution times of tasks, working schedules of resources etc.). These periods must be reflected in the behavior property.

The PROPER model represents a function structure where each function consists of an operational and control function. The steady state model of in 't Veld (see appendix A) shows all subfunctions in both functions. The model offers a structural way to make an inventory of which functions may be necessary to achieve the desired results. Consequently, defining the properties of each of these functions gives an overview of all internal properties of the overall function. Analogous to the use of the steady state model not all properties are required (either because the subfunction is not required or the property itself is not required for the modeling purposes); therefore the required properties will be denoted by "significant" properties.

With respect to the behavior reflecting the periods mentioned, the influence of the environment must be taken into account. The regular part of environmental influence is formed by the input flows; above that, the function is influenced by the environment in other ways. One may think of weather conditions or unpredictable phenomena that influence the progress of elements through the operational function e.g. the moment when a machine disturbance will occur. For this reason the period will be denoted as an "expected" period.

Where the definition of in 't Veld in par. 4.2. defines the state in relation to input and output (it is in fact an 'outside view'), a definition is added in this thesis, which describes how the state is composed (an 'inside view'):

"The state at some moment in time of a logistic function with a given structure is a set of significant properties of the function each having a value and an expected period of time during which each value holds."

These "expected periods" will constitute the behavior property of a logistic function.

4.4. The behavior of a logistic function

The behavior of a logistic function can now be composed in two complementary ways (see figure 4.2)

1. as a set of moments in time (also called "events") where the state of the function changes.
2. As a set of periods during which the state of the function remains unchanged.

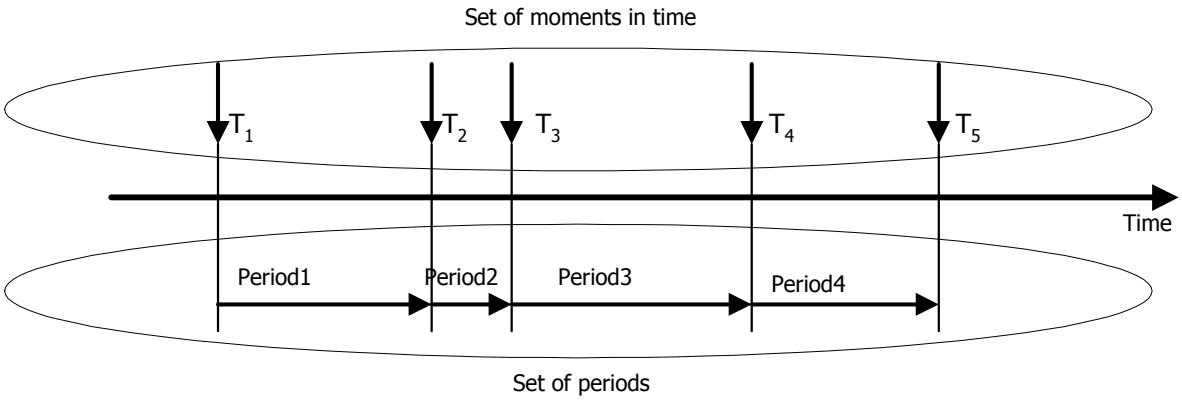


Figure 4.2: *Approaches to behavior descriptions*

Both moments in time and periods are determined by the function itself, by other functions and/or by the external environment. If they are determined by the function itself, the term 'internal' will be used in this thesis.

To decide on which representation of behavior is most appropriate, the purpose and use of the behavior description (as an extension of the PROPER model) are recapitulated first.

Firstly, in the function and process design the PROPER model is used as a common reference model for all disciplines involved and thereby supports communication between the disciplines. For this purpose the PROPER model provides a coherent frame of reference that can be used in an interdisciplinary way. It is however a static picture of the system; communication and decision making on behavior should be supported as well. In paragraph

4.3. behavior and state are added to this concept for the time dependent description. This description should also be interdisciplinary.

Secondly it must be easy to structure and detail the behavior description; during the design process functions are structured in different ways and zooming in will add details to the descriptions (see chapter 3).

Finally it must be possible to relate descriptions at different levels of detail (aggregation strata) to each other. Decision making proceeds during the design process and zooming in to one specific function should not be isolated from preceding decisions.

Therefore, the major requirements for the behavior description are: clarity, structure flexibility, ability to detail and hierarchy.

Behavior is a complex property and it is better represented by a linguistic description rather than a value. The description explains when a state change occurs and how the function reacts to a state change. The latter part is by definition a part of the behavior description of the function itself. A state change can be caused by an internal or external cause. Therefore the behavior of a function is divided into an internal part and an external part. The internal part contains the way in which the function reacts to a state change and the way in which the function itself causes state changes (for itself or for other functions). *This internal part of the behavior is now defined as the "process" of the function.*
The external part is represented in the behavior of other functions. Therefore, the state of a function is not always completely established by its own set of values (including the process), but might be connected with the states of other functions and the environment. This conclusion is completely in line with the principles of the systems approach: ”an element cannot be studied in isolation from its environment”.

Returning to the two possible ways of describing behavior, example 4.2. is used to illustrate both and to formulate the arguments for the approach to be preferred.

Example 4.2. An operational function transforms semi-finished products from an inventory into finished products. It is assumed the function is allowed to operate without tasks and only one resource is permanently assigned to the operation. So the operation can focus on products only and the order and resource flow can be skipped.

Products are selected from an inventory in a first-in-first-out order; each product transformation takes a certain execution time. If there are no products available the function waits for a new product to arrive in the inventory.
This is already a first process description completely in natural language. To structure this description, three moments in time are distinguished where the state of the function changes:

1. a product arrives in the inventory
2. the function starts a transformation
3. the function finishes a transformation.

For the transformation there are two periods:

1. a period during which the transformation 'works' on a product
2. a period during which the transformation 'waits' for a product to arrive in the inventory.

These periods cannot overlap by definition, so they can be described as one sequence of periods and must proceed in the order described. Another period must be defined for the environment of the function:

3. a period between two successive arrivals of a product.

Figure 4.3. shows the moments in time and periods mapped to one time axis.

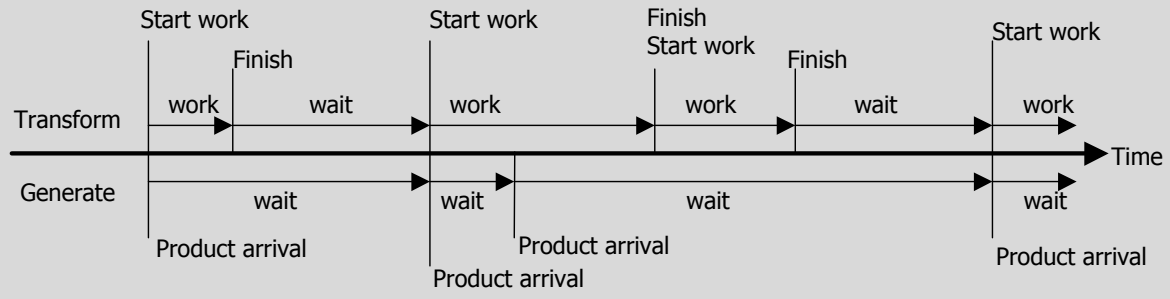


Figure 4.3: *Moments in time and periods for an operational function*

The terms 'work' and 'wait' describe the type of "activity" during a period. The generator's

periods are apparently all waiting activities for the next product arrival.
As shown is figure 4.3. moments in time may coincide to the same instant. There are two sequential period descriptions required: one for the transformation and one for the generation of products. The next table shows both ways of process descriptions using natural language. To interpret these descriptions the reader should always be aware of the clock time of the system (which is used by all process descriptions). The current clock time is denoted by "Now".

Processes with moments in time	***Processes with periods***
Moment_1 of Operation *If inventory is not empty* *Schedule Moment_2 Now* *Moment_2 of Operation* *Select first product from inventory* *Schedule Moment_1* *after product's execution time* *Moment_1 of Generate_products* *Put new product in Operation's inventory* *If Operation is idle* *Schedule Moment_2 of Operation Now* *Schedule Moment_1 at next arrival* *To start the processes:* *Schedule Moment_1 of* *Operation Now* *Schedule Moment_1 of* *Generate_products Now*	*Periods of Operation* *Repeat* *Wait while inventory is empty* *Select first product from inventory* *Work product's execution time* *Periods of Generate_products* *Repeat* *Put new product in Operation's inventory* *Wait for next arrival* *To start the processes:* *Start Periods of* *Operation Now* *Start Periods of* *Generate_products Now*

Table 4.1. Two ways of process descriptions

Each Moment in Time and each Process has to belong to the behavior of some function Operation or Generate_Products. This "owner" function is always explicitly mentioned. The same holds for properties. Therefore the property 'inventory' is denoted by Operation's inventory. The owner is not mentioned explicitly when the owner refers to a property of its own. The term 'Select' means here 'select and remove from inventory'. Once a period

description is started it runs in a repetition of succeeding periods. This repetition is denoted by "Repeat".

To illustrate the interpretation, the first steps of the processing sequence will be explained for both approaches.

Using Moments in time:

The clock time is set to zero, so Now = 0. First of all two moments are "scheduled" to start the processes, one for Operation and one for Generate_products. Scheduling means both moments are marked on the clock, at position "Now". Moment_1 of Operation is marked first, so the first line to be interpreted is the one of this Moment description. There the Operation checks the inventory. The inventory is still empty, so Operation does not schedule Moment_2 and becomes "idle"; now the next mark on the clock can be handled. This next mark is Moment_1 of Generate_products. A new product is put in the inventory of Operation. Having done this, Operation must be notified of the value change of one of its significant properties (it is a state change of Operation). However, this value change is only significant if Operation is "idle". If it's already handling a product, then Operation must finish this product first before it can react to this state change. This first time a product is added to the inventory, Operation is idle indeed, so Moment_2 of Operation can be scheduled at this very moment ("Now"). Finally Generate_products marks its own Moment_1 on the clock to be notified at the moment a next product arrives. This moment lies somewhere in future. Now the actions of Moment_1 are finished and the next mark on the clock can be executed; this mark is Moment_2 of Operation etc.

Using periods:

The clock time is set to zero, so Now = 0. First two period descriptions are "started", one for Operation and one for Generate_products. Both descriptions are marked on the clock at position "Now". The period description of Operation is marked first, so the first line to be interpreted is the one of this description. It reads "Repeat", and this tells the reader to return to this point when the indented lines are passed. The next line says that Operation will wait as long as the inventory is empty. Just doing what this line says, the condition is "kept in mind" and no mark is added to the clock for Operation. The description cannot be continued, because the inventory is empty, so returning to the clock, the description of Generate_products is entered being the

first next mark on the clock. After the repeat line, a new product is put in Operation's inventory. This changes a condition kept in mind (Operation is watching the inventory), but the description proceeds with 'wait for next arrival', which marks this period description somewhere in future and will return then to the following line (which is Repeat). Looking at the clock now, a condition is 'pending' and one clock mark in future is available. Before the clock advances to the next mark, the condition is checked. Because there is a product available, the reader now returns to the line in the description of Operation following this condition check. The first product is selected and execution starts. This results in another clock mark in future. At that moment the clock time can be advanced etc.

The next general conclusions apply (moment in time is denoted by MIT) :

1. when using MIT's, there may be descriptions where apparently no change of state occurs. In the example nothing seems to happen at Moment_1 of Operation when the inventory is empty. One could argue that the operational state of the function changes at that MIT from busy to idle. Although true this is at least not clear from the description itself and introduces ambiguity. The period approach explicitly states the function is 'waiting' in this case. Periods always express a kind of 'activity' or 'passivity'.
2. The order in which MIT's are specified does not influence the course of the process, if the first MIT remains the same. One could exchange the order of Operation's Moment_1 and Moment_2; if the operation still starts with Moment_1 the course of the process stays the same. However, the course must be explicitly mentioned by using 'Schedule' expressions. When using periods, there's no need for this, because the description order of the periods determines the course of the process. It agrees to the way of reading natural language: from top to bottom.
3. In the description of MIT's, a next MIT is created by specifying when it occurs. If no next MIT can be specified (as is the case in Moment_1 of Operation if the inventory is empty), it is impossible to 'delay' the current MIT itself until the next MIT can be specified. For this reason Generate_products must reschedule Moment_1. The description using periods is able to delay MIT's by extending the current period e.g. by specifying conditional finish times ('wait while inventory is empty').
4. The sentence 'Select first product from inventory' can be added to the description of Moment_1 and to the description of Moment_2. There is no reason why it should

exclusively be assigned to Moment_2 (because they are the same moments if a product is available); the only condition is that the product must be selected before the function starts executing. This sequence condition is automatically fulfilled by using period descriptions.

5. In the example the MIT's are named Moment_1 and Moment_2 of Operation. In complex situations it is advisable to name the MIT's in such a way that they express what actually happens at that moment. The best expression for Moment_1 of Operation would be "the function eventually starts waiting for a product". The period description actually uses this expression ("wait while inventory is empty").

With respect to the first requirement of the descriptions – they must be clear - all conclusions show that the period approach is to be preferred so far. Each function also has one and only one description of periods. From now on a period description will be called a process description.

The example is now extended to include tasks and resources. Then according the PROPER model the behavior of both a perform function and a use function must be described.

Example 4.3. The operation function of example 4.2 now requires both a task and a product to start the transformation. Above that the resource must be shared with other operations, so the resource must be explicitly assigned to this operation. It is assumed each order consists of one task only. A task is added to a task list of the operation and selected in a FIFO-order.
The processes are shown in the table 4.2.

Processes with moments in time	***Processes with periods***
Moment_1 of Operation *If task list is not empty* *Schedule Moment_2 Now* *Moment_2 of Operation* *If inventory is not empty* *Schedule Moment_3 Now* *Moment_3 of Operation* *Enter request list of Use*	*Process of Operation* *Repeat* *Wait while task list is empty* *Wait while inventory is empty* *Enter request list of Use* *Wait until resource assigned by Use* *Select first task from task list* *Select first product from inventory* *Work product's execution time*

```
If Use is idle
    Schedule Moment_2 of Use Now

Moment_4 of Operation
Select first task from task list
Select first product from inventory
Schedule Moment_5
    after product's execution time

Moment_5 of Operation
Remove resource from busy list of Use
Put resource in idles list of Use
If Use is idle
    Schedule Moment_2 of Use Now
Schedule Moment_1 Now

Moment_1 of Generate products
Put new product in inventory of Operation
If Operation is idle
 And Operation Not in request List of Use
    Schedule Moment_1 of Operation Now
Schedule Moment_1  at next arrival

Moment_1 of Perform
If order list is not empty
    Schedule Moment_2 Now

Moment_2  of Perform
Select first order from order list
Put new task in task list of Operation
If Operation is idle
 And Operation Not in request List of Use
Schedule Moment_1 of Operation Now

Moment_1 of Generate orders
Put new order in order list of Perform
If Perform is idle
    Schedule Moment_1 of Perform Now
Schedule Moment_1 at next arrival
```

```
    Remove resource from busy list of Use
    Put resource in idles list of Use

Process of Generate_products
Repeat
    Put new product in inventory of Operation
    Wait for next arrival

Process of Perform
Repeat
    Wait while order list is empty
    Select first order from order list
    Put new task in task list of Operation

Process of Generate orders
Repeat
    Put new order in order list of Perform
    Wait for next arrival
```

Moment_1 of Use *Put new resource in idles list* *Schedule Moment_2 Now* *Moment_2 of Use* *If request list is not empty* *Schedule Moment_3 now* *Moment_3 of Use* *If idles list is not empty* *Schedule Moment_4 now* *Moment_4 of Use* *Select first resource from idles list* *Select first operation from request list* *Schedule Moment_4 of Operation Now* *Put resource in busy list*	*Process of Use* *Put new resources in idles list* *Repeat* *Wait while request list is empty* *Wait while idles list is empty* *Select first resource from idles* *Select first operation from request list* *Assign resource to Operation* *Put resource in busy list*

Table 4.2. Two ways of process descriptions (continued)

Looking at the descriptions of table 4.2, the following general conclusions are drawn:

1. A number of conditions can be checked sequentially and be fulfilled at different moments when using period descriptions. Using MIT's sequential stepwise checking introduces a MIT for each condition.
2. By using period descriptions, the sequence of actions and conditions becomes and stays clear. Interpreting a period description puts the reader in the position of the transformation itself. This enriches the problem formulation of the system. For example here tasks and products are selected at the moment a resource is assigned (knowing at least one task and one product are available). What happens if task and/or product data are needed for the assignment of the resource? Questions like these can easily pop up in confronting each discipline with period descriptions and by this, decision making is supported during the design process.
3. Extending the number of functions leads to extending the description with one period description for each function (plus one generator eventually). The number of MIT's however increases with more than one MIT for each function. In complex systems

the number of MIT's explodes and reasoning about the behavior becomes a complex task itself. Kirkwood [2002] states (using "event" for MIT): "Many people try to explain business performance by showing how one set of events causes another or, when they study a problem in depth, by showing how a particular set of events is part of a longer term pattern of behavior. The difficulty with this "events causes events" orientation is that it doesn't lead to very powerful ways to alter the undesirable performance. This is because you can always find yet another event that caused the one that you thought was the cause".

4. With period descriptions, each function is able to express that it is continuously watching for a condition to become true by means of while - and until – clauses. With MIT's the condition can be checked only once; other functions must take care of forcing a check again.
5. If in the MIT description names don't express the action then the description is difficult to interpret. One must be careful in what state the function is. For example at Moment_1 of Generate_products and Moment_2 of Perform one cannot simply test for 'Operation is idle' because the operation can be idle for more than one reason.
6. Combining or splitting functions is straightforward using period descriptions. Suppose the perform function and operation function must be combined into one organizational unit. In this case the description of operation can easily be adapted by replacing the first line of its behavior by the three lines of the perform function. It's not straightforward using MIT's. Moment_1 and Moment_2 of Perform cannot simply replace Moment_1 of Operation, because Moment_1 is referred to by other MIT's and all condition checks must be reviewed.

From the examples and the above given reasons it is concluded, that compared to MIT-descriptions the period descriptions are superior to describe the behavior of functions. They are clear and to some extent unambiguous. The item of ambiguity will be addressed further in chapter 5. Period descriptions can also easily be detailed and are flexible in dealing with different structures. For these reasons only period descriptions will be elaborated further.

4.5. Basic concepts of process descriptions

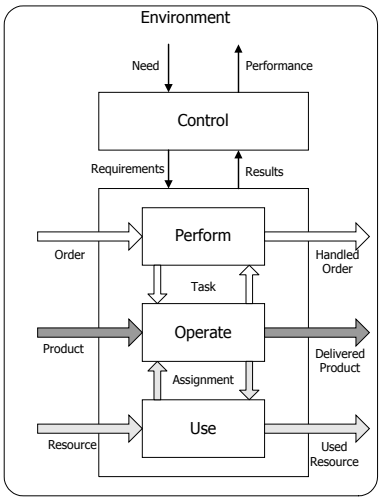
In paragraph 4.3. the state of a logistic function was defined as a set of significant properties, each having a value and an expected period during which the value holds. To describe these periods, the process description was derived in paragraph 4.4. In this

paragraph, the way in which the properties themselves appear in a process description are investigated and general concepts will be derived. For illustration purposes, the process descriptions of table 4.2. will be used. Then the way in which the time duration of periods can be described will be defined, which will lead to a property describing the state of the process of a function. The PROPER model is used during the design process evolving from a global model to a network of detailed models. This process of "aggregation" or hierarchy and its effects on process descriptions will be investigated in paragraph 4.5.3.

4.5.1. Properties

The types of properties of a logistic function are derived from the PROPER model of fig. 2.9. The first distinction is:

1. Properties of the horizontal flows: order, product and resource (flowing through perform, operate and use)
2. Properties of the assignment and task flows between aspects (between use and operate and between perform and operate)
3. Properties of the control and transform function themselves
4. Properties of vertical flows (flowing through control)



4.5.1.1. Properties of horizontal flows

Horizontal flows represent input elements that will be transformed to desired resulting elements.
Properties of these flows therefore consist of two different types again:

- properties describing the state of the elements in a flow.
- properties describing the state of the flow itself.

Significant properties of the elements are the ones that are changed or used by the transformation and the ones that describe the "position" of the element in the transformation: is it on the input side, being transformed or has it been transformed?
In fact, in this way the flow itself is divided in three interconnected parts: an input flow, a throughput flow and an output flow. The flow properties therefore, reflect collections or sets of elements in each part of the flow. This "set" notion will appear to be in the next pages a generic concept. To describe the dynamic behavior of these sets the flow's input rate must be defined representing the number of elements entering per time unit. This input rate can be the result of a preceding transformation function; in this case no explicit definition is required. If the elements enter the function from the external environment then the input rate must be defined explicitly.

The flow rate of elements passing through or leaving the transformation is the result of the transformation itself, so they are not defined explicitly. Instead they will be measured.
This leads to the following general property types.

Flow properties are: Input set
Throughput set (Work In Progress)
Output set
Flow rate: number of elements per time unit entering the input set.

A flow element has: Properties that will be changed or used by the transformation
Properties of set membership

Referring to the example of table 4.2. the following properties are recognized.
The inventory is the input set of the product flow of Operation, the order list is the input set of the order flow of Perform and no input set of the resource flow of Use is defined.
Operation and Perform don't have throughput and output sets defined in their respective flows. The Use function has two sets representing the throughput set: an idles list and a busy list. Each resource can be in only one of these sets. The conjunction of both sets represents the total number of resources in the Use transformation. The throughput set is split to reflect the "state" of the resource: idle or busy. It illustrates the use of set membership to represent state properties of elements.
Flow rates are only indicated globally. Both the order flow and the product flow are generated from the environment; for this purpose two generating functions are introduced.

Their process descriptions contain the sentence: "Wait for next arrival". This sentence can be extended to quantify the rate; "next arrival" for example could be replaced by "20 minutes" or some stochastic expression as "sample from exponential distribution with average 20 minutes".

Tasks are the only elements having a property defined: the execution time. There are, however, some other properties implicitly defined by the process description itself. "An order consists of one task only" and "there is only one resource". Changing these properties will lead to adaptation of the process description.

4.5.1.2. Properties of flows between aspects

To define these properties the black box of the Operate function is opened as illustrated in figure 4.4. In this figure, sets are represented by triangles, "I" stands for the Input set, "T" for the Throughput set and "O" for the Output set.

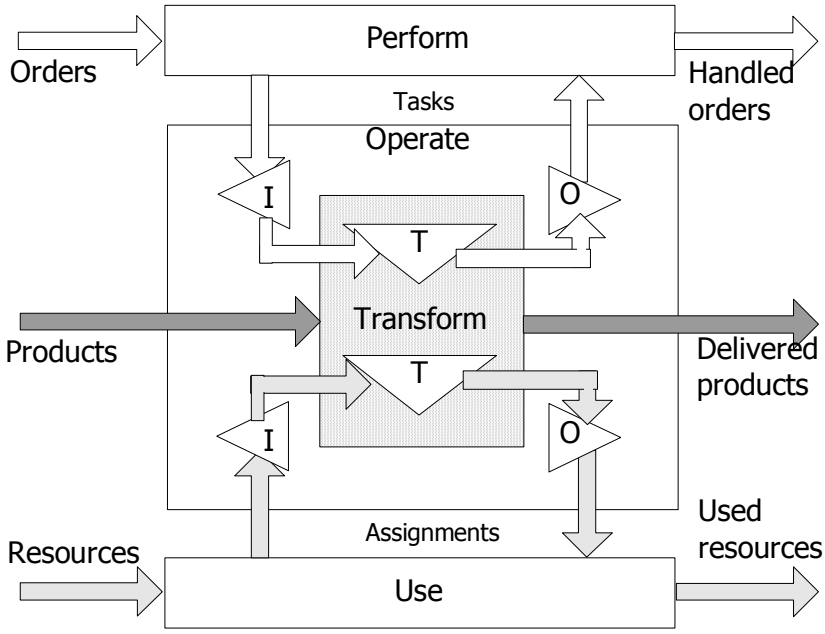


Figure 4.4: *The course of flows between perform, operate and use.*

Task and assignment flows are considered input flows of the operation function. In the example of table 4.2. only one input set has been defined explicitly: the task list. For readability purposes the input set of resources has been implicitly used by "Assign resource to operation" and "Wait until resource assigned". If the input set of resources would have

been called assignments, then this can easily be rephrased by "Put resource in assignments" and "Wait while assignments is empty".
In general the properties of these flows are therefore the same as the properties of the horizontal flows.

The concept of sets clearly facilitates the expression of state changes of a function. It even supports decision making in cases when these sets can contain more than one element, because automatically a selection method is required. In the example all selections are made in a FIFO-order, but it can easily be changed to more compex strategies with the same concept.

4.5.1.3. Properties of the control and transform functions

Most state properties can be derived from the properties defined so far (flow rates, sets including their contents), when functions are considered black boxes. Contents of the black box can be represented with the set concept of the flow properties; the dimensions of a function can easily be derived, if the entities have properties of space, weight etc. Measuring the length of stay of elements in a set even results in properties as lead time, waiting time etc. The set concept can thus be used to represent properties of the black box as a whole. One property however is not considered yet: disturbances. The exact moments of disturbances are by definition unpredictable, but disturbances can well be represented by a flow and its rate (the well known Mean Time Between Failure MTBF). These flows are described in analogy with the stochastic flows from the environment. Elements of a disturbance flow are called disturbance; the only special property compared with the other flows is, that immediate reaction is required when a disturbance "arrives". A disturbance influences the running period of the process. How to deal with this property will be explained paragraph 4.5.2.

4.5.1.4. Properties of vertical flows (flowing through control)

Data flow through the control functions vertically. Usually transformation functions are assumed to react on the data resulting from control. The control function itself may or may not use periods to transform data.

The control function of figure 2.9. consists of two connected data handling functions:

- *the standardization function*, consisting of initiation and evaluation.
  The initiation function

    - translates the needs of the environment into standards for all significant properties of the transformation
    - may change the standards in case of structural deviation of the results of the transformation.

  The evaluation function

    - measures the results of the transformation
    - reports structural deviations between standards and results to the standardization function.

- *the intervention function* containing feedback, feed forward and repair of deficiencies. They all react to individual deviations between measured ("real") values and standard values .

The properties of control functions are often quite specific. The standards must be formulated in terms of the contents of the transformation. Usually this results in standards for the number of elements in sets (representing maximum or minimum inventory levels, number of tasks waiting, available resources etc.) and the expected lengths of stay in sets (representing desired lead times, accepted waiting times etc.). It depends on the stage of the design process to what level of detail the control function is described. Paragraph 4.3. explained the relation between input, efforts and results. If two out of these three factors are known, the third factor can be determined. Assuming the input of a function is known, there are two different types of goal setting for simulation modeling:

- Determine the efforts to achieve the desired results
- Determine the results under the condition of given efforts.

The first type is characteristic for decision support during function design, the second type for decision support during process design. During function design control will usually not be

described. Here the goal is to determine expected values for the properties mentioned by means of best and worst case analysis and by sensitivity analysis. These expected values can be used as standard values during process design. Control functions will usually evolve during process design. This means that during design it must be possible to add standard values to the sets of a transformation and to the output flows. This requirement will be investigated further in paragraph 4.5.3.

4.5.2. Characteristics of "Periods"

4.5.2.1. Discrete and continuous systems

Periods describe intervals in time, during which the state of a function does not change with respect to its significant properties. Systems where these 'constant' periods occur and the state changes instantaneously, are called discrete systems. In continuous systems the function states change continuously. If a continuous system can be modeled in such a way, that the values of its continuous properties are only significant at discrete moments in time, then the behavior of such a system can be modeled by means of periods. Therefore the distinction between discrete and continuous systems must be considered a modeling criterion; it's not an a priori difference between systems.

4.5.2.2. The state of a process

It can be concluded from the definition of that a period starts after a state change and ends by another state change of the function. Between these state changes the system's clock time advances. The system's clock time is the time all functions are synchronized to.
A state change is represented with a moment in time and a (finite) description of the property values changing at that moment in time. A period requires a state change to start. From that point on, it is sufficient to specify the moment in time of the first future state change to fully define the coming period. Knowing this moment in time, the period can be described by "advance system's clock time until next state change".

The fact that during the period no state change occurs, does not mean the function is "doing nothing" or "makes no progress". It only expresses that the current state remains unchanged

with respect to the significant properties, but the current state can be 'working' (of a machining function), 'driving' (of a transportation function) or be just 'waiting' (of a transformation for a next job). So "work until next state change", "drive until next state change" or "wait until next state change" all express a period (see also tables 4.1. and 4.2.) It is even one of the major advantages of using periods - with respect to communication between disciplines- that they can be expressed with verbs related to the ongoing activity. The general verb "advance" will be used in this thesis to express a period.

The other aspect of a period definition concerns the phrase "until next state change". Three situations are distinguished:

1. the moment in time of the next state change is exactly known.
2. The moment in time of the next state change is unknown, but the condition for it to occur is known.
3. Both the moment in time and the condition are unknown.

Ad 1. If the moment in time is exactly known the duration of the period is exactly known, so it is fully defined by "advance x time units". For time units, the usual units as seconds, minutes etc. can be used. The only condition is: they must be clear.
In this case, the process is called to be in the state *"scheduled".*

Ad 2. Here the moment in time is unknown, but the condition is known. In this case an unambiguous expression of this condition is sufficient. A condition can be expressed in two ways:

- the period continues as long as a condition holds; the period is defined by "advance while condition is satisfied"
- the period ends if a condition is satisfied; the period is now defined by "advance until condition is satisfied"

In the example of table 4.2. both alternatives are used. The operation specifies a period with "Wait while task list is empty" and another period with "Wait until resource assigned".
The process is now called to be in the state *"conditioned".*

Ad 3. Both the moment in time and the condition are unknown: the length of the period is indefinite. For these cases the word "advance" without further specification is used. No state change will occur, unless another function (in the surrounding system or the environment) creates one. It can only be awakened by external circumstances.
In these cases the process is called to be in the state *"suspended".*

To conclude this paragraph, there is one state of a process left that needs to be defined. Until now the states suspended, scheduled and conditioned are defined. They all reflect periods, but at the very moment in time of a state change there is no period defined at all. Therefore at the moment a function handles a state change the process is called "active". A process automatically becomes active at the end of a period in its process description.

The conclusion is that the 'value' of behavior consists of two parts: a process description and a process state, which so far can take the values active, scheduled, conditioned or suspended.

Figure 4.5 summarizes the state transitions of a function's process. These transitions are caused by the function itself by means of "Advance" clauses in its process description.

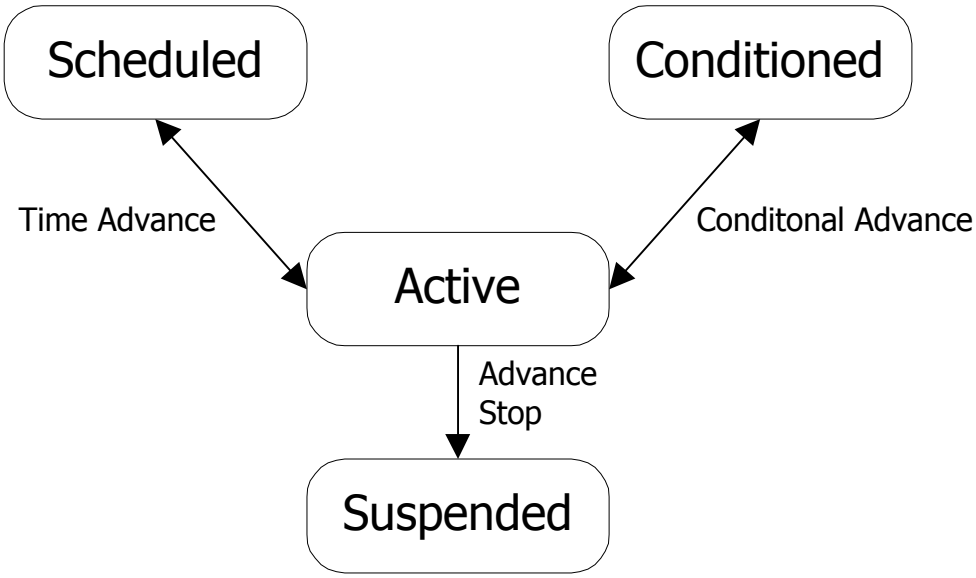


Figure 4.5: *State transitions caused by "Advance"*

4.5.2.3. Process interventions

If a process is suspended, it will not proceed without intervention from the function's environment. Therefore, facilities are required to intervene with the progress of the process of a function. Interventions can be divided in two categories:

- regular interventions
- disturbance interventions.

Regular interventions can best be explained with the example of table 4.2. Consider the next lines from the processes of Operation and Use:

Process Of Operation	*Process of Use*
Enter Request list of Use Wait until resource assigned	Select first Operation from Request list Assign resource to Operation

This part can also be formulated as follows:

Process Of Operation	*Process of Use*
Enter Request list of Use Wait	Select first Operation from Request list Assign resource to Operation Resume Operation

Here the process of Operation proceeds only if another process (in this case the process of Use) tells it to do so. For this purpose the expression "Resume" is introduced.

Resuming a process of a function is expressed as resuming the function, because each function can have only one process description. By using resume, the process of operation resumes progress with the line directly succeeding "Wait". Resume introduces a new state change from outside the function. The causing function must assure itself, that the receiving function is able to respond to this state change (i.e. is "suspended"). If the Operation's process would be in a scheduled or conditioned state, the period of Operation would have

been defined already (in other words Operation is watching the clock time or the condition specified for that period and nothing else). Operation would not recognize the state change introduced.
For reasons of readability (and therefore communication), it is preferred in this thesis to minimize the use of indefinite periods (and thus to use formulations like the ones of table 4.2).

There are however two regular interventions that necessarily are caused by the environment of the function:

- the process of a function must once be started
- there may be circumstances where the process must be stopped.

To start the process at some moment in time, the sentence: "Start function at T" is used (the process now changes from a suspended into a scheduled state). As soon as the clock time T is reached, the process will start with its first line and follow its description.
To stop a process the sentence "Stop function at T" is used and the process description is abandoned at clock time T.
Before starting and after stopping a process the state of the function is considered suspended.

Disturbance interventions interrupt a process, whatever state it is in, and the function must react immediately to it. The only exception is a process being active. Such a process cannot be disturbed, because the active state takes place instantaneous: the simulation clock does not advance. As soon as the process description encounters an advance statement, the disturbance is effectuated.
Mostly the reaction of a function will be to do nothing until the disturbance disappears. This is a typical reaction in case of bad weather conditions or technical disturbances. Repairing disturbances can be a complex task of course, but they are usually performed by other functions (maintenance or service). A disturbance will be expressed with "Cancel function". The state of the process immediately changes to "cancelled". Resuming the process after resolving the disturbance can occur in three ways:

- the process must finish its period that was proceeding before the disturbance. The process returns to the original state. If it was scheduled, the duration of the period

will be the remaining period at the moment it was cancelled. In terms of the process description the process "proceeds with the line it was executing". This result can be accomplished with the sequence:

Cancel function (start of disturbance)

…

Proceed function (end of disturbance)

In between, the causing function can describe its actions to deal with the disturbance.

- The process must resume, but skip the period it was in at the moment the disturbance occurred. For example a product that was being handled, is removed during the disturbance and transferred to another function. The process may now proceed as if the period finishes and deal with the next state change. Again in terms of the description, the process resumes with the line immediately following the period before the disturbance. Now the sentences will read:

  Cancel function (start of disturbance)

  …

  Resume function (end of disturbance)

- Finally it may be required to restart the process completely. In this case the sentences are:

  Cancel function (start of disturbance)

  …

  Start Function (end of disturbance)

So after a Cancel phrase a process may proceed, resume or start over again.

The process interventions cause transitions of the function's process state. They are shown in figure 4.6. It shows that cancel and proceed can be requested for any state of the process that was initiated with an advance statement. A stop request changes the state of the process into suspended. Proceed always returns the process into the state it was in at the moment of cancel. Start and resume always result in the process to become scheduled. Start forces the process to execute its description from the first line, resume forces it to continue its description with the line immediate following the point where it was cancelled.

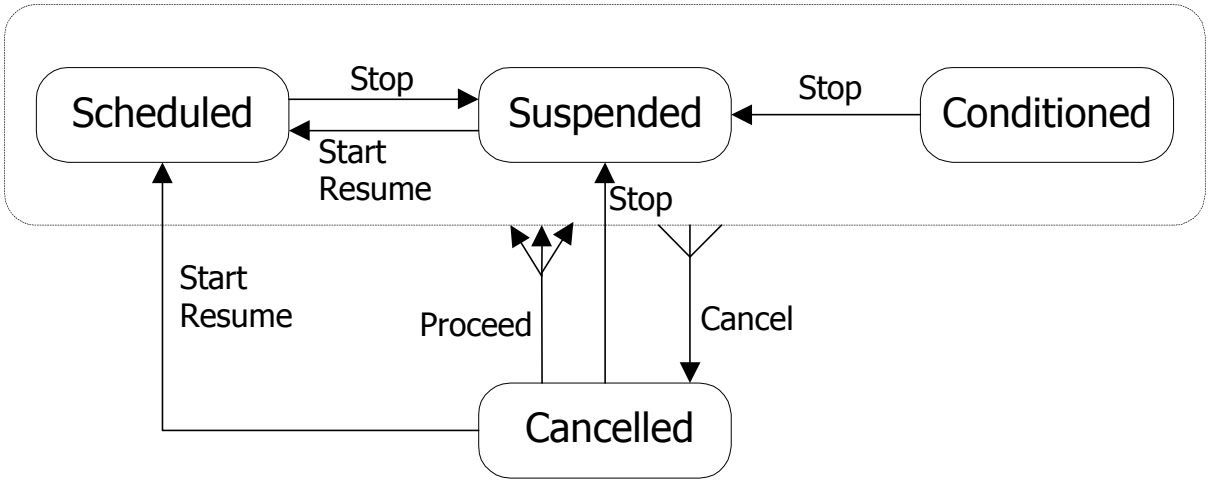


Figure 4.6: *State transitions with an external cause.*

4.5.3. Aggregation

Each function is part of a function structure. The static relations are visualized by the PROPER model and the time dependent relations have been described in terms of processes in the preceding paragraphs. During the design process a hierarchy of PROPER models will be created representing a number of aggregation strata. Each next aggregation stratum is based on the results of decision making in preceding aggregation strata. This shows itself in:

- refining standards and efforts.
- Dividing functions into a structure of subfunctions.

Both actions result in added detail to the original input signals. In figure 4.7. the positions are shown where input signals are transferred between aggregation strata.

A function XY is shown at aggregation stratum N. Zooming in one step results in two subfunctions X and Y at aggregation stratum N+1. There are two kinds of areas where signals are transferred:

- areas A: for the physical signals of order, product and resource flows
- area B: for the control signals

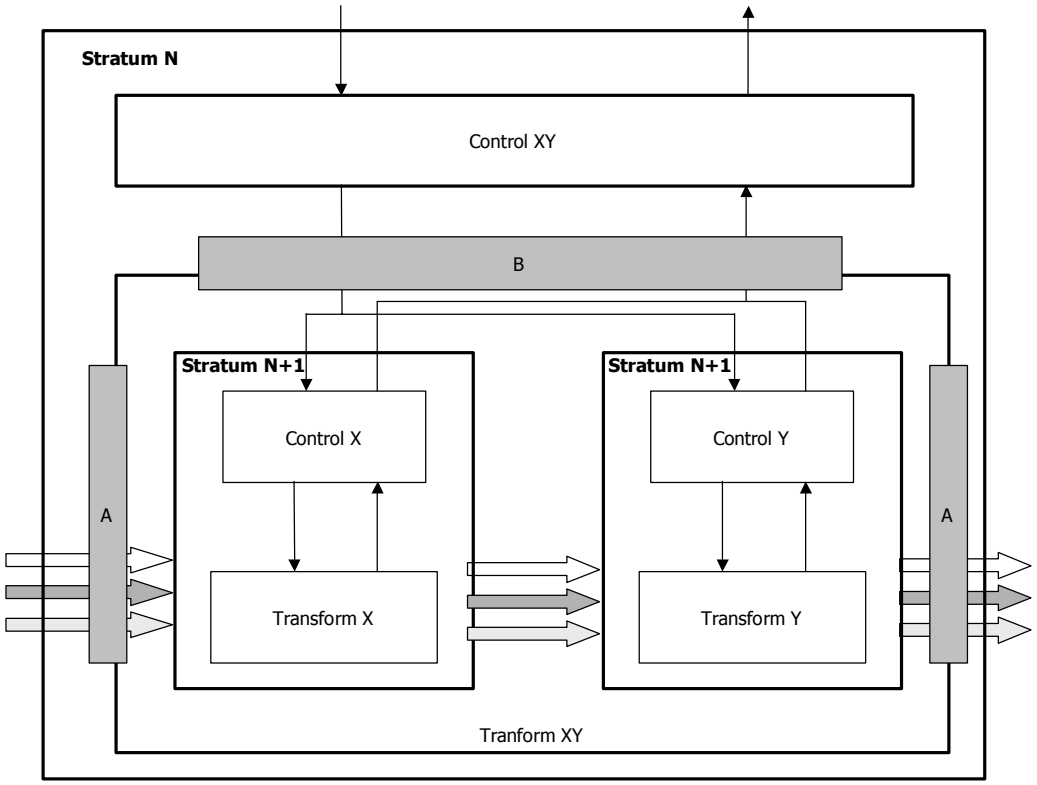


Figure 4.7: *Signals passing between aggregation strata*

First focusing on the areas A, an example will be used again to illustrate the effects of aggregation.

Example 4.4. Suppose the transformation XY at stratum N represents a painting function. Transformation X at stratum N+1 represents cleaning and transformation Y represents coloring. Products can be painted in two colors; 50% of the products must be painted color A and 50% color B. If two successive products have different colors then a cleaning action is required first. This takes 2 hours. The painting operation itself takes 5 hours. The function model at stratum N does not distinguish between cleaning and coloring. The following table shows the process description at each stratum as if they were developed completely stand alone, without any exchange of elements.

Processes at stratum N	***Processes at stratum N+1***
Process of Painting *Repeat* *Wait while inventory is empty*	*Process of Cleaning* *Repeat* *Wait while inventory is empty*

Select first product from inventory *If Sample from Uniform[0,1] < 0.5* *Work 2 hours* *Work 5 hours*	*Select first product from inventory* *If product's color differs from last color* *Work 2 hours* *Resume Coloring* *Wait while Coloring is busy* *Process of Coloring* *Repeat* *Work 5 hours* *Wait*
Process of Generate_products *Repeat* *Put new product in Painting's inventory* *Wait for next arrival*	*Process of Generate_products* *Repeat* *Put new product in Cleaning's inventory* *Determine product's color* *Wait for next arrival*

Table 4.3. Two aggregation strata compared

The description at stratum N can be considered part of a model to estimate the painting costs to be made. No specific color information is needed and it is therefore modeled by the sentence "If Sample…". The descriptions at stratum N+1 can be used to determine product selection strategies. In the example the first product is chosen, but this can also be rewritten with "Preferably select a product with the same color first" to minimize the number of cleaning operations. The main differences between the processes at stratum N and N+1 are:

- the function Painting has been split into two functions Cleaning and Coloring.
- A product is assigned a property 'color' at stratum N+1.

The models can be used in a hierarchic structure; for example the description of stratum N is part of a complete manufacturing model and one wants to check the effects of the descriptions at stratum N+1 to the results of stratum N. In this case the process description of the Painting function becomes dependent on the Cleaning and Coloring. The lead time of the product is now determined at stratum N+1, so the sampling phrase and resulting periods of 2 and 5 hours must be replaced by 'Work'. The Painting process now simply waits until a signal is received from stratum N+1 that a painting operation has finished. The selection of products has also shifted to stratum N+1, so this line must be skipped from the description

of Painting. The products are still generated at stratum N (in this case by a generator, but they can also be the result of preceding functions at this stratum) and this cannot be shifted to the next stratum. The generator at stratum N+1 must be removed.

According to figure 4.6. there are two moments in time the strata exchange elements. The first one is when an element enters the lower stratum and the second one where it leaves the lower stratum (or enters the higher stratum). These moments can be positioned precisely and are assumed to be caused by the stratum the element leaves. Leaving a stratum will be expressed by "Send element to stratum X".

Products must enter stratum N+1 at the moment they enter the painting function; this is where they are put into the Painting's inventory. Therefore the description of the generator must be expanded. Products leave stratum N+1 at the moment they are finished by the coloring, so this description will also be expanded. If elements can be send to a stratum, the addressed stratum must be able to receive them. For this purpose a reception function is introduced at each stratum.
This leads to the following hierarchically related descriptions.

Processes at stratum N	***Processes at stratum N+1***
Process of Painting *Repeat* *Wait while inventory is empty* *Work* *Process of Generate_products* *Repeat* *Put new product in Painting's inventory* *Send Product to stratum N+1* *Wait for next arrival*	*Process of Cleaning* *Repeat* *Wait while inventory is empty* *Select first product from inventory* *If product's color differs from last color* *Work 2 hours* *Resume Coloring* *Wait while Coloring is busy* *Process of Coloring* *Repeat* *Work 5 hours* *Send product to stratum N* *Resume Cleaning* *Wait*

Process of Reception_from_N+1	*Process of Reception_from_N*
Remove product received from Painting's inventory *Cancel Painting* *Resume Painting*	*Determine product's color* *Put product in Cleaning's inventory*

Table 4.4. Hierarchically related processes

The process of Painting will be 'working' now as long as there are products in its inventory, just as it would be in the description of table 4.3. Selecting products and operating them is now completely moved to the next aggregation stratum.

The transfer of control signals is represented in figure 4.7. by area B. In paragraph 4.3. they were divided into standards and interventions. Transferring standards is a matter of defining new (more detailed) standards. Different aggregation strata are however unaware of each other's standards. In example 4.4. stratum N is unaware of cleaning and coloring as separate functions, so it certainly doesn't know what kind of standards are defined. The same statement holds for measuring results in terms of standards. There is one condition however: the functions must preserve the same behavior as in the stand alone approach. For example if the occupation of the painting function in table 4.4. is calculated by measuring the periods where this function 'works' or 'waits', then this can be achieved in the same way as in table 4.3. The measured values can be different of course.
Interventions can be handled similar to the physical signals. Each intervention at aggregation stratum N may have consequences at all lower strata. For example, if a control at stratum N in example 4.4. decides to stop the painting function for a while, then this action must be transferred to stratum N+1, because cleaning or coloring must be stopped also. This can all be achieved by using the same send and reception concept.
Finally disturbance signals are considered local to each aggregation stratum and these signals do not pass through strata. A disturbance signal is received at one stratum and may influence the processes at another stratum (up or down), but this must be done by a control function at the receiving stratum. For example a disturbance of the coloring function also interrupts the painting function at the higher stratum. This intervention is equal to the 'stop' sequence discussed earlier.

4.6. Conclusions

In the previous paragraphs, it has been shown that the behavior of a function can be completely described with a coherent frame of reference, which is based on natural language. It also uses natural language to express decision making items and offer the flexibility to describe behavior in any circumstance and to any degree of detail. The number of strictly defined terms in this concept is limited and as easy to adopt as the schematic representation of static function structures such as the PROPER model. Above that, the terms are not bound to a specific discipline and thus interdisciplinary. As such they support the common description and understanding of the system's behavior.

Behavior is a time-dependent property and the notion of time (or periods) is introduced in process descriptions by using time consuming verbs. The interpretation of process descriptions is straightforward and can be achieved by reading them function by function.

For human interpretation the descriptions seem unambiguous. However to be used for simulation purposes they must be unambiguous with respect to automated interpretation. The next chapter will show that the descriptions must be formalized further for implementation in a software environment. This formalization is considered a task for the simulation expert. Here the interdisciplinary approach is considered complete and the next chapters are considered part of the simulation expertise.

Chapter 5

The Process Description Language

"England and America are two countries separated by the same language"

- George Bernard Shaw -

5.1. Introduction

The behavior of a function was described in chapter 4 in a structured but informal way. In this chapter, a Process Description Language PDL will be introduced for the design of a computer simulation model. The description is made less informal, but still not completely formal. There are two reasons to be not completely formal:

- making a simulation program is an iterative process. Not all details and algorithms are known beforehand. Programming and testing will start before everything is clear in detail and often iterations with informal descriptions as the ones in chapter 4 will be required. "Empty shells", only denoted by clear headings, must be allowed.
- the PDL must be 'programming language independent' as much as possible. Each programming language uses its own terminology and syntax and has its own restrictions. Forcing these to be used in the PDL would easily lead to ambiguity of terms between the "design language" and the "programming language".

The major criteria for the design of the Process Description Language PDL are:

- the ability to be unambiguously implemented in a programming environment. For that purpose the process descriptions of chapter 4 must be made unambiguous for the simulation modeler.
- The ability to communicate the model with the problem owners. This supports the simulation modeler in the verification process (i.e. proving the correctness of the simulation model) and extends the usual way of debugging a model.
- The ability to be easily maintained and extended.

In the next paragraph, an overview of informal system methodologies is given and a choice is made that best suits the simulation purpose. By means of an Object Oriented Computing approach the structure of a PDL description will be derived, consisting of a definition part, a

process part and an initialization part. Finally each part will be elaborated in detail, assuming the operation of the logistic system takes place in a simulation world.

5.2. Informal system methodologies

Kiniry [1998] distinguishes three generations of informal system methodologies:

- First Generation Methodologies focus primarily on building small to medium scale object-oriented systems. Most of these methodologies are either process-oriented (e.g. Booch [1994]) or data oriented (e.g. Shlaer and Mellor [1988]).
- Second Generation Methodologies incorporate constructs that assist in the specification of large-scale and/or distributed systems. Additionally, these methods are capable of a more formal specification of systems. Best known example is the Unified Modeling Language UML [Jacobson et al., 1999]. UML uses diagram representations belonging to one of four classes: System Interaction, Static system structure, System Behavior diagrams and Implementation Diagrams.
- Third Generation Methodologies are now emerging from the modeling community. They incorporate ideas and terminology from cognitive science, knowledge engineering and representation, and information theory. The Object-Oriented Change and Learning (OOCL) modeling language belongs to this generation [Swanstrom, 1998]. Basic notions of OOCL are agents, organization, constellation, resources and processes.

First and second generation methodologies are used to design just software systems; OOCL is used to design and understand business systems as a whole -- software is considered only one component of a complex business system.

In this thesis, the main interest is a methodology to design a simulation model of a logistic system. The PROcess - PERformance model (PROPER - model) is already defined to represent the system design. If the design would be directly translated into UML or OOCL terminology, this would reintroduce the walls between simulation and other disciplines; in the previous chapters of this thesis, a well-defined meaning has already been assigned to terms being used differently in UML and OOCL.

Whatever generation a methodology belongs to, it uses the basic idea of object orientation. Kiniry [1998] considers the object oriented approaches of Booch [1994] and Shlaer and Mellor [1988] as the historic archetypes of behavior and data-oriented methodologies. He states: "Data-oriented techniques focus on describing the data that is at the core of all computational systems; behavior-oriented techniques concentrate on describing the activities of a system, independent of the data representation. Data-oriented modeling techniques primarily originate in the database and information processing communities. Behavior-oriented techniques, on the other hand, originated with the 'traditional' computational-oriented practitioners".

As most software designers Kiniry doesn't mention the procedural / relational approach (or structured programming) as a methodology preceding the first generation methodologies. Object-orientation has become a paradigm of system design. However, as stated on the website of Findy Services [2002]: "The procedural paradigm has not died, it has simply found its place as the Yin to the Yang'.

Simulation primarily deals with behavior; Chapter 4 showed that behavior is represented by a procedural description. For this reason the object-oriented approach originating from the procedural approach will be used here. It will be called 'Object Oriented Computing' (OOC) [1] from now on. This description however is not to be executed in the usual uninterrupted sequential way, because it is implicitly synchronized to a 'simulation clock'. A possible conflict with OOC methods can therefore not be excluded beforehand. The application of the object-oriented approach to model the functions and behavior of logistic systems will be investigated step by step. In this thesis the goal of modeling is not to force the use OOC but to make use of OOC for a correct mapping of the PROPER model and its behavior. This is even more necessary, because usually OOC is applied and illustrated for software with a well defined purpose in a stable system structure. During the design process this is not the case.

5.3. Object Oriented Computing (OOC)

The approach of Booch [1994] is used here to apply object oriented computing to the PROPER model of a logistic system with the goal to simulate the behavior of the system.

[1] In this thesis the term 'computing' is preferred to the usual term 'programming'. Object orientation includes more than programming alone.

In his book 'Object-Oriented Analysis and Design with Applications', Booch [1994] describes a complex system as a combination of seven concepts: two hierarchical structures (class and object) together with five common attributes of a complex system.
Where Booch mentions a 'system', he implicitly means a 'software system'. Therefore the attributes of complex systems according to Booch will first be viewed in relation with the PROPER model of a logistic system. The five attributes are:

1. *"a complex system is composed of interrelated subsystems that have in turn their own subsystems and so on, until some lowest level of elements is reached".* This fully complies with the PROPER model. Above that the approach of this thesis restricts subsystems to be PROPER models again.
2. *"The choice of what elements in a system are primitives is relatively arbitrary and is largely up to the discretion of the observer of the system".* The PROPER model contains no explicit definitions of these primitives. In theory one can keep on zooming into lower strata and finally reach some base stratum where the elements cannot be divided any further. At any stratum one can also stop zooming and consider the elements indivisible; it depends on the goal of the design project.
3. *"Intra-element linkages are generally stronger than inter-element linkages. This fact has the effect of separating the high-frequency dynamics of the elements – involving the internal structure of the elements – from the low-frequency dynamics – involving interaction among elements".* By 'dynamics' Booch points to state changes of the elements, which are reflected by a change of property values. This statement reformulates the principle of system boundaries and is in line with selecting subsystems within a system. Projecting this to the PROPER model it states that the relations between aspects, controls and transformations within the model are more frequent than the relations between two PROPER models.
4. *"Hierarchic systems are usually composed of only a few different kinds of subsystems in various combinations and arrangements".* The PROPER model consists of "process-performance" subsystems. The number of combinations and arrangements however is large, but each combination has the properties of a PROPER model again.
5. *"A complex system that works is invariably found to have evolved from simple systems that worked. A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over, beginning with a working simple system".* In this thesis this attribute is considered a recommendation for the use of aggregation. Starting a design at a high aggregation stratum and

gradually zooming in and adding granularity is a necessary condition to get the system work successfully.

As a conclusion, the attributes apply to the PROPER model of a logistic system.
To represent the decomposition of systems into subsystems Booch introduces two hierarchies:

- a class hierarchy, containing element definitions of the system and their relations
- an object hierarchy, representing the state of the system.

Objects are considered "observable things" in the real world: a book, a car, a dog etc. An object is defined by its identity, its state and its 'behavior'. The state of an object is a set of attribute values at a particular time. In software, 'behavior' refers to data handling activities and it is modeled by means of services (the term "methods" is also used). A service acts upon the internal attribute values of the object.

There are three basic modeling principles in OOC:

- *Inheritance*. the starting idea is to describe similar objects by means of a 'class'. A class defines both the attributes and the set of services common to all objects in this class. Analogously, similarities in classes can be described by a superclass and so on. In this way a hierarchy of class definitions is created. Objects in a class 'inherit' all attributes and services of the classes above its class in the hierarchy.
- *Encapsulation*. Each object controls its own state changes; other objects are not allowed to change the state of an object directly, but must 'call' services offered by the object. The environment only knows what an object does, but doesn't know how the object does it. This is in fact a client – server view: the environment asks for services, the object performs operations. The set of services is also called the interface of the object.
- *Polymorphism*. Many definitions of polymorphism exist ([Strachey, 1967], [Cardelli, 1985], [Meyer, 1988] , [Booch, 1994]). In this thesis polymorphism will be considered the possibility for an object to perform different operations depending on its context. This only means redefinition of a service, depending on the level of the class in the class hierarchy.

Applying these principles is called *abstraction*.

To be clear about the terms *property*, *attribute* and *service* the following distinction will be made from now on. A property is part of the state of a function. Properties are represented in an object oriented way as attributes of a class, while services are defined for a class to obtain the value of attributes or perform operations using the attributes.

During the execution of a program objects will be created and destroyed. The creation of an object is called 'instantiation' and an object is an 'instance' of a class.

Object Oriented Computing consists of 3 steps:

a. Object Oriented Analysis (OOA): modeling the system by identifying classes and objects, that form the vocabulary of the problem domain.
b. Object Oriented Design (OOD): inventing the abstractions and mechanisms that provide the behavior required for this model.
c. Object Oriented Programming (OOP): implementing the design into a software platform.

The programming part will be described in chapter 6. The following paragraphs explain the object oriented analysis and design steps for simulating a PROPER model of a logistic system and these steps can be applied to all object oriented software platforms.

5.4. Object Oriented Analysis of the PROPER model

5.4.1. Introduction

The goal of OOA in the context of the PROPER model is to identify objects and classes with their attributes and services to be used for simulation. A software implementation therefore must reflect this simulation purpose by considering the model and its elements to be part of a simulation environment.
The use of the PROPER *model* is briefly described: "structuring the *functions and flows* in a logistic system by offering decision making support on different *aggregation strata*".
Aggregation strata range from the single function representation of figure 2.8. to the

detailed stratum where (eventually) real resources will be tested in a *simulation environment*.

The entities that can be derived from the above description are:

- a simulation environment
- the model
- functions and flows to be simulated
- aggregation strata

At the following pages these entities will be examined and classes, attributes and services based on an outside view (what is an object asked for) will be derived. The inside view (how the objects work internally) is a matter of implementation and will be discussed in chapter 6.

5.4.2. Functions and flows

In chapter 4 the properties of functions and flows (and the elements of a flow) have already been defined. The common (and general) properties are summarized in table 5.1.

Functions	Flows	Flow elements
Process State Set membership	Flow rate Flow elements	Set membership

Table 5.1. Common properties

Properties like input set and output set are not mentioned here, because they are not generally present. Contrary to this, the set concept itself is a general concept, representing collections of elements and functions for different purposes. So the first class to be defined is the class of "`Set`".

Using this class, the property 'set membership' of functions and flow elements will be further analyzed in paragraphs 5.5.3 and 5.5.4. Common services of the set class will perform operations like adding elements to the set, removing elements from the set, sorting elements, calculate statistics about the number of elements in the set etc.

Functions differ from flow elements by the process property and (consequently) the state property (static, current, scheduled, interrupted, etc.). In this thesis however functions and flow elements will not be modeled by two different classes. The use of two different classes is only profitable if functions and flow elements can always be distinguished by the process property. It may occur however that the resource flow and product flow fuse together during (part of) the process. For example in simulating a public bus service the product flow represents passengers to be transported from source to destination; the function is fulfilled by a bus. The functions of loading and unloading the bus at a bus stop will be usually performed by the passengers themselves, playing the role of the resources for this function (the function 'loading' is then fulfilled by the task 'get on' of a passenger and 'unloading' by 'get off'). This is the reason why there won't be made a distinction between the class definitions of functions and flow elements in this thesis: they may have or have not a process property.

Both functions and flow elements will be represented by a class "`Simulation_Element`". The process property will always be represented by the "`Process`" service.
With respect to the 'set membership' property a `Simulation_Element` now may actively enter or leave a `Set` or passively being added to or removed from a `Set`. The passive set membership services are already covered by the `Set` class itself. The active services belong to the `Simulation_Element`.

Services required from a `Simulation_Element` with respect to the defined class attributes are:

- process control: start, stop, cancel and resume its process
- inspect the state value of the process
- enter and leave sets.

5.4.3. The model

The model contains objects of the classes `Simulation_Element` and `Set`. It also contains the simulation control with respect to the definition of the initial model state and finish time or condition. The initial model state contains:

- the objects of the classes `Simulation_Element` and `Set` present at the start of a simulation run
- the initial (simulation) clock time
- the starting sequence of the processes of the `Simulation_Element` objects.

From this it is concluded that a model consists of three parts:

1. Definition of classes
2. Description of processes of objects
3. Initialization.

The model itself requires services from the objects defined; the only service that can be asked from the model itself, is to `'Initialize'`. The class `'Model'` is hereby introduced.

5.4.4. The simulation environment

The simulation environment controls the progress of a simulation run. Progress control means 'advancing the simulation clock time'; the value of the simulation clock time is advanced when no `Simulation_Element` changes its state (see the periods of chapter 4). The environment is considered a class offering a clock functionality and will be called `"Simulation_World"`. The only service required by a model during a simulation run with respect to the clock is the current value of the clock time. For this the service `"Now"` is introduced. As shown in chapter 4 the process description of a `Simulation_Element` contains phrases to express the fact its state will not change during a certain period. The expression "Work 2 hours" means the simulation clock time may advance 2 hours as far as the element is concerned. It's up to the `Simulation_World` object to deal with these requests and for that purpose its general service `"Advance"` is introduced. This service can be used in three ways:

- `Advance` for a certain time
- `Advance` until or while a condition is true
- `Advance` indefinitely.

The Simulation_World may contain more than one model. During a design project many function structures will be created, that are simulated both separately and combined. The models that are combined into one model configuration will all have to be synchronized to the same simulation clock. In a Simulation_World there is one and only one clock.

To start a simulation run, the Simulation_World has to ask all models to 'Initialize'.

5.4.5. Aggregation strata

From the outside view there is no difference between models at different aggregation strata. The communication between different strata was already discussed in paragraph 4.5.3. and was considered a communication between model objects. Each model therefore needs a "Receive" service. Communication between models at the same aggregation stratum should also be supported, because the model configuration is not necessarily composed of models at different aggregation strata. In this case the communication is considered a communication between simulation elements (for example two succeeding functions may exchange products). Each element therefore offers a 'Receive' service also.

Figure 5.1. summarizes the identified classes with an indication of their services as a result of the preceding analysis. An arrow towards a class denotes a service acting on the internal attributes of the class. An arrow leaving a class denotes a service returning information about internal attributes. All classes offer services to inspect their internal values. These services are not shown in the figure. The internal clock of the Simulation_World is shown explicitly.

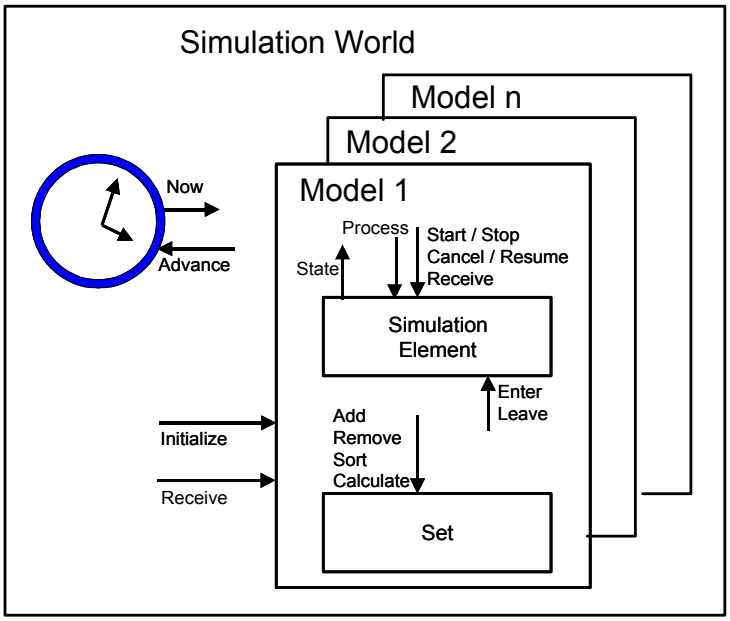


Figure 5.1: *Classes for the simulation of a PROPER model*

5.5. Object Oriented Design of the PROPER model

5.5.1. Introduction

All classes are identified, and the abstractions and mechanisms are now to be defined to provide the required "behavior". Behavior of an object in a software environment is provided by a set of services of that object. Each service inspects attribute values or acts on attribute values. In a simulation software environment behavior not only means the representation of the time dependent behavior of an object in the logistic system. Behavior also includes the services an object is required to deliver to the simulation modeler for programming purposes.

During Object Oriented Design the difference between the class notion and the object notion is essential to understand the model descriptions. Therefore the next paragraph explains this difference by using the usual computer implementation terminology. After that the services for each class of figure 5.1. are defined.

5.5.2. Classes and objects

Objects are the "individuals in" or "members of" a class, with all the attributes as specified by the class definition. Objects have values assigned to the class attributes. For example a class defines an attribute of type color, then an object in the class has an attribute color with value (e.g.) 'red'. A class is a "blueprint" for objects; one class definition results in many objects with the same attributes but different attribute values.

Each object in each class must have a unique identity. People are used to denote objects by using unique names. This identity of an object is denoted by `'Name'` attribute and at the moment of instantiation a unique value for it is required. The `Name` attribute helps to interpret output presentations of the model. Computers however identify objects by using their unique memory location. By instantiation memory is reserved according the class definition and assigned to the object. This location can be referred to by means of a so-called 'object reference'.

Example 5.1. Suppose a task class is defined with attributes '`Name`' and '`Task time`'. There are two instances of the class: one with name 'TASK1' and task time 93.6, the other with name 'TASK2' and task time 112.0. The memory location of 'TASK1' is referred to by Taskobject1, the memory location of 'TASK2' by Taskobject2. The `Name` value of the first task can be achieved by 'Name of Taskobject1'.
The figure below shows the differences between reference and Name.

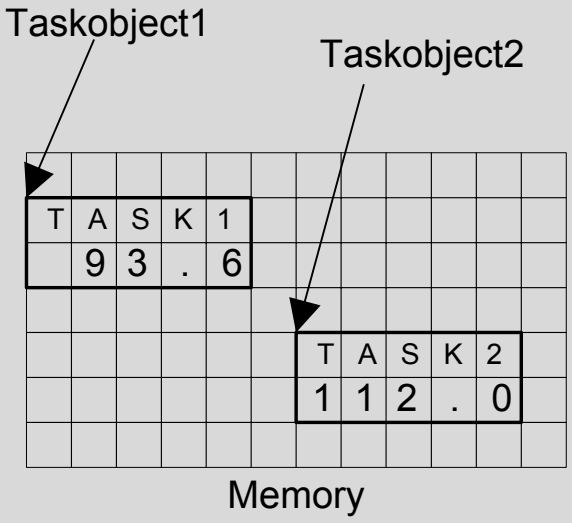


Figure 5.2: *Name of an object and reference to an object*

Where `Name` is an attribute of a class, a reference is not. References are used to tell the computer, which object is meant and they should therefore be considered normal 'variables'. During the execution of a simulation program, many objects may be instantiated. In paragraph 5.5.4. it will be shown that the set concept helps to minimize the number of references without loss of accessibility to objects.

Instantiation is a common service also. It is denoted by '`Create`'. The value to be assigned to the `Name` attribute is an input parameter for this service, because it has to be specified at the moment of instantiation. Input parameters are specified italic and between brackets. So instantiation is achieved by calling the service '`Create`(*name*)' of the class. The service returns a reference to the created object.[1] The definition scope of the object is the Model in which it is created.

During the execution of a simulation program many objects are only present during a certain period of time. Especially flow elements usually have this "life span". They can be removed from the model after they are transformed and are delivered to the environment. The memory location can be released and for this purpose all classes own a "`Destroy`" service. A mechanism like "garbage collection" where an object is automatically destroyed if there exists no reference to it, as implemented in many object oriented software environments, is not used in the approach of this thesis. In a simulation environment destroying an object is explicitly considered the responsibility of the modeler, because the object represents the physical presence of an object in the logistic system, requiring space and other resources. A function to remove or destroy objects can be an essential part of the system.

For all classes the definition now contains at least the services:

`Name`
`Create`(*name*)
`Destroy`

The next paragraphs explore the specific services for each class shown in figure 5.1.

[1] When many objects of the same class exist, it's quite a job to specify a unique name for every object. Therefore the `Create` service should support uniqueness by extending the input name in such a way that it will be unique.

5.5.3. The class `Simulation_Element`

In the context of this thesis the class `Simulation_Element` represents the functions and flow elements of a logistic system. Instantiating a `Simulation_Element` is creating it in the `Simulation_World`. By creating it, not only a memory location is provided, but also a time of arrival related to the simulation clock time of the `Simulation_World`. This time of arrival can be obtained during the life span of the object by calling the service '`Arrivaltime`'.

Another common service to be precisely defined, is the '`Process`' of a `Simulation_Element`. This service represents the time dependent behavior and in chapter 4 it was concluded that it should be described as a sequence of periods and momentary actions between these periods. The momentary actions between two periods are considered an indivisible program sequence and it will be called an 'event'. The term 'event' expresses the actions more clearly than the term moment in time as used in chapter 4. The generic representation of a '`Process`' service is therefore a procedural description composed of a sequence of periods and events. At this point the role of the simulation clock becomes important. Two aspects must be made clear now:

1. Simulation time and program execution time are quite different.

Example 5.2. Suppose a process procedure contains the following lines:

```
…
I = 0
While I < 100
        Wait( 1 day)
        Increment I by 1
…
```

It takes less than 1 millisecond for a PC to perform this sequence, but the simulation clock advances with 100 days.

2. There is an essential difference between program listing sequence and the simulation sequence. Each procedure in a computer program usually reflects the sequence of program execution, because there is only one processor interpreting it. The lines in a `Process` procedure however are not executed in this order. Every time the `Advance` service of the `Simulation_World` is called, program execution jumps out of the description and continues at a point in some other process description. The simulation clock determines the final execution sequence of procedure lines.

Example 5.3. Consider the process descriptions A and B below. Both processes start at time zero. Process A is scheduled to start before process B.

	Process A		*Process B*
A1	*Set Counter to 0*	*B1*	*Advance(2 hours)*
A2	*Advance(3 hours)*	*B2*	*Add 1 to Counter*
A3	*Add 1 to Counter*	*B3*	*Advance(6 hours)*
A4	*Advance(2 hours)*	*B4*	*Subtract 1 from Counter*
A5	*Subtract 1 from Counter*		

Table 5.2. Sequential versus parallel execution

Sequential execution would result in the following sequence of Counter values: 0, 1, 0, 1 and 0. The lines are executed in the sequence A1, A2, A3, A4, A5, B1, B2, B3 and B4. Taking the simulation clock into account the values will be: 0, 1, 2, 1 and 0. The lines are now executed in the sequence A1, A2, B1, B2, B3, A3, A4, A5, and B4. The usual program sequence is interrupted by the use of the Advance service of the Simulation World.

For process control the services have already been defined:

`Start(T)`: to start the process at some moment in time T

`Stop`: to stop the process of the object unconditionally

`Cancel`: to interrupt the execution of a process, remembering the remaining period length.

`Resume`: to resume the process skipping the remaining period (if present)

`Proceed`: to continue the process with the remaining period at the moment it was cancelled.

To apply these services correctly the `Simulation_World` must recognize three "starting" points in a process description:

- the primary starting point being the first line of a `Process` service
- a starting point at the line following a call to `Advance`
- a starting point at the line containing the 'current' call to `Advance`.

In chapter 4 the state property of a process was defined, with possible values Suspended, Active, Scheduled, Conditioned and Cancelled. To inspect the state value the service '`Status`' will be used.

Events mark the transition from one period to another period for one object. Whenever the first future moment of transition (the next event) is known this moment in time can be inspected by the service '`Eventtime`' of the object. According the defined state values the next event is only known when the process state of an object is 'Scheduled'.

The next property concerns set membership. The use of sets will be discussed in detail in par. 5.5.4. Looking at services required from an object according to sets the next collection of services is generally required:

- to enter a `Set` and take a certain position. Usually an object joins a `Set` at the tail, sometimes objects are sorted to some criterion. To enter a `Set` a reference to the `Set` is required and a position selection mechanism. The service is therefore defined by '`Enter`(*set reference, position*)'.
- To leave a `Set`: `Leave`(*set reference*).
- To approach the direct neighbors in a `Set` of a `Simulation_Element` two services are provided: `Successor`(*set reference*) and `Predecessor`(*set reference*).
- At the moment in time a `Simulation_Element` enters a `Set`, this moment in time (on the simulation clock) is registered. This moment can be retrieved by the '`Entrancetime`(*set reference*)' service.

- To verify whether an element is present in a `Set` the service `IsInSet`(*set reference*) is required.

Finally the mechanism of the Receive property must be defined. This property was introduced to deal with communication between `Simulation_Elements`. The specific contents of a message (and thus the way in which Receive interprets a message) are not generally defined. In this thesis some kind of agreement beforehand is assumed between modelers on the meaning of the contents of messages. A general message in this thesis consists of three parts at least: the sender (source) , the receiver (destination) and the content. A new class '`Message`' is now introduced with these attributes. It will be formally defined at the end of paragraph 5.5.6, where the `Simulation_World` is discussed.

The delivery of a message requires a unique destination. `Simulation_Elements` are part of a `Model` that is part of the `Simulation_World`. Within the `Simulation_World` a destination is uniquely defined by the combination `Model` and `Simulation_Element`. However, a function doesn't "know" how it is implemented at a lower aggregation stratum (the function is a black box), so the corresponding `Simulation_Element` is unknown. This reflects the way in which a design process adds granularity during the project. Many different structures may be possible. Only the model as a whole at the lower stratum will "know" the objects that are part of that stratum. The destination therefore only specifies the destination model. Still to support a correct delivery, a '`Type`' service is added to the message class. The receiving model must be able to decide for the destination class based on the `Type` value of the message object. Types are defined "globally" as a result of agreement between modelers. The type value will be an input parameter for the `Create` service of a `Message`.

Delivering a message is a responsibility of the `Simulation_World` and the sending and receiving models together. If two objects belong to the same `Model` and send messages to each other, then the delivery can be completely handled within the `Model` itself. The `Send` service requires the object to specify a destination and to "hand the `Message` over" to the `Model` it belongs to. `Messages` with a destination outside the `Model` are "handed over" by the `Model` to the `Simulation_World` that handles this message further. "Handed over" is interpreted as 'calling a service'. For this reason a `Simulation_Element` doesn't need a

Send service of its own, but both Model and Simulation_World do have a Send (and Receive) service.

Inheritance and polymorphism

All classes derived from Simulation_Element inherit the attributes and services defined so far. The services Process and Receive however have no contents by default. The generic Process service shows no time dependency at all and the generic 'Receive' service doesn't know how to handle messages. These services must be specified for the derived classes and are thereby polymorphisms. This approach fits very well to the distinction between functions and flow elements as discussed in par. 5.4. If a Simulation_Element shows no time dependent behavior, then no specification of the Process service is required. If a Simulation_Element does not communicate with other aggregation strata or does not act in a distributed environment there's no need to specify a Receive service. Both Process and Receive services are only *virtually* defined with the class Simulation_Element.

The complete class definition of Simulation_Element is summarized below:

Class Simulation_Element	
Service	*Return value*
Create(*name*)	Reference to object
Destroy	-
Name	Unique object name
Arrivaltime	Simulation time of creation
Process {virtual}	-
Status	Static, Current, Scheduled, Standby, Cancelled
Eventtime	First future event of object
Start(*T*)	-
Stop	-
Cancel	-
Resume	-

`Proceed`	-
`Enter`(*set reference, position*)	-
`Leave`(*set reference*)	-
`Successor`(*set reference*)	Reference to next object in set
`Predecessor`(*set reference*)	Reference to preceding object in set
`IsInSet`(*set reference*)	TRUE if object in set, FALSE if not
`EntranceTime`(*set reference*)	Simulation time of entrance into the set
`Receive`(*message reference*) {virtual}	-

Table 5.3. The class `Simulation_Element`

5.5.4. The class `Set`

Sets are being used to represent general collections of objects, waiting rooms, queues, stocks etc. In general two functions are fulfilled by the set concept:

1. Control the collection
2. Register statistics about the set contents

The set concept in this thesis is more than the pure mathematical set concept.

Collection control is achieved by services analogous to the set services of a `Simulation_Element`, but now expressed from the viewpoint of the `Set` itself. For example the `Set` service `Add` (*Simulation_Element reference, position*) is analogous to the service `Enter`(*set reference, position*) of the `Simulation_Element` class. Collection control also includes `Remove`, `Successor`, `Predecessor` and `Contains`.
To retrieve all objects in a `Set` one usually asks for the object in front or the object at the tail of the `Set` and then repeatedly asks for the `Successor` or `Predecessor` respectively. The services `First` and `Last` are added for this purpose. In this way the set concept enables the modeler to access large numbers of object references, without the explicit definition of reference variables.

Statistics concern the number of elements and the length of stay of the elements in the `Set`. In this way the set concept supports the generation of output on required dimensions (number of elements) and throughput times (length of stay). To control the measurement period, the service '`ResetStatistics`' enables the modeler to start registration at a suitable moment in time. All statistics are cleared by this service.

The minimum required services are summarized in table 5.4.

Class `Set`	
Service	*Return value*
`Create`(*name*)	Reference to set object
`Destroy`	-
`Name`	Unique set object name
`Successor`(*object reference*)	Reference to next object
`Predecessor`(*object reference*)	Reference to preceding object
`Contains`(*object reference*)	TRUE if object in set, FALSE if not
`First`	Reference to heading object
`Last`	Reference to the tailing object
`Length`	Present number of objects
`NumberPassed`	Number of objects that passed the set
`MeanLength`	Average number of objects
`MininumLength`	Minimum number of objects
`MaximumLength`	Maximum number of objects
`MeanStay`	Average length of stay
`MinimumStay`	Minimum length of stay
`MaximumStay`	Maximum length of stay
`ResetStatistics`	-

Table 5.4. The class `Set`

5.5.5. The class `Model`

To collect objects and check the presence of objects, the `Model` class has an attribute representing an object of class `Set` called the `Elementroom`. Each `Simulation_Element` is stored in the `Elementroom` at the moment of creation and removed from it at the moment that it's `Destroy` method is called.

Three services of the `Model` class were already encountered in the preceding paragraphs: `Initialize`, `Send` and `Receive`.

The `Initialize` service creates the initial state of the `Model`, by creating objects of class `Simulation_Element` and `Set`, and starting the required `Process` services. The contents of `Initialize` are specific for each model and therefore this service is a virtual service.

The `Send` service checks whether the receiver is contained in the `Model` itself. In this case the `Model` calls its own `Receive` service to address the object concerned. Otherwise it calls the `Receive` service of the `Simulation_World`.

The `Receive` service checks the presence of the receiver object in the `Elementroom`. For each message `Type` value a receiving class within the `Model` should be defined. If the class is present it calls the `Receive` service of this class, which interprets the contents.

Class `Model`	
Service	*Return value*
`Create`(*name*)	Reference to model object
`Destroy`	-
`Name`	Unique model object name
`Send`(*message reference*)	-
`Receive`(*message reference*)	-
`Initialize`	-

Table 5.5. The class Model

5.5.6. The class `Simulation_World`

During a simulation run there is only one object of the class `Simulation_World` defined. This object takes care of the simulation clock based on '`Advance`' requests from the `Simulation_Elements` in the different `Models`. An `Advance` request results in one of the following (future) events:

- a time event in case of an `Advance`(*T*)
- a condition event in case of an `Advance`(*until/while condition*).

Time events are sorted on a time axis in increasing order of simulation clock time. During the period between '`Now`' (the current simulation time) and the first time event there are no state changes in the system. The simulation clock can be advanced to the time of this first event if the moment in time of this event lies in the future. All events with the same moment in time can be handled (resulting in new future events etc.) . As soon as the simulation clock is to be advanced all condition events are evaluated. If a condition is satisfied then this results in a time event at this moment in time.

The `Simulation_World` has to know which `Model` objects participate in a simulation run in order to deal with `Send` and `Receive` requests. For this purpose this class offers an `Announce` service. `Announce` registers the `Model` and calls its `Initialize` service.

A simulation run can start if all models are announced that are required at the start of the run. Starting a run is considered the responsibility of the modeler and is achieved by the '`StartRun`' service of the `Simulation_World`. After `StartRun` the simulation clock is running. Each `Model` is assumed to work with the same time unit.

Class `Simulation_World`	
Service	*Return Value*
`Create`(*name*)	Reference to object
`Destroy`	-
`Name`	Name
`Advance`(..)	-
`Now`	Current simulation time

Send(*message reference*)	
Receive(*message reference*)	
Announce(*model name*)	
StartRun	

Tabel 5.6. The class Simulation World

Finally the message class is defined in the next table.

Class C_Message	
Services	*Return value*
Create(*Type,Src,Dest,Contents*)	Reference to message object
Destroy	-
Type	Type of message
Source	Reference to source
Destination	Reference to destination
Contents	Contents string
SetContents(*string*)	-

Table 5.7. The class Message

5.6. The definition of a Process Description Language

5.6.1. Introduction

The previous paragraph introduced five class definitions to be used for the simulation of logistic systems: Simulation_Element, Set, Message, Model and Simulation_World. These are all "primitive" classes, no hierarchic relations have been defined. Hierarchic relations have to be defined by the modeler by using these classes as primitives for modeling functions, flow elements and sets. A function with a process property must be defined as a class derived from the Simulation_Element class and "override" the default Process service. If the modeler wants a set to represent "volume" instead of

"number of elements" a derived class of `Set` is to be defined with a service to calculate volume derived from the number of elements etc.

The classes offer services that cover all actions needed for simulation time and set handling. During event handling more linguistic constructs are needed to express decision making and condition handling. These constructs can not be formalized, because all details of decision making are part of the design process itself and will continuously change or extend. Some constructs for the formulation of decision making however are generic and can be found in any (programming) language. These constructs will be introduced in par. 5.6.2.

A description of a model was found to consist of three sections:

- a definition section of the derived classes
- a process section containing all `Process` services
- a control section representing the `Initialize` service.

These parts are sufficient to describe single model simulations.
To describe multi model simulations an inter model communication section is required also.
Paragraph 5.6.3. will describe the use of the PDL in a single model simulation, paragraph 5.6.4. will do the same for the multi model case.

5.6.2. Language elements

Some operations are generic whatever programming language is being used.
The syntax to be used in PDL is kept as simple as possible. A few syntactic rules are recommended for readability purposes:

1. Sections start with a bold faced section header:
    - **`Definition section`**
    - **`Process section`**
    - **`Control section`**
    - **`Communication section`**
2. Indentation is used to group lines into logical blocks
3. Process descriptions start with "`Description of Process of` *class*"

4. The owner of attributes or services is denoted by "of" or by a dot. For example `Process of Painting` or `Painting.Process` .
5. Attributes or services referred to within the `Process` description without owner specification are assumed to belong to the owner of the `Process`.
6. References and variables are composed of one single string. For example 'Machine 1' is not allowed, 'Machine_1' is.
7. A derived class is defined by `newClass`: (parent `Class`), followed by its specific or overriding attributes and services. For example a Painting function is derived from `Simulation_Element` overriding the `Process` service and the definition is:

```
Painting: (Simulation_Element)
  WorkInProgress: Set
  Service       : Process
```

   New primitive classes are defined by: `newClass: Class`, followed by its attributes and services.
8. To represent stochastic items a class "`C_Distribution`" can be used. No formal rules are established for defining or sampling from a distribution.
9. The assignment operator is denoted by '='. For example the line
   `ThisJob = First of Painting.WorkInProgress`
   Is interpreted as "the reference variable `ThisJob` gets the value of the reference to the first object in the set `WorkInProgress of Painting`".
10. The usual precedence rules apply on mathematical operators.
11. For conditional sequences the following statements can be used:

    `If` condition
    "do something"
    `Else`
    "do something else"

    `Loop while` condition
    "do something"

    `Loop until` condition
    "do something"

"`Until` condition" is the same as "`While NOT`(condition)".

12. Repetition is denoted by "`Loop`" or "`Loop`(n)", where n denotes the number of repetitions for the succeeding block. "`Loop`" without a number indication means indefinite repetition.
13. If clauses checking for many different values of the same variable can be formulated by using "`Case` variable `is`". For example if each value of a dice asks for a different action, this is formulated as:

```
Case Dice_value is
1: …
2: …
3: …
4: …
5: …
6: …
```

14. Comments are specified italic

In a PDL description the `Simulation_World` and the `Model` objects are assumed to be present by default; no definition is required. The `Model` object has to be described only in the header of the PDL description by a bold faced "**`Model of`** *short description of the (sub)system*".

There are no further rules defined. It is good practice however to add comments to the lines in the Definition section to explain the purpose of newly defined attributes and services. In normal language objects are said to be of type "*classname*". Therefore it is recommended here to start class names with "C_". By using this convention it is also made clear that the definition section contains type definitions. The classes "`Simulation_Element`" and "`Set`" will be the exceptions to this recommendation.
References pointing to nothing (there's no object associated with it) have a '`Nil`' value.
There is no type definition for numeric variables. They can be defined by simply stating they represent a 'value'.

5.6.3. A single model PDL example

In this example the description of table 4.2. is described in terms of PDL. It is shortly repeated here.

An operational function transforms semi-finished products from an inventory into finished products. Products are selected from an inventory in a first-in-first-out order; each product transformation has a certain execution time. If there are no products available the function waits for a new product to arrive in the inventory. The function needs a task to start the transformation. Above that the resource must be shared with other operations, so the resource must be explicitly assigned to this operation. It is assumed each order consists of one task only. A task is added to a task list of the operation and selected in a FIFO-order.

```
Model of the operation function of example 4.3.


Definition Section
C_Operation: (Simulation_Element)
      Inventory: Set          input products
      TaskList : Set          tasks to do
      Resource : C_Resource   refers to assigned resource
      Task     : C_Task       refers to task in progress
      Product  : C_Product    refers to product in progress
      Service  : Process
C_Perform: (Simulation_Element)
      OrderList: Set          orders to do
      Order    : C_Order      order selected
      NewTask  : C_Task       task of order selected
      ExecutionTimes: C_Distribution
      Service  : Process
C_Use: (Simulation_Element)
      RequestList: Set        requests for a resource
      IdlesList  : Set        idle resources
      BusyList   : Set        assigned resources
      Resource   : C_Resource Resource selected
      Operation  : C_Operation Operation selected
      Service    : Process
C_ProductGenerator: (Simulation_Element)
```

```
      ArrivalRate: C_Distribution  product inter arrival times
      Service    : Process
C_OrderGenerator: (Simulation_Element)
      ArrivalRate: C_Distribution  order inter arrival times Service
      : Process
C_Task: (Simulation_Element)
      ExecutionTime: value
C_Product: (Simulation_Element)
C_Resource: (Simulation_Element)
C_Order: (Simulation_Element)

Operation       : C_Operation           object references
Perform         : C_Perform
Use             : C_Use
OrderGenerator  : C_OrderGenerator
ProductGenerator: C_ProductGenerator

Process Section
Service: Process of C_Operation
  Loop
      Advance(While TaskList.Length = 0)
      Advance(While Inventory.Length = 0)
      Enter(RequestList of Use)
      Advance(While Resource = Nil)
      Task = First of TaskList
      TaskList.Remove(Task)
      Product = First of Inventory
      Inventory.Remove(Product)
      Advance(Executiontime of Task)
      Task.Destroy
      Product.Destroy
      Use.BusyList.Remove(Resource)
      Use.IdleList.Add(Resource,ToTail)
      Resource = Nil

Service: Process of C_Perform
  Loop
      Advance(While OrderList.Length = 0)
      Order = First of OrderList
      OrderList.Remove(Order)
```

```
      NewTask = TTask.Create('Task')
      NewTask.ExecutionTime = Sample of ExecutionTimes
      Operation.TaskList.Add(NewTask,ToTail)
      Order.Destroy

Service: Process of C_Use
  IdlesList.Add(TResource.Create('Resource'),ToTail)
  Loop
      Advance(While RequestList.Length = 0)
      Advance(While IdlesList.Length = 0)
      Resource = First of IdlesList
      IdlesList.Remove(Resource)
      BusyList.Add(Resource,ToTail)
      Operation = First of RequestList
      Operation.Resource = Resource

Service: Process of C_ProductGenerator
  Loop
    Operation.Inventory.Add(TProduct.Create('Product'),ToTail))
    Advance(Sample of ArrivalRate)

Service: Process of C_OrderGenerator
  Loop
     Perform.OrderList.Add(TOrder.Create('Order'),ToTail))
     Advance(Sample of ArrivalRate)

Control Section
Operation = C_Operation.Create('Operation')
Operation.Start(Now)
Perform = C_Perform.Create('Perform')
Perform.Start(Now)
Use = C_Use.Create('Use')
Use.Start(Now)
OrderGenerator = C_OrderGenerator.Create('GenerateOrders')
OrderGenerator.Start(Now)
ProductGenerator = C_ProductGenerator.Create('GenerateProducts')
ProductGenerator.Start(Now)
StartRun
```

The PDL description still shows a large resemblance to the description in chapter 4. It has been formalized further and from this description the modeler can raise questions about the model to be simulated for example:

- what are the arrival rates of orders?
- What are the arrival rates of products?
- There is now only one resource involved. Will there be more resources in future?
- Will there be more Operation functions in future?

Even before the first line is coded in a software environment questions can be formulated that influence the structure and contents of the model. The model can easily be adapted here for example to deal with more than one resource and operation. Then the assignment of resources to operations becomes more complex and raises new questions etc.
By using PDL one is in fact 'simulating' the construction of the simulation model.

5.6.4. A multi model PDL example

To illustrate the communication between models the example of table 4.4. will be used. The example concerns two models, each at its own aggregation stratum. At the highest stratum a painting function is defined and at the lower stratum this function is split into a separate cleaning and coloring function. First the high stratum model is described in PDL.

```
Model of painting function at high aggregation stratum

Definition section
C_Painting: (Simulation_Element)
  Inventory : Set          products to be painted
  NewMessage: C_Message     message to be sent
  Service   : Process
  Service   : Receive(Message)

C_ProductGenerator: (Simulation_Element)
  ArrivalRate: C_Distribution  product inter arrival times
  NewProduct : C_Product       product being generated
  Service    : Process
```

```
C_Product: (Simulation_Element)

Painting        : C_Painting          object references
ProductGenerator: C_ProductGenerator

Process Section
Service: Process of C_Painting
  Loop
      Advance(While Inventory.Length = 0)
      Advance

Service: Receive(Message) of C_Painting
  Product = Message.Contents as C_Product
  Painting.Inventory.Remove(Product)
  Product.Destroy
  Painting.Resume

Service: Process of C_ProductGenerator
  Loop
      NewProduct = C_Product.Create('Product')
      Painting.Inventory.Add(NewProduct,ToTail)
      NewMessage = C_Message.Create(producttype,'HighStratum',
        'LowStratum',NewProduct.Name)
      Send(NewMessage)
      Advance(Sample of ArrivalRate)

Control Section
Painting = C_Painting.Create('Painting')
Painting.Start(Now)
ProductGenerator = C_ProductGenerator.Create('GenerateProducts')
ProductGenerator.Start(Now)
Announce('HighStratum')
```

The model at the lower stratum is shown below.

```
Model of painting function at lower aggregation stratum


Definition section
C_Cleaning: (Simulation_Element)
```

```
      Inventory: Set         products to be cleaned
      Product  : C_Product   product being cleaned
      Service  : Process
      Service  : Receive(Message)

C_Coloring: (Simulation_Element)
      Product   : C_Product   product being colored
      LastColor : value       color of last product colored
      NewMessage: C_Message   message to be sent
      Service   : Process

C_Product: (Simulation_Element)
      Color: value

Cleaning: C_Cleaning            object references
Coloring: C_Coloring
Colors: C_Distribution

Process section
Service: Process of C_Cleaning
  Loop
      Advance(While Inventory.Length = 0)
      Product = First of Inventory
      Inventory.Remove(Product)
      If Color of Product ≠ LastColor
            Advance(2)
      LastColor = Color of Product
      Coloring.Start(Now)
      Advance

Service: Receive(Message) of Cleaning
  Product = Message.Contents
  Color of Product = Sample of Colors
  Cleaning.Inventory.Add(Product,ToTail)

Service: Process of Coloring
  Advance(5)
  NewMessage = C_Message.Create(producttype, 'LowStratum',
     'HighStratum', Name of Product of Coloring)
  Send(NewMessage)        Send product to stratum N
```

```
  Product.Destroy
  Cleaning.Resume
  Stop

Control Section
Cleaning = C_Cleaning.Create('Cleaning')
Cleaning.Start(Now)
Coloring=C_Coloring.Create('Coloring')
Coloring.Start(Now)
Announce('LowStratum')
```

Both models announced themselves to the Simulation_World with a name to be used for communication purposes. The send service of the high stratum model doesn't know anything of the way the painting function is fulfilled at the lower stratum. The only destination known is the model of the lower stratum. By defining the type of the message to be 'product' the lower stratum model is capable of deciding where the message can be interpreted, in this case by cleaning. This is shown in the communication section.

```
Communication section
Service: Receive(Message) of Model 'HighStratum'
  If Message.Type = producttype
      Product = Find Simulation_Element in ElementRoom
                with Name = Message.Contents
      If Product ≠ Nil
            C_Painting.Receive(Message)
  Message.Destroy

Service: Receive(Message) of Model 'LowStratum'
  If Message.Type = producttype
      NewProduct = C_Product.Create(Message.Contents)
      Message.SetContents(NewProduct.Name)
      Cleaning.Receive(Message)
  Message.Destroy
```

5.7. Conclusions

The PDL was defined to connect the informal descriptions of time dependent behavior to an informal system methodology for the design of software systems. The PDL definition is mainly object oriented , based on the definitions of Booch [1994].

It was shown that the description of a simulation model by means of PDL supports the modeler in the preparation of software implementation and reveals the open ends in the quantification and/or the structure of the model as defined by the problem owners.

Five primitive classes to be used in simulation were defined; from these classes other classes can be derived that incorporate the necessary simulation services. These classes will now be implemented in a software environment to simulate the time dependent behavior of the functions in a PROPER model.

Chapter 6

TOMAS: an implementation of PDL

Inside every well-written large program is a well-written small program.

- C.A.R. Hoare -

6.1. Introduction

Simulation, as described in this thesis, is the use of a computer model of a system to study its behavior as it operates over time. Discrete event simulation deals specifically with modeling of systems in which the system state changes instantaneously at discrete moments in time, rather than continuously. Logistic systems are considered discrete systems. In this chapter programming language concepts and features are developed to represent descriptions of such systems using the Process Description Language (PDL) of chapter 5.

Zeigler [1976] formulated a discrete event system specification (DEVS) formalism, to be used in a single processor approach. In Zeigler[2000] the formalism is extended and covers distributed simulation as well.

Many simulation software packages evolved from DEVS. The packages show large differences in the way the formalism is implemented. In this thesis, an implementation is preferred that closely matches to PDL and serves the purpose of decision support during the design of logistic systems.

First a survey of the DEVS formalism is given. After that, the requirements will be formulated for a software implementation that fits to PDL. Then an inventory will be made of the existing simulation software focusing the DEVS implementation. As a conclusion, there is no software available meeting all requirements and therefore a new implementation is proposed. This implementation is called Tool for Object oriented Modeling And Simulation (TOMAS) and is the subject of the rest of this chapter.

6.2. The DEVS formalism

Zeigler [2000] distinguishes three system formalisms for simulation (see table 6.1):

- DESS: Differential Equation System Specification
- DTSS: Discrete Time System Specification
- DEVS: Discrete Event System Specification

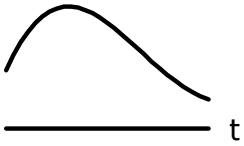
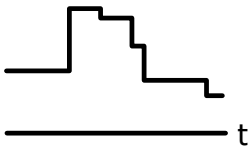
Formalism	System	Model	Simulator
DESS		dq/dt = f(q(t), x(t), t)	Numerical integrator
DTSS		$q_{t+1} = f(q_t , x_t)$	Recursive algorithm
DEVS		Time to next event, State at next event,	Event processor

Table 6.1. Simulation formalisms (from Zeigler[2000])
(q represents "state")

The DESS formalism specifies a system by means of differential equations and is used for continuous system simulation. The DTSS formalism is used for systems operating on a discrete time base with a constant time advance. DEVS is used for discrete systems with a variable time advance, concentrating on interesting events only. Zeigler considers DTSS a subclass of DEVS, because the periods between events in DEVS are completely flexible. The main difference between DESS and DEVS is the representation of time. Where DESS uses a continuous representation, DEVS considers time a discrete set of moments where the state of the system changes instantaneously. These instantaneous state changes are called events.

For the simulation of logistic systems the pure DEVS formalism will be used. Time is conceived as flowing along independently; all changes are ordered by this flow.

A DEVS is a structure

$$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

where

X is the set of inputs

S is a set of sequential states

Y is the set of outputs

δ_{int}: $S \rightarrow S$ is the internal state transition function

δ_{ext}: $Q \times X \rightarrow S$ is the external state transition function, where

$Q = \{(s,e) \mid s \in S, 0 \leq e \leq P(s)\}$ is the set of total states

e is the time elapsed since last transition

λ: $S \rightarrow Y$ is the output function

P: $S \rightarrow R^+$ is the time advance function, where R^+ is the set of positive reals including 0 and ∞

At any time the system is in some state s. If no external event occurs, the system will stay in state s for period $P(s)$. If $P(s) = 0$ the stay in state s is so short that no external events can intervene. If $P(s) = \infty$ the system will stay in state s forever unless an external event interrupts it. When the resting time expires i.e. when the elapsed time $e=P(s)$, the system outputs the value $\lambda(s)$ and changes to state $\delta_{int}(s)$. If an external event $x \in X$ occurs before the expiration time (i.e. the system is in total state (s,e) with $e \leq P(s)$), the system changes to state $\delta_{ext}(x,s,e)$. Now the system is in some state s again.

Reading 'function' for 'system' shows that a DEVS can be described for each function. A function structure is therefore simulated by interrelated DEVS all using the same time axis. This is called a multicomponent DEVS. A simple example of a generator and a processor is shown in appendix B.

The state of a system is defined by the current property values and the period during which it stays unchanged (see chapter 4). According the DEVS definition this period can always be expressed by some moment in time, because the time advance function always results in an event at a moment in time: a so-called "time event". In chapter 5 however conditional events were introduced where an element waits for some condition to occur. These events are called "state events" and there is no moment in time attached to such an event. In terms of DEVS, if the condition occurs the state event should be changed into a time event. In this thesis the transition of state event to time event is considered a responsibility of the time advance function of the element waiting for the condition to occur. This function must be

able not only to accept period values expressed as time durations, but also period values expressed as conditions.

According the DEVS definition the time dependent behavior is not uniquely defined when both an internal event and an external event occur at the same time. For good definition an order of processing has to be defined. The resolution is handled in "classical" DEVS by introducing a *Select* function, which will allow only one element to be activated at any moment, thus assuring that the collision can't happen. "Parallel" DEVS deals directly with collisions. In this thesis the resolution of classical DEVS is used, because real parallelism can't be achieved in a single processor environment. The Select function serves in fact two goals:

- It prevents collision of events for one single element
- It decides on order of execution where events of two or more elements collide.

In par. 6.6 this item will be addressed further for cases where prototyping using the real time clock is introduced.

There are three world views with respect to DEVS:

- *event scheduling: identifying when actions are to occur.*
  Event oriented models preschedule all events. There is no provision for conditional events. The moment-in-time approach in chapter 4 (tables 4.1. and 4.2) illustrates the event scheduling world view. An event scheduling algorithm employs a list of events that are ordered by increasing scheduling times. The event with the earliest scheduled time is removed from the list and the simulation clock is advanced to the time of this event. The routine associated with the event is executed. Execution of the routine may cause new events to be sorted in the event list. Existing events in the list may be rescheduled and cancelled. Because of its simplicity event scheduling is the most used strategy for implementing DEVS.

- *activity scanning: identifying why actions are to occur*
  With the activity scanning world view, a modeler concentrates on the activities of a system and the conditions that allow an activity to begin.

At each clock advance, the conditions for each activity are checked by the event scheduling mechanism and if the conditions are true, the corresponding activity begins (which means that the finishing event also is being scheduled). Activity scanning does not use periods explicitly but state events. In its purest form activity scanning implies the clock value to be part of the condition too. A time event to occur at time *t* can only be scheduled by a condition "*if time = t*". In this pure form the DEVS formalism turns into a DTSS formalism where the clock advances with fixed time increments. Contrary to this: if conditions are only influenced by time events, they only have to be checked at event times.
The main advantage of this world view is that it is much simpler to think about and formalize. The main disadvantage with this world view was computational inefficiency until the increased speed of processors outweighed this disadvantage.

- *process interaction: identifying the elements and describe the sequence of actions of each one.*
  In the process interaction world view, the simulation is considered a collection of interactions among processes. Processes interact with each other by services. This world view focuses on a *sequence* of events and/or activities that are logically connected. There are two different views to connect events and activities. Each view corresponds to a different assumption as to what are the active and passive elements in building a model. In the first view the active elements are taken to be the elements that do the processing, in the second view the active elements are the flow elements. The first one is called pure "process interaction" and the second one the "transaction" world view. Tables 4.1. and 4.2 illustrate the use of a pure process interaction approach.
  From a modeler point of view the system looks quite different: events have no meaning. From a processor point of view the system is the same: events still have to be scheduled and an event scheduling mechanism is still required.

From the discussion so far it is concluded that a process interaction world view of DEVS is preferred here for simulating logistic systems. It is directly related to the proposed object oriented PDL description of chapter 5. It is also directly related to the static PROcess-PERformance model (PROPER-model) of chapter 2 if the pure process interaction approach is used: a function (or a resource doing the operational function) is taken to be an active element and owns a process.

6.3. Requirements for logistic simulation packages

The primary goal of simulation in this thesis is to offer decision support during the design of a logistic system. Chapters 4 and 5 already derived how the time dependent behavior of such a system should be described. A simulation platform must fit to these descriptions and from this the first requirements are:

1. the platform has to support descriptions of the time dependent behavior of 'functions' according the pure process interaction world view. Other DEVS world views don't agree with PDL.
2. the platform should support object oriented computing (as defined by PDL)
3. it has to support both time events and state events
4. it has to support a powerful language to specify control systems. Most design projects develop new control algorithms, which require powerful linguistic descriptions. Visual representation and reuse of former control algorithms usually don't satisfy.

During the design project many models may be created. The simulation effort can easily grow beyond the capacity of one single modeler. Therefore the requirement is added:

5. concurrent modeling has to be supported. The package should enable both stand alone and networked (or distributed) simulation. In order to be able to use a simulation package in a distributed environment the event scheduling mechanism must notify the environment at the moment that it is about to advance the simulation clock for synchronization purposes. This type of scheduling mechanism is called here 'open scheduling'.

Furthermore, some parts of an already simulated system may be studied in more detail. To preserve the decision results of the higher aggregation strata it should be possible to embed detailed models in (parts of) global models. This leads to the requirement that:

6. reuse of models and/or elements should be supported.

At the end of a design project real equipment and control will be tested. In this context two further requirements are formulated:

7. the package should support 'prototyping' i.e. connecting real equipment to a simulated environment and synchronizing this environment to the real time clock.
8. It should be possible to reuse once designed control software for simulation purposes into the real control software. There is even a tendency to use simulation models as the mechanism of decision making itself ([Ottjes, 1996]). It should therefore be possible to connect the simulation package to operational software environments.

Finally standard simulation requirements concern:

9. Reproducibility; if a simulation run is repeated with the same initial state it should deliver the same results.
10. Speed; a high speed simulation package is preferred.

The reproducibility and speed requirements seem quite trivial. They are however the main reasons why two implementation alternatives will not be considered further.

First of all real parallelism is excluded. As mentioned before the collision of events is solved by allowing only one element to be activated at any moment. This serializing provides an unambiguous execution order of events. It will also be used as a principle in distributed simulation and is called the 'conservative' distributed approach. Only when real-time simulation is considered, real parallelism must be supported. In this case speed is not a real issue anymore. A DTSS approach will be used then (as discussed in par. 6.6).

Secondly implementations using 'external' mechanisms by default, will be excluded.
One of the most widely used approaches to implement object oriented process interaction between elements is using 'threads' to represent elements. A thread can best be considered a separate task in a multi-tasking environment (e.g. Windows®). It's the responsibility of the environment to divide processor time among the tasks running. This 'time slicing' mechanism slows down the simulation speed dramatically, especially in models with a large number of elements. A single task implementation (for stand alone simulations) is therefore preferred.

6.4. Selection of the simulation package

The number of software packages for discrete event simulation is enormous.
Most commercial packages are directed mainly towards modeler support. Arguments as user friendliness and development speed are most used to recommend a package. Vast libraries of predefined elements emerged. During the last decades the development emphasized advanced visual model building, graphical and animated representation and connectivity to general purpose reporting environments.

The packages can be divided according the world view of DEVS used. The following table summarizes the world view of some common used packages.

Package	*World view*	*Implementation*
Automod	Process	Transaction
Arena	Process	Transaction
Simple++	Activity scanning	n.a.
Taylor II	Process	Transaction
Witness	Process	Transaction

Table 6.2. World views of common simulation packages.

None of these packages uses the process interaction world view.

Until recently these off-the-shelf packages paid little attention to the interface with the supporting simulation programming language. The consequence is that the implementation of control algorithms is complicated. The simulation of control systems requires a simulation programming language of high quality. Offering only standard control algorithms does not satisfy the requirements of innovative design. The same holds for the availability of standard elements (equipment, vehicles, resources). They may be important for redesign, but cover only partly the requirements of innovative design. The latter type of design requires the simulation of completely new ways of processing and combinations of functionality.

In the simulation world the emphasis is nowadays shifting towards "distributed" simulation. This increased attention for distributed simulation requires an open scheduling mechanism. Off-the-shelf packages hide their scheduling mechanism for the user and offer no direct possibilities to intervene. The only way to control clock time advance is making a user

element acting on some regular event basis. This element can only schedule time events (the condition "if time is about to advance" cannot be specified). Therefore the use of distributed simulation immediately leads to the change of a DEVS scheme to a DTSS scheme.

For the reasons above only the simulation *languages* will be considered further that are based on the process interaction approach and do not use threads by default. It restricts the research to five languages. Table 6.3. summarizes the characteristics of each of these languages and rates them with respect to the main criteria of par. 6.3.

	SimScript	PROSIM	Must	MODSIM	SIMULA
Supporting language	FORTRAN Native	Native	Pascal	C++	Algol Native
Object oriented	Yes	No	No	Yes	Yes
State events	No	Yes	Yes	Partly*	Partly*
Open event scheduling	No	No	No	No	No
Reusability	Yes	Yes	No	Yes	Yes
Reproducibility	Yes	Yes	Yes	Yes	Yes

Table 6.3. Process oriented simulation languages

* Partly means that conditional events can be specified but testing the conditions is a responsibility of the modeler.

Simscript [CACI,1997], PROSIM [de Gans, 1999] and SIMULA[Kirkerud,1989] are all 'native' languages, although Simscript and SIMULA were originally based on general programming languages. Having a native language compiler facilitates the use of special simulation constructs. It complicates however the use of general or non-simulation-specific modules and it becomes difficult to embed these type of models in other information system applications.

PROSIM and Must [van der Ham,1993] are not object oriented. They make use of component constructs, but these constructs don't combine data and services. Inheritance and polymorphism are not implemented. Above that Must has not been developed for Windows® environments.
MODSIM [CACI,1996] implements time dependent behavior in a process oriented way, but allows one object also to perform several processes at the same time. For the function approach used in this thesis this facility is not needed; two parallel processes will always belong to two different functions and one process description can contain as many events as

needed. According the DEVS scheme of Zeigler, two simultaneous internal events are even impossible.

Only Must and PROSIM support state events in a way described in chapter 5 completely. The way in which MODSIM and SIMULA implement state events, results in computational efficiency. Conditions are only tested at the moments that some parts of a condition may have changed. However, if these moments are known, state events are not needed at all. In general the use of state events must be minimized, if computational efficiency is one of the primary goals of a model. The implementation of PROSIM and Must requires the conditions to be tested after each event (or when the clock is about to advance). This is computationally inefficient, if there are many conditions pending or many clock advances. Only in cases where many situations occur that influence the state of a condition, it can be profitable from a computational point of view to use state events. This conclusion differs from the PDL and process descriptions in chapters 4 and 5, where conditional statements are preferred. There the main goal was not computational efficiency, but readability and communication.

None of the simulation languages supports intervention with the event scheduling mechanism.

It's the nature of object orientation that objects can be easily reused in different applications. PROSIM supports reusability by offering a kind of 'module organization' in its platform.

Reproducibility is a trivial requirement. Every discrete event simulation must be reproducible by default. All languages satisfy this requirement.

None of the languages completely fits the PDL definition and for this reason it was decided to develop a simulation language that completely conforms to PDL.
Instead of defining a new native language, it was decided from the very start to develop the language as an extension to a general programming language. In this way it becomes possible to use all of its facilities, to take advantage of on-going developments in the software community and to embed the simulation models in a general software application. At this moment there are four leading object oriented languages: Visual BASIC, Visual C++, Delphi and JAVA.

The first three are object oriented extensions of languages that exist already for a long time: BASIC, C and Pascal respectively. JAVA has the looks of C, but it is originally object oriented and it is the only platform independent language.

All languages offer sufficient facilities to develop (regular) simulation requirements like sets and communication. The primary goal is to show the feasibility of a pure process oriented DEVS including distributed simulation and to use this implementation for educational and research purposes. For educational purposes it is decided to use Delphi in this thesis; this software environment is based on Pascal, a well defined and structured language, which forces the (student) modeler to use object orientation and to be clear with respect to definitions. The here developed implementation however can also be developed in the other languages[1]. The implementation is called TOMAS: Tool for Object oriented Modeling And Simulation. For the same educational purpose it is decided to develop TOMAS as an open source application: the source code is available for anyone and is provided at a web site (www.tomasweb.com). This enables students to study the object oriented implementation and may mobilize users of the internet community to exchange experiences and propose additions.

6.5. Implementing the process interaction DEVS

6.5.1. The TOMAS application structure

A stand alone Delphi application (a "project") consists of two main parts:

- a programming part
- a visual part

The programming part is composed of one project control source file and one or more "units" containing source code, which describes the behavior of the application. The visual part is composed of a number of "forms", which are the 'windows' that appear to the user on the screen. A form is always connected to a unit that contains the programmed reaction for pressing buttons and editing text on the form by the user. A unit does not have to be connected to a form, it may consist of program code only. Units without a form usually

[1] The mechanism has already been developed in Visual C++ too and the first steps for an implementation in JAVA have started last january.

contain code that is used by all other units (common objects and common functions and procedures). The general structure of a Delphi application is shown in figure 6.1.

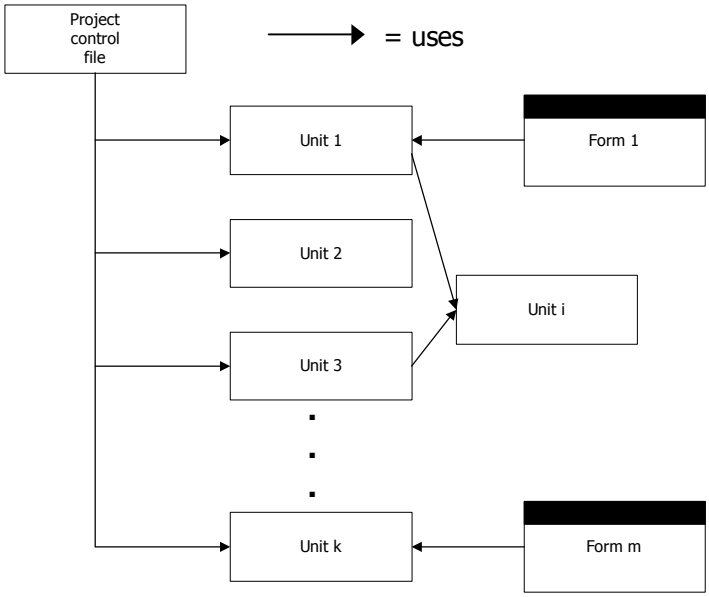


Figure 6.1: *General structure of a Delphi application*

A Delphi application always has one 'main' form that is visible at startup. An application will therefore be represented by a rectangle with dark bar at the top from now on.

A TOMAS application is a structure of models and programs in a simulation world [Veeke, Ottjes, 2001]. Each model and each program will be considered separate Delphi applications. The simulation world itself is a Delphi application too. Models have a notion of time duration and should be synchronized to the clock of the simulation world. Programs usually represent control algorithms that may act instantaneously. They may know the simulation clock time but do not take (simulation clock) time and therefore need not to be synchronized.

As shown in chapter 5 the services to be provided by the simulation world application are synchronization and communication between models. In cases where a model is running stand alone without any communication, the simulation world application would only slow down the simulation speed. Each model application therefore has its own simulation clock that can eventually be synchronized to the clock of the simulation world. The simulation world application will be called *TomasServer* from now on. A general TOMAS application has a client server structure as shown in figure 6.2. All inter model communication is directed via

TomasServer. No direct communication between the models will be supported in the general TOMAS language. This enables the run time construction of a simulation structure to be based on only one address that needs to be known beforehand: the address of the TomasServer. The TomasServer application will be described further in paragraph 6.6.

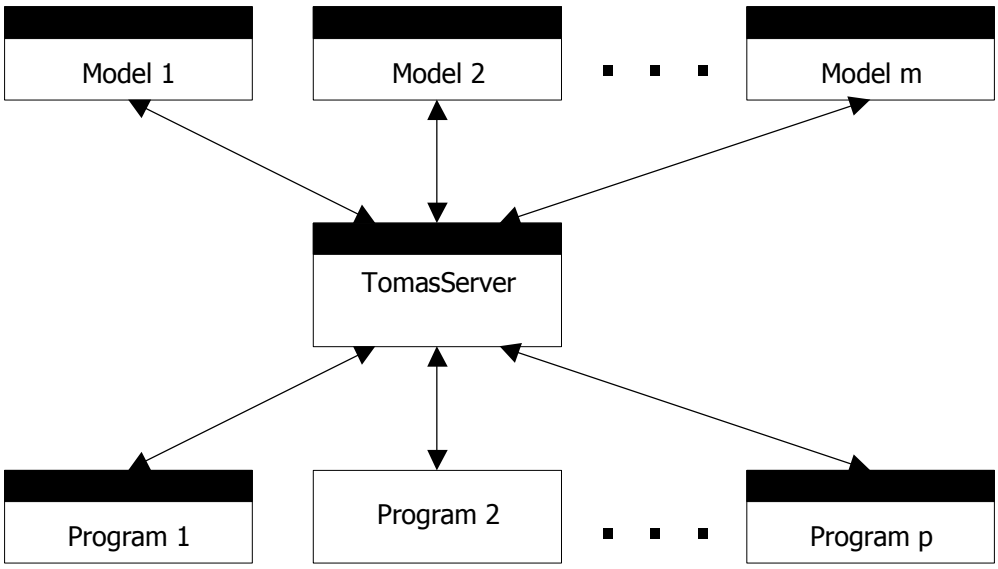


Figure 6.2: *Structure of a TOMAS application*

The core of a TOMAS stand alone model is the scheduling mechanism that is implemented in a separate unit *TomasProcess*. This core takes care of the simulation clock time, keeps track of the "simulation run status" and contains a "local" `Tomas_World` object as defined in table 5.6.

The `Model` class of table 5.5. is implemented in a surrounding unit with form called *Tomas*. The visual part supports the modeler with run control (start, interrupt etc.) and verification items as tracing the events, showing the contents of a model at some instant etc. The source code contains the class `Simulation_Element` and `Set`; they are called TomasElement and TomasQueue respectively.

Tomas and TomasProcess are units that can be used in a general Delphi application. They contain the basic tools needed. Usually a simulation model will create its own user unit with forms to represent user defined classes derived from TomasElement and TomasQueue.

The basic structure of a stand alone TOMAS model is represented in figure 6.3.

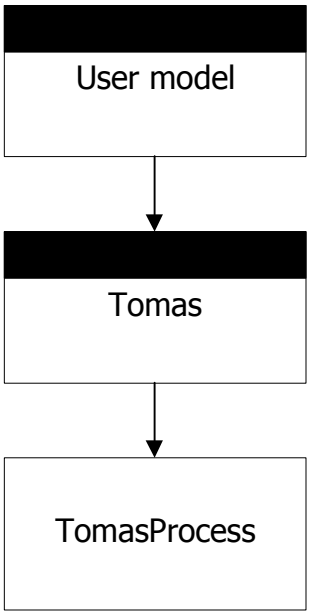


Figure 6.3: *The structure of a stand alone Tomas model*

The next paragraphs will explore the contents and backgrounds of each of these units. The full source code of TomasProcess can be found in appendix C. The other source codes that are part of the Tomas environment, can be found at the web site www.tomasweb.com.

6.5.2. The event scheduling mechanism of TomasProcess

TomasProcess represents the whole simulation world for a stand alone simulation run. To prevent collisions between simultaneous events the DEVS is implemented in such a way that one and only one TomasElement can handle an event ('be active') at any one moment. Simultaneous events are handled in a first come first served order.

A TomasElement handles an event means "executes its process code". The Advance service (of the simulation world) finishes the event handling and tells the scheduling mechanism that the simulation clock may advance with respect to this TomasElement. Advance also indicates to the scheduling mechanism when this TomasElement wants to proceed with the execution of its process code. It introduces:

- a new future time event if the Advance service called was: Advance(T)
- a new state event if the Advance service called was: Advance(while / until condition)
- nothing if the Advance service was called without any further specification.

TOMAS uses different expressions for these advance services:

- Advance(T) is implemented as : *Hold(T)*
- Advance(While / until condition) is implemented as: *While condition Do StandBy*.
- Advance is implemented as: *Suspend*.

In all cases the sequential execution is interrupted and should return to the next statement only when the simulation clock reaches the time of the time event, or when the condition is satisfied or when another element 'resumes' the process. In the mean time (with respect to the simulation clock) other elements may handle events (i.e. execute process code).
To explain this mechanism the example of table 5.2. is shown below in correct Delphi code.

Process of TomasElement1	**Process of TomasElement2**
Procedure TomasElement1.Process **Begin** Counter := 0; Hold(3); Counter := Counter + 1; Hold(2) Counter := Counter – 1; Finish; **End;**	**Procedure** TomasElement2.Process **Begin** Hold(2); Counter := Counter + 1; Hold(6); Counter := Counter – 1; Finish; **End;**

Table 6.4. Two simple process descriptions.

Both TomasElement1 and TomasElement2 are derived from the class TomasElement. In figure 6.4 the statements are projected to the axis of the simulation clock.

The process services are regular Delphi procedures. Execution of a Delphi program can be represented as a sequence of procedure calls, repeatedly entering a procedure and after code execution returning to the statement directly following the calling statement. Delphi keeps record of this sequence by using an instruction pointer (IP) and a stack mechanism.

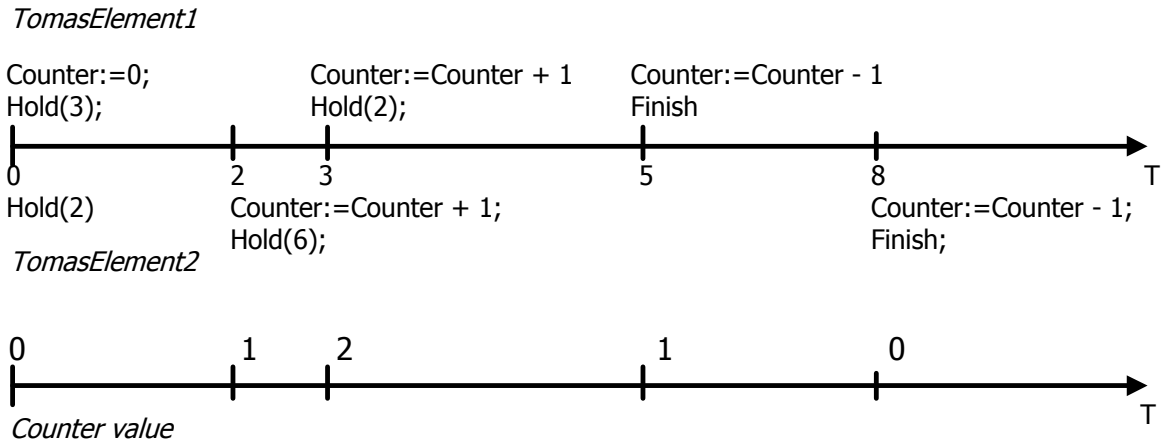


Figure 6.4: *Statement execution and simulation clock*

At the moment a statement is being executed the IP points to the succeeding statement. If the statement in execution represents a procedure call, the current IP is pushed on to the stack. Each assembly coded procedure finishes with a 'return' statement that pops this IP value from the stack and jumps to it, continuing in this way the execution of the calling procedure.
The same mechanism is used in the procedure calls of Hold, StandBy and Suspend. These are calls to procedures in the scheduling mechanism. This mechanism can break up the normal sequence now by popping the return IP address from the stack prematurely, saving it as an attribute of the calling element and replacing it with the return IP address of another element. Jumping to this IP address reenters the process procedure of the other element at the statement following Hold, StandBy or Suspend.

This is the basic idea for breaking up the sequential execution. There are some other factors that complicate the implementation of this idea; within a procedure the assembly registers are assumed to reflect the object to which the procedure belongs (the service owner) and there may be "local" variables present on the stack already. These factors are well-documented and are therefore easily solved by pushing the register values on to the stack also and store them together with the return IP address as an attribute of the element. Dealing with the stack and filling registers should be coded in assembly language. Delphi enables the insertion of assembly statements in the high level Pascal source code.

In a stand alone model there is only one processor and thus only one code line at a time can be executed. As a consequence there is always only one element that can call Hold, Standby or Suspend namely the "current element". In order to keep the point of progress and local

data of this current element, TomasProcess requires the notion of a TomasElement object. TomasElement however is only known in the higher level unit of Tomas. In order to preserve the hierarchic structure of figure 6.3. it is decided to implement a parent class of TomasElement in TomasProcess. This class is called "TomasProcessElement" and it owns all internal services related to the simulation clock; it can schedule or remove itself from the time axis. TomasElement as defined in Tomas is now a derived class from TomasProcessElement and adds the set services and external process interaction services.

TomasProcessElements (including their stack contents) are collected by the scheduling mechanism into a time event list called "SequentialMechanism" and ordered by increasing event time first and by time of arrival if event times are equal. If the current element calls the scheduling mechanism by means of Suspend, only the stack contents and return address are saved, but the element is not added to the time event list. If the Standby service is called, the current element is collected into a state event list called "StandbyMechanism". After the handling of each time event, all state events are checked by jumping to the code of each element in the state event list. The Standby call is a part of a "While condition Do" clause. So the next statement to be executed after return from a call to Standby is always this clause. If the condition is false then Standby is called again and the element reenters the state event list. If the condition is true the statement after this clause will be executed as a normal time event until another Hold, Standby or Suspend is encountered.

As mentioned in paragraph 5.5.3. three points in a process of a TomasElement should be identifiable:

- the starting point of a process
- the current period
- the point of resumption at the end of a period.

The starting point is simply the address of the process service itself. This address is always known as part of the element's definition. The point of resumption is represented by the saved IP-address. The current period is also known, because the element is present in the time event list or in the state event list. To be completely unambiguous only the "state" of the element must be kept when another element cancels the period. If the cancelled element is in the time event list, the rest duration can be calculated and saved. This duration

will be considered the actual rest duration again at the moment the element is ordered to proceed.

Finally, the scheduling mechanism should be open to support distributed simulation. In TOMAS the opening is offered by means of a procedure call from the scheduling mechanism to some external procedure. It calls a procedure "StatusChange" that is implemented as an empty procedure within TomasProcess. The Model environment (Tomas) can assign its own procedure to this call and do whatever it wants to do. StatusChange is called just before the scheduling mechanism selects the next current element and (eventually) advances the simulation clock.

The code and the user manual of TomasProcess can be found in appendix C.

6.5.3. The standard model definition of Tomas

The Tomas unit and form represent the Model object. It also defines the basic object classes TomasElement and TomasQueue that are implementations of the `Simulation_Element` and `Set` classes of chapter 5.
The TomasElement class inherits all internal process services of the TomasProcessElement class (including the virtual process service), adds services for the external control of its process and all set handling services. If a modeler wants to define a simulation class with a specific process, then a class should be derived from TomasElement and the process service must be overridden. The general definition looks like (bold faced words are Delphi keywords):

```
myElementClass = class(TomasElement)
Published
Procedure Process; override;
End;
```

myElementClass can be any user defined string.
In the implementation the process service is specified by:

```
Procedure myElementClass.Process;
Begin
        …
        user defined process code
        …
End;
```

The TomasQueue class offers all services required for the set handling concept specified in chapter 5. Only TomasElements are allowed to enter a TomasQueue. Searching an element in a set can be very time consuming, especially when the number of elements becomes large. For this reason TomasElements and TomasQueues are double linked in a matrix structure. The TomasQueue itself owns a double linked list with all elements coupled by a successor-predecessor relation. Each TomasElement owns a double linked list with all TomasQueues it is a member of, coupled by a successor-predecessor relation. Usually the number of queues an element is in, is much smaller than the number of elements in a queue. The speed of searching a successor or predecessor of an element in a queue increases by entering the queue directly via the double linked list of the TomasElement (see figure 6.5.)

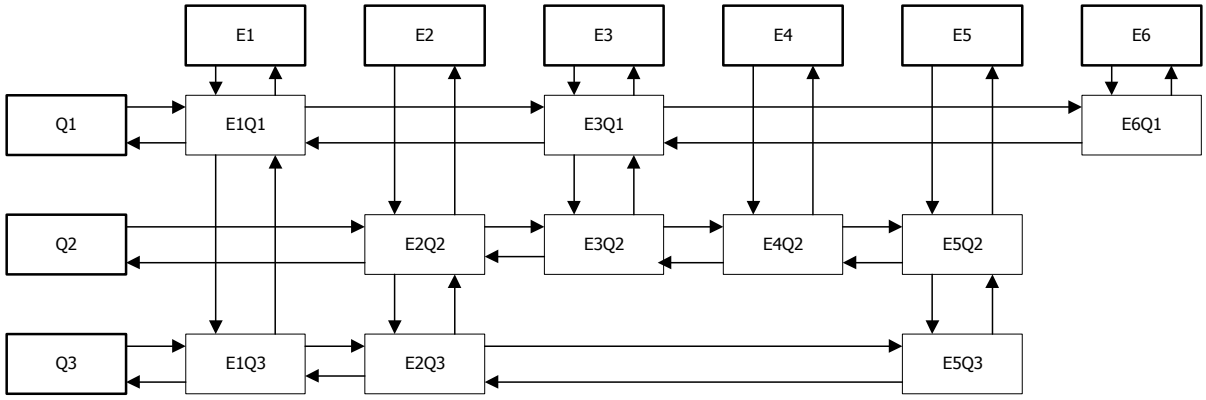


Figure 6.5: *Matrix of Element-Queue linking*
Ei = TomasElement i, Qi = TomasQueue i
EiQj = TomasElement i in TomasQueue i.

A modeler can add all the specific set services he needs, by defining a derived class again. For example, if a TomasQueue contains objects of class VolumeElement (derived from the class TomasElement) with a "Volume" attribute, the TomasQueue can be expanded with a GetVolume service as shown in example 6.1.

Example 6.1. A derived TomasQueue class.

```
MyVolumeQueue = class(TomasQueue)
Published
        Function GetVolume: double;
End;

Function myVolumeQueue.GetVolume: double;
Var
        Elm: VolumeElement;
Begin
        Result:=0;
        Elm:=FirstElement;
        While Elm <> Nil Do
        Begin
                Result:=Result + Elm.Volume;
                Elm:=Successor(Elm);
        End;
End;
```

The form of Tomas is called "TomasForm" and it is shown in the figure 6.6. The form is used to offer the modeler full run control and verification services. At the left side a so-called "TraceMemo" can be activated to show all occurring events during the simulation run. The run can be executed in "step mode", waiting after each event for the modeler to press the button for the next event to be executed.

During the simulation run the modeler can double click the Elements and Queues fields at the bottom; Tomas then shows an overview of all TomasElements and TomasQueues in the system.
At the right hand side the simulation run can be ordered to run until a specified time and the output can be directed to a user specified file. The option "Server Address" will be discussed in paragraph 6.6.

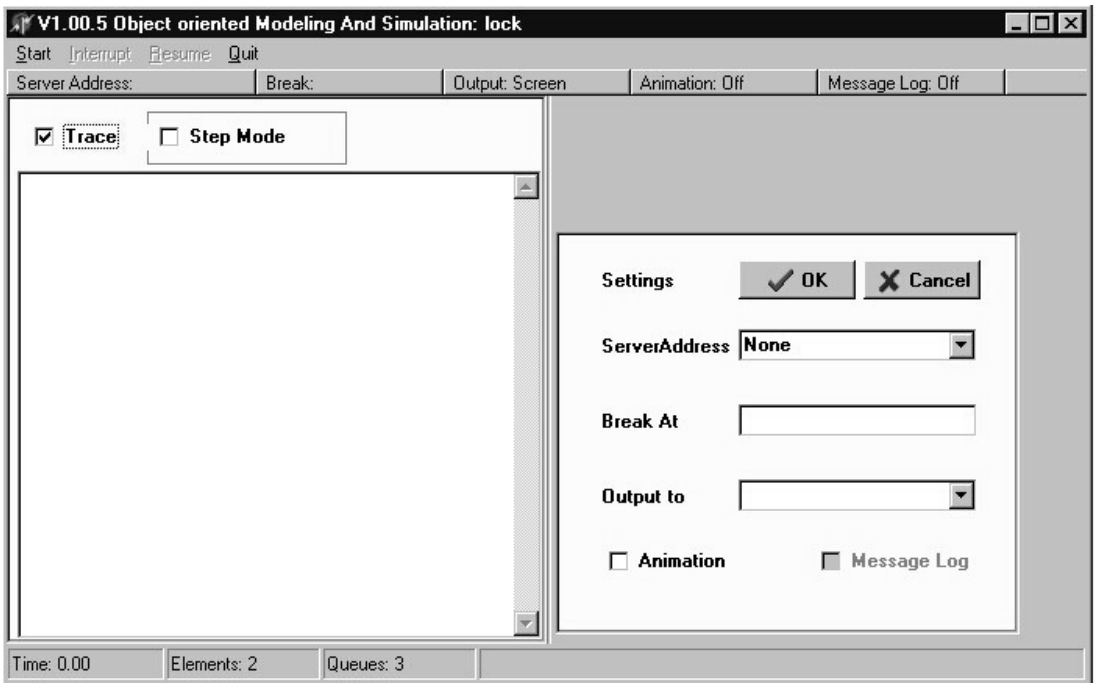


Figure 6.6: *The form TomasForm*

Finally by pressing the "Start" menu item in the left upper corner the simulation starts by giving control to the scheduling mechanism of TomasProcess.

6.5.4. Random number generation and distributions

To represent stochastic behavior all simulation languages require a random number generator. In Tomas the generator used is a so-called 'Lehmer random number generator' which returns a pseudo-random number uniformly distributed between 0.0 and 1.0. Each next number R_{i+1} is calculated from R_i as follows:

$$R_{i+1} = (16807 \times R_i) \bmod (2^{31} - 1) \qquad (6.1)$$

The sequence $\{R_i\}$ repeats itself in this way after the maximum period of $2^{31} - 1$. To check the quality of random generators several statistic tests can be applied. Applying the most recommended "Kolmogorov – Smirnov" test to the generator of (6.1) proves the formula delivers a well fitting uniform distribution [Chakravarti et al., 1967].
In Tomas a class TDistribution is introduced to be the parent class for all statistical distributions required. The class TDistribution owns a "seed" attribute value, representing R_i of formula 6.1. A call to its (virtual) method "Sample" first calculates the next random

number R_{i+1} and returns the value 1/ R_{i+1}. The resulting number series represents the standard uniform distribution U[0,1].
In this way derived classes can be created representing different statistical distributions. Tomas offers the following distribution classes:

- TUniformDistribution: the uniform distribution between a and b
- TNormalDistribution: the normal distribution with mean μ and standard deviation σ.
- TExponentialDistribution: the exponential distribution with mean μ.

Other theoretical distributions (e.g. the gamma distribution or the Erlang distribution) can also be derived from the parent class.

For ease of use two special distribution classes are derived:

- TTableDistribution, a distribution composed of paired values (X,F), X is the independent variable and F is the probability value, $0 \leq F \leq 1$. The values can represent a discrete distribution or a cumulative distribution. In this way each set of measurements can directly be used as a distribution without efforts to fit these measurements to a theoretical distribution.
- TInputDistribution, to enable the user to change distributions for different simulation runs by means of an input file.

The user has to specify a starting Seed value for each distribution defined. As can be seen from formula 6.1 there is a danger that two random series unintentionally coincide if the seed values are chosen close to each other in the series. For this purpose a separate unit TomasSeeds is created containing 2000 seed values that guarantee a non-overlapping random series for about one million successive samples. Each seed value can be retrieved by using "seed(i)" where $1 \leq i \leq 2000$.

6.5.5. Tomas extensions

By using the software platform Delphi, no further items are required to create a stand alone simulation model. Delphi is a so-called Rapid Application Development environment and

offers all facilities a modeler needs; the modeler however is assumed to know, how to develop Delphi applications. Extensions have therefore been added to TOMAS for educational and illustration purposes. The extensions are summarized below.

- *Graphical representation.*
  In order to support a graphical representation of observations during a simulation run Tomas uses a TCollection class. A TCollection class is able to collect the value of a variable and to store it together with the simulation clock time of observation. Based on this series of observations three types of graphical observation can be selected that are updated during the simulation:

  - a line graph showing the time series of the variable
  - a barred graph showing bars at each moment of observation
  - a histogram showing the statistics of the values, neglecting the time of observation.

- *Resources and semaphores*
  Resources and semaphores are predefined TomasElements for a generic representation of limited capacities. This type of elements is often encountered in transaction based simulation packages and is added to Tomas to support the development of simple educational models.
  Limited capacity occurs in two ways:
  a. TomasElements with a process want to use the same resource. For example cars want to enter a highway at the same entrance point. For this purpose the '*TomasSemaphore*' is introduced. TomasElements can ask for, wait for and release a certain capacity level of a TomasSemaphore.
  b. Flow elements are handled by a TomasElement with limited capacity. For example a machine in a factory is used for different jobs. This kind of problems can easily be modeled with the '*TomasResource*' concept. A TomasResource has a predefined Process service and capacity attribute. Not only quantities but also durations are involved. A TomasResource is automatically scheduled and triggered by claim requests or releases of flow elements.

  TomasSemaphore is a primitive class of objects representing limited capacity that is shared between active TomasElements (the "claiming" TomasElements will be

delayed). TomasResource is a derived TomasElement class to share limited capacity between arbitrary objects.

- *Animation*
  Each modern simulation package is required to offer animation functionality. The main reasons to use animation are:
  - to present the operation and results of a simulation model. Especially for commercial simulation applications a 3D-animation feature is indispensably. The statement "to see is to believe" still holds.
  - To support the modeler in verification (is my model working right?). For this purpose 2D-animation is sufficient.

  Delphi itself offers 2D-animation possibilities by means of extensive drawing services for the "canvas" of a Delphi form. To support 3D-animation Tomas is extended with a separate form: TomasAnimationForm, containing the required services to define a 3D-world and draw figures in it. In order to show and move TomasElements in this 3D-world, a class "TomasShape" is introduced. A TomasShape element is a TomasElement with a shape, a position and an orientation. Services are included to specify shape, position and orientation, and to describe movements of the TomasShape.

6.6. Distributed simulation for logistic design

The previous paragraphs dealt with stand alone models. The `Simulation_World` and the `Model` classes of chapter 5 are combined into one `Model` class so far, because there is only one `Model` object, so the `Simulation_World` services for the simulation clock, `Send` and `Receive` can be kept locally. In this way unnecessary loss of computing speed by communication delays is prevented, even in cases where the model is split into different aggregation strata.

In order to support distributed simulation, explicit communication with the TomasServer application is required (see figure 6.2). TomasServer now serves as the `Simulation_World` object synchronizing the different models and providing the required communication facilities. In the TomasServer a "leading" simulation clock is present. Each

participating model requests the TomasServer to execute its next event. Depending on the event times, the model with the smallest event time is allowed to proceed and execute this event.

The reasons to make a simulation model a distributed model are:

- To support concurrent modeling.
  As mentioned before, simulation is nowadays being used is almost all logistic design projects. Together with the growing complexity of designs, the simulation efforts exceed the capacity of one single modeler.
- To enable prototyping and testing of real resources and control functions.
  A simulation models can be used to create a "virtual environment" in which newly developed resources and control functions can be tested as if they are operating in reality. Costs, reliability and safety are the main motives behind this development.
- To increase the running speed of a complex simulation.
  Although computing power has increased enormously, there is still a need to speed up simulation experiments. The object oriented approach stimulated this type of distribution, because objects can easily be isolated and implemented separately.

For logistic design projects the first two reasons apply without question. The last reason however can be questioned. The basic idea to increase the speed is to use more processors in parallel for the same task. To achieve this a program should be split into well-defined parts and object orientation fits to this purpose. It is not trivial however to isolate elements of a simulation model from their surrounding system. In chapter 4 it has already been shown that the time dependent behavior of an element not only depends on the element itself, but is also effected by other elements. Contrary to standard applications, the element's interface should be synchronized to the simulation clock. In order to preserve reproducibility this interface synchronization should be defined unambiguously. This is the main restriction to achieve significant speed increase [Fuji et al, 1999]. Usually parallelism is achieved by allowing each member model to simulate a predefined period in advance autonomously, before allowance is asked to proceed. If interventions occur in the mean time one of two possibilities is selected:

- to trace back to the situation where the first intervention occurred and repeat the execution form this moment in simulation time ("back-tracking")

- to decrease the time period in which models may proceed autonomously.

Both possibilities introduce an amount of overhead by which part of the speed increase is lost again. For both back-tracking and period decrease, the model with the highest event frequency determines the overhead. Another complication is reproducibility. Dependent on the synchronization to the real time clock of the parallel processors, different results may occur.

The above mentioned complications have led to the decision to implement two extremes of parallelism in Tomas:

- for pure virtual experiments the conservative method is used. This method implies that only one model is active at any moment, just as in the stand alone situation. In this case there is no real parallelism at all. This approach therefore only supports concurrent modeling. In order to minimize speed loss because of communication with the TomasServer, the method has been improved as follows. TomasServer knows the first future event times of all participating models. The model that is allowed to execute its event receives besides the allowance the first future event time of the next model in line. In this way the model is able to proceed autonomously until this future event time, even if it generates events in between. As soon as the next event time equals or passes the received event time of the other model, allowance has to be asked to proceed. This implementation proves to be very effective in cases where different time granularities per model occur; this usually occurs when models at different aggregation strata are combined (e.g. one global model and one or more detailed models).
- For prototyping and testing of real resources and controls the simulation clock is synchronized to the real time clock. The user is able to define the time interval by which the simulation models proceed, before TomasServer synchronizes to the real time clock. During this time interval the models proceed according the improved conservative method. Only the models on the one side and the real resources on the other side run in parallel.

The improved conservative method preserves reproducibility, if the models are started in the same order. If the simulation is synchronized to the real time clock, reproducibility is not

required. Both resources and controls are to be tested for real circumstances and usually the goal is to achieve robustness.

The communication between TomasServer and participating models is implemented with the standard Transmission Control Protocol/Internet Protocol (TCP/IP). TCP/IP is actually a collection of protocols, or rules, that govern the way data travels from one machine to another across networks. If the internet address of the machine where TomasServer is running is known, then each participating model is able to establish a connection. As shown in figure 6.6 this address can be entered into the TomasForm (or read programmatically). In order to make the connection transparent for the user the communication procedures have been implemented in TomasProcess. At the start of a simulation run, TomasProcess checks if a server address is specified. If this is not the case, the model is assumed to run stand alone; if an address is specified, a connection is made to the TomasServer.

The concept has been successfully applied in the construction of a "simulation backbone" for the research program FAMAS MV.2 [Veeke et al, 2002, Boer C.A. et al, 2002].
The program investigates terminal structures for container handling on the planned extension of the Rotterdam port area "Maasvlakte 2". The concept allows also other simulation platforms (Arena, em_Plant) to participate in the distributed experiments.

Chapter 7

Case: The Delta Sea-Land Terminal (DSL)

There are two ways of meeting difficulties: you alter the difficulties or you alter yourself meeting them.

- Phyllis Bottome -

7.1. Introduction

In November 1988 a project program was launched to create an innovative container terminal at the ECT peninsula of the Maasvlakte in Rotterdam, the Netherlands. ECT (Europe Combined Terminals) is the major container stevedore in the port of Rotterdam. The growing volume of container transport urged ECT as a stevedore and Sea-Land as a carrier to reach an agreement about the container handling facilities that would be profitable for both parties. ECT was operating in an environment with severe competition, where handling costs and service rates were subject to great pressure. On the other hand, ships and volumes were growing and Sea-Land had to realize shorter and guaranteed turnaround times on its main shipping lines.

In the years preceding the formulation of the project program, ECT performed several feasibility studies and it was found that a container handling facility could become cost effective and competitive by automating a great part of container handling. In terms of function performance as defined in par. 2.5.3., the theoretical productivity of this facility was considered better than other alternatives that were investigated.

In this chapter, the design process of the terminal is described as it started at the moment the project program appeared. At that time ECT and Sea-Land had reached an agreement about the functionality and performance of the new terminal. So the overall strategic phase was finished and the project entered the tactical design phase. The main goal of this phase was to define a standard productivity for the terminal and all of its components by which the real operation could be evaluated. To achieve this, all project groups involved should have a clear objective in terms of a theoretical productivity (results-in-view and efforts-to-be-expected) with respect to their field of design. In order to assign these productivity figures, the project program of November 1988 is interpreted by using the PROcess – PERformance

model (PROPER). A global simulation model in TOMAS will be used to interpret these results and quantify the efforts by means of time dependent behavior descriptions.
After that, the use of the approach will be discussed compared with the real execution of the design project.

It is in no way the intention of the analysis presented in this chapter to criticize the former project group and participants (the writer of this thesis was one of them), but the outcome will be used to show that some of the problems that appeared after the start of real terminal operations, could have been predicted from this project program. Nevertheless, the project was a masterpiece of innovative design by creating a completely new container system that "worked" at the intended start in 1993, including the fundamental innovations throughout the whole system. Proof of this can also be found in practice, because it took almost 10 years before real improvements could be realized (in Rotterdam and Hamburg).

It should also be realized that computer technology in the early nineties was not that advanced; complex models required the newest hardware and distributed simulation was not used for this kind of projects yet. Wherever possible the use of afterward knowledge is avoided, but even if afterward knowledge is used then the PROPER model proves to be a suitable means to save this knowledge in a structured way for future use.

First a summary of the project program (par. 7.2) and the main requirements (par. 7.3) are described without comments. After that these requirements are positioned in the PROPER model and sorted to three system levels: the terminal as a whole, the level of the different aspects (order, product and resource) and finally the function level within the product flow. The requirements for the automated system will then be quantified by means of a global TOMAS model.

7.2. The project program

The project program was a result of a negotiation trajectory between ECT and Sea-Land [ECT,1988]. The decision was made to create a new terminal at the Maasvlakte in Rotterdam fully dedicated to the handling of containers carried by Sea-Land (figure 7.1).

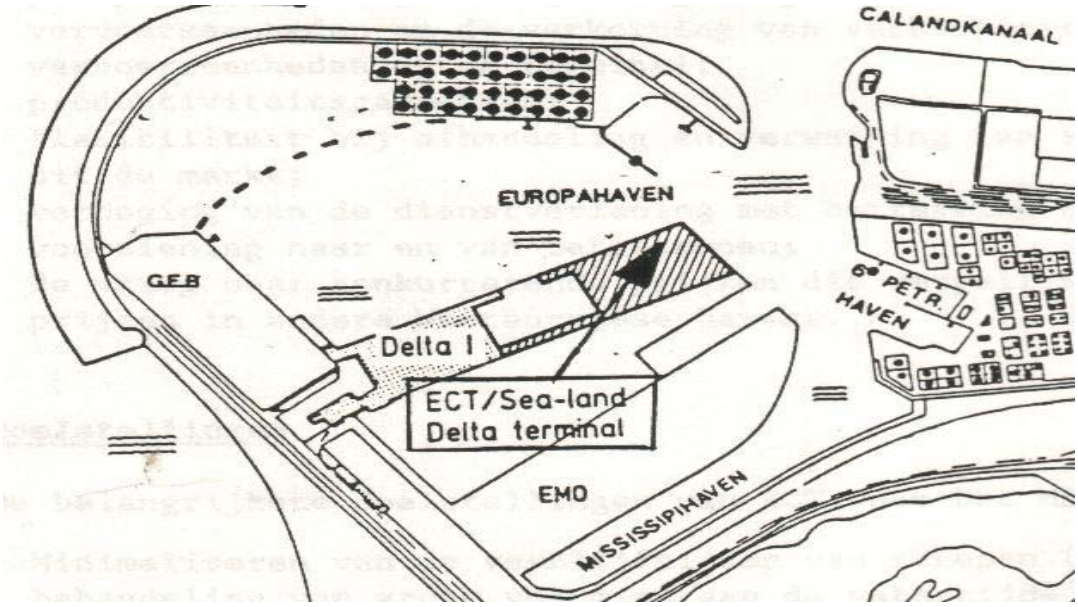


Figure 7.1: *Position of the Delta Sea-Land terminal [from ECT, 1988]*

The terminal to be build is based on the concept of "Majority Sea to Sea" (MSS); it is a system for container handling primarily aimed at water-water transfer of containers and was therefore called the "MSS system".

The project organization (see figure 7.2) reflects the multidisciplinary approach, coordinated by a project steering committee in which all project groups were represented.

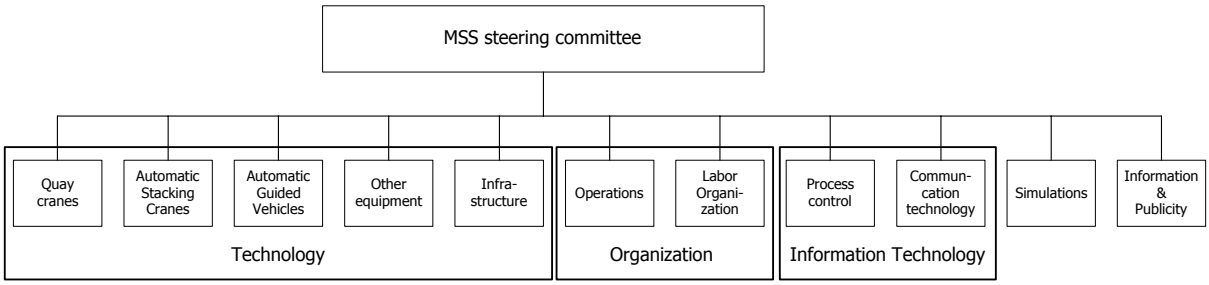


Figure 7.2: *Project organization of the MSS project*

The group "Simulations" should support all other groups where necessary, and was leading in the development of process control algorithms.

The ideas about the system resulted in the following prescribed combinations of operations and resources (see figure 7.3):

- at the water side quay cranes (QC) will load and unload deep sea ships and feeders
- Automatic Guided Vehicles (AGV) transport the containers between stacking area and quay cranes
- Automatic Stacking Cranes (ASC) store and retrieve containers into and from the stacking area.
- Straddle Carriers (SC) load and unload trucks and multi trailer systems (MTS) at the land side of the terminal. One MTS transports up to five 40 feet containers between rail or barge terminal and the MSS terminal.

Figure 7.3: *An artist impression of the MSS system by Rudolf Das.*

The MSS system should cope with the next changes in the market:

- a growing volume of container transfer.
- Larger transportation units, both at ocean shipping and at inland shipping and rail, with the consequence of increased peak operations
- A reduction of berth times and service times
- The demand for guaranteed productivity
- The demand for competitive prices compared to other harbors in Western Europe.

These changes were translated into objectives. The main (specifically operational) objectives of ECT for the MSS project were:

- to minimize berth times of ships by optimal use of resources
- to achieve a high quay crane productivity
- to increase customer service levels
- to reduce and control costs by increased productivity, decreased labor and maintenance costs and improved use of information technology.

These objectives resulted in four types of requirements:

- preconditions that cannot be influenced by ECT
- functional requirements with respect to performance, functionality and properties of the terminal
- operational requirements with respect to the use of the terminal
- design restrictions dictated by the organization.

Preconditions mainly express what system should be used, what type of guarantees should be given and where the terminal should be build; they contain almost no quantitative data. For the purpose of this thesis the functional requirements are most important and they will therefore be elaborated further. Operational requirements and design restrictions will only be mentioned where necessary.

7.3. Functional requirements

The functional requirements are formulated under the following basic assumptions:

- the following source-destination rates of containers (the so-called *modal split*): 55% Of the containers imported at waterside will leave at water side, 38% will leave by rail and road and 7% will leave by barges. 57.5% Of the containers leaving at water side entered at water side, 35% originated from rail and road and 7.5% from barges.
- the arrival pattern of ships consists of 3 main line ships together with some tens of feeders each week
- ship sizes are approximately 4000 TEU (Twenty feet Equivalent Unit) for main line ships and approximately 1250 TEU for feeders

- an average dwell time of 3 days for filled containers. The dwell time of a container is defined as the period between unloading from ship and truck and loading to a ship or truck.
- the composition of the container flow (full containers, empty containers, reefers and off-standards) will be the same as in 1988
- containers for export arrive on time and the loading sequence is known in time in order to guarantee a continued operation of quay cranes
- there are 22 net operating hours each day (24 hours).

The contract with Sea-Land provides a review of requirements if these assumptions would change. The assumptions are translated into requirements for each aspect, and should be evaluated regularly.
Finally it should be mentioned that the contract with Sea-Land is based on a cost price that is composed as follows:

- labor costs	36%
- materials	3%
- third party services, quay rent	18%
- debit and capital costs	43%

At the level of the terminal as a whole the following (major) requirements were defined:

- The new terminal should be able to handle 500.000 quay moves per year (incl. barges). Barges are handled at a separate location. At the sea side of the terminal itself there will be 475.000 moves per year.
- The system as a whole should be able to handle with each quay crane 600 containers a day (of 24 hours) and after 3 years 630 containers a day.
- The system (excluding the quay cranes) should have an operational reliability of 99%. The average time of a system failure should not exceed 0.5 hours.

Two general requirements were formulated:

- the number of container movements during the stacking and transfer processes should be minimized. This will be achieved by minimizing storage layers and a smart stacking control.

- The different container flows will be controlled by a programmed and centralized control.

With respect to the customer's transport modalities the following requirements were formulated:

A1. as mentioned before, 3 main line ships per week are expected with several tens of feeders. In addition to this, the requirements specify the contractual port times for these ships as follows:

- Deep sea 24 - 32 hours (depending on the type of ship)
- Feeders 12 hours

The project program continues with: in order to realize these port times an average of 100 containers per gross operational hour (GOH) should be achieved.

A2. The total service time (gate-in gate-out) for trucks should be approximately 30 minutes. To achieve this the average length of stay at a delivery position should be 20 minutes. The latter is further specified in relation to the arrival rate (as shown in table 7.1.)

Truck arrivals / hour	**Average length of stay**	**95% level**
90 – 140	30 min	60 min
40 – 90	20 min	40 min
< 40	15 min	30 min

Table 7.1. Length of stay at delivery point

The arrivals assume single operations (one container). For trucks that require more operations (unload and load for example or two 20 feet containers) the length of stay for an arrival may increase by 50% (independent of the number of operations).

A3. The relations between arrival and departure of a container were also defined. Import containers from sea should be available for delivery to road from 2 hours after unloading. For delivery to rail 3 hours after unloading is required. Export containers can be delivered to the terminal til the time of mooring of the ship. In

cooperation with carriers it will be attempted to have trains and barges unloaded 3 hours before ship arrival.

The project program specifies several requirements for the container flows.

M1. Even under non-optimal conditions the transfer and stacking system should be able to handle an average of 260 containers each Net Operating Hour (NOH) with 8 operational quay cranes.

M2. ECT adds the requirement that an average of 40 containers per quay crane each NOH with 6 quay cranes should be handled by the system. This should result in 240 containers each NOH.

M3. The latter requirement is further refined by the demand that during two consecutive hours 1 or 2 quay cranes should be able to handle a peak load of 60 containers per hour. The maximum number of containers to be handled with all 6 quay cranes each NOH remains 240 in this situation.

M4. At the land side of the terminal the system should be able to receive and deliver 140 containers per hour (at least during two consecutive hours). Of these containers 110 containers belong to road transport and 30 to rail transport (which is equivalent to 6 MTS arrivals). The system should be able to handle 90 containers per hour at the land side.

Besides requirements for operational circumstances, requirements for maintainability and the aspiration to use proven technology the following operational requirements were formulated for the resources:

E1. A number of 8 quay cranes will be provided, 3 of which should be exchangeable with the neighbor terminal.

E2. The MSS system will use AGV's, ASC's and straddle carriers.

E3. The MSS system (excluding quay cranes) should have an operational reliability of 99% (with 8760 hours in a year, it means that the system may be out of action for only 90 hours). The failure time should be less than 0.5 hour in the average and may not exceed 1 hour.

E4. Each quay crane should have a reliability of 99%. The failure time should be less than 0.5 hour in the average and should not exceed 1 hour.

E5. Strategic parts of the system (hard- and software, data communication) should have a reliability of 99.9%.

7.4. Application of the PROPER model

7.4.1. The terminal level

From an organizational point of view, each terminal operation is divided into periods of 4 hours. These "shifts" have a fixed number of operational manned equipment (quay cranes and straddle carriers) and of personnel for process control. Between each shift the current operation status and the responsibility for it, is transferred to the new shift. This organizational condition is not mentioned in the project program, but it plays an important role in process control.

In 't Veld [2002] defines three conditions to reach an effective process control anyhow:

- there should be a clear objective
- the objective should be feasible
- it should be possible to intervene with the system's behavior.

In the project program no objectives for shifts are found in terms of intended results and expected efforts; only intended results per hour, day and year are formulated. Applying the conditions of in 't Veld to the project program and with the restriction of a shift oriented operation, the project program should be interpreted in a way that at least the intended results and expected efforts for a shift are clear and feasible. Defining interventions is considered a next step in process design and will not be discussed further in this application. In order to intervene however the progress of an operation should be clear with respect to the objective and this factor will be considered here.
In the project program different intended results are defined. The major intended results are summarized in table 7.2.

Type	**Result**	**Meas. Unit**	**Location**
Quay moves	475.000	Year	Sea side
Containers (main line)	100	GOH	Sea side

Containers	240 / 260	NOH	Sea side
Containers	90 / 140	Hour	Land Side
Port times (main line)	24-32 hour	Ship	Sea side
Port times (feeder)	12 hours	Ship	Sea side
Gate in- Gate out times (truck)	30 minutes	Truck	Land side
Handling times	20 minutes	Truck	Land side
Throughput times	2, 3 hours	Container	Terminal
Operational hours	22 hours	Day	Terminal

Table 7.2. The results-in-view mentioned in the project program

In the intended results, the terms "moves" and "containers" are used. At the level of the terminal as a whole, they do have a different meaning: for each container at least two moves are required, because each container will be stored into the stack by definition. So at least one move is required to unload and store it and one move to retrieve and load it. At the level of the container flow the terms "move" and "container" are interchangeable: each function within the transfer process handles one container with one move.

The PROPER model of figure 2.9. is taken as a starting point to position these results in view. Figure 7.4. is the result of replacing the general terms of figure 2.9. with terms that are specific for this system.

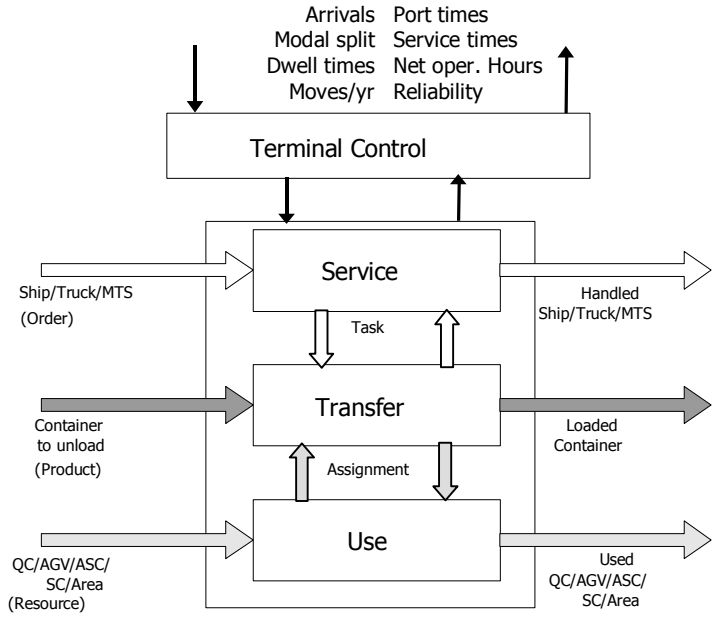


Figure 7.4: *The PROPER model for the DSL terminal*

Terminal control reports the performance to the environment, in this case the contractual partner Sea-Land and terminal management. These reports mainly concern the assumptions made and the guaranteed productions. In figure 7.4. they are positioned between the top arrows because reported results should be formulated in the same terms as the required results.
To the inside of the system they should be translated to required results for each flow separately. Results at this level are flow bound and in the model 5 flows can be distinguished:

1. ships, trucks and MTS's are considered *the order flow*. The order process is concerned with the required "customer service". For the modalities ship and truck the intended results are defined in two ways:

    - by means of the port and service times. For MTS's they are not defined yet.
    - By means of arrivals (and thus departures) per week

    One condition is added : in order to realize the required port times, 100 containers per gross operational hour (GOH) should be handled. This condition is considered here as a first progress indicator for a ship's operation.
2. Containers represent *the product flow*. This process is concerned with the execution of moves, not necessarily directly related with customer service. For this flow different types of intended results are defined:

    - in terms of moves per hour. Different numbers are formulated: at sea side 260 moves per hour with 8 quay cranes and 240 moves per hour with 6 quay cranes, at land side 90 moves per hours and 140 moves per hour with an undefined number of straddle carriers. Quay cranes and straddle carriers operate at both sides of the transfer function; they handle both import and export containers. There is however no distinction made between the input flow rate and the output flow rate (during an hour). It is not clear yet whether this distinction is important.
    - In terms of quay moves per year. No land side figure is defined explicitly, but it can be derived from the assumed modal split. If 55% of the import quay moves represent 57.5% of the export quay moves and the total number of quay moves is 475.000 per year then there must be around 243.000 import

moves and 232.000 export moves per year. Using the percentages of the modal split, the terminal should be able to handle around 209.000 land side moves per year (truck and rail), divided over 109.000 import moves and 100.000 export moves.

Above that, the project program formulates the ambition to minimize the number of container movements. This defines the intended result that each container will be moved only once by each function in the transfer process.

3. *The resource flow* contains the quay cranes, AGV's, ASC's, straddle carriers and "space" (stack positions, quay length etc.). The resource process is concerned with technological items as reliability, maintenance, safety and operational speed of container moves. In this example, no flow rate will be defined, although the project program mentions their life spans in terms of operational hours. The operational functions for each of them have already been defined (see figure 7.3).
4. *The task flow* between the service function and the transfer function. A task is defined here as one order to store or retrieve one container. One complete container transit therefore consists of two consecutive tasks, decoupled by the ASC stack. A task's lead time is a measure for the response time of the system to container orders, which results in a required service time or port time; by adding priority levels to tasks the service function is able to intervene with the progress of sea side and land side operations.
   The project program defines the moments when tasks should be ready to be released, e.g. a truck with a container to be loaded in a ship, should be present before mooring time of the ship. The intended results for this flow are expressed in terms of number of tasks per shift and task lead times per shift.
   This example clearly illustrates the difference between order flow and (internal) task flow. One ship (a customer order) results in a large number of separate tasks. This introduces a number of choices to be made. One could decide to release all tasks immediately, but this neglects the formulation of a shift operation objective. One could also decide to release tasks one by one, but this could introduce delays in the operation and exclude optimization possibilities for task scheduling.
5. *The assignment flow* between the use function and the transfer function. The results-in-view of assignment are not defined in the project program. They should be expressed in equipment moves per shift, availability and occupation per shift,

because manned equipment will be assigned per shift. These results form the basis for the calculation of expected efforts at the terminal level.

The task and assignment flows contribute to the insight into the causing factors of an operation's performance. A bad performance can be the result of a late task release, long task lead times or a shortage in assigned equipment.

The shift based operation urges the intended results of the project program formulated in different time units, to be translated into intended results per shift. Some intended results are even missing. In order to define all these results, a zooming step to the model of figure 7.4. will be performed, resulting in separate models for each of the horizontal flows.

7.4.2. The order flow

Two geographically divided flows are distinguished at the terminal (figure 7.5.):

- a sea side flow consisting of ships and feeders
- a land side flow consisting of trucks and multi trailer systems (MTS).

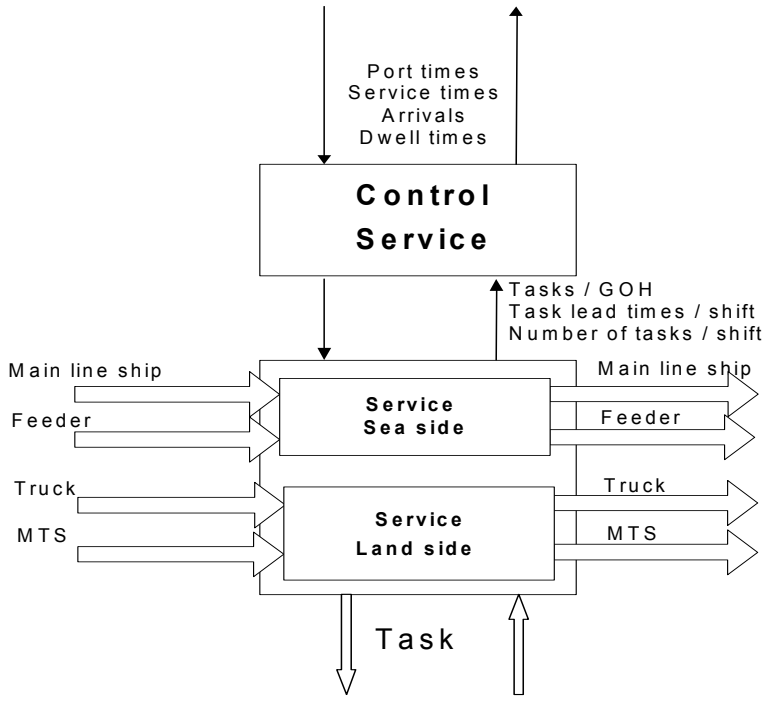


Figure 7.5: *The order flow at the DSL terminal*

Multi trailer systems (MTS) can transport up to five 40-feet containers. They transport containers to and from trains, barges and other terminals on the ECT territory. The MTS transports are not a direct responsibility of the DSL terminal and therefore they are considered a "customer" modality.
Tasks should be divided in tasks for sea side and land side and in tasks to store and to retrieve a container. By measuring the performed tasks / GOH the control service function is able to judge the progress of the service. The value is transformed into performed tasks / shift by just adding the values of the 4 shift hours (a shift is in fact a period of 4 GOH's). The measured task lead times per shift and the number of released tasks per shift enable the control service function to derive the standard results per shift for different arrival compositions and ship operation phases. The period between the unload task and the load task of a specific container represents the dwell time.

7.4.3. The product flow

In the container flow two geographically separated flows are distinguished again. The flows have an overlap in the ASC-stack (figure 7.6).

The equipment used for each function is shown in the bottom-right corner of a function. At land side the straddle carriers perform the functions unload, transport and load.

The control transfer function combines released tasks with assigned resources to perform the transfer function. The goal is to achieve the required tasks per shift with the resources assigned to the operation. The tasks will be split into separate tasks for each function. Consecutive functions within the flow should be executed in an optimal way.

Handled tasks are returned to the service function and assigned resources are released at the end of a shift.

The intended results as formulated in the project program are expressed in terms of "net operational hours". No explanation is given on how to measure this unit. By just measuring moves per hour (which is in fact a gross operational hour) the relation with the number of moves per year is clear and it can be used by terminal control as a progress indicator during the year.

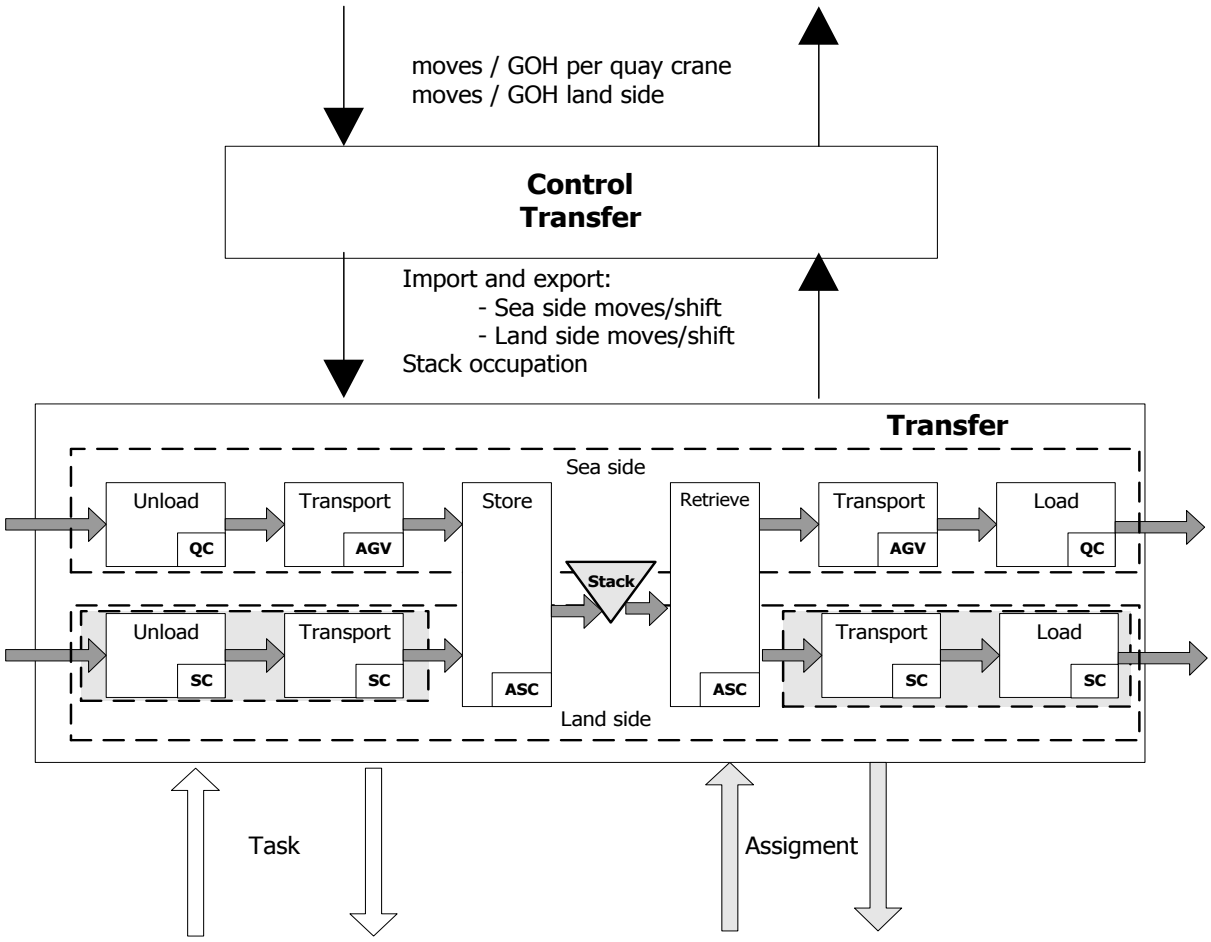


Figure 7.6: *The container flow at the DSL terminal*

The ASC's are the only resources involved in both the sea side operation and the land side operation. This means that the effective operational capacity of the sea side system and the land side system fluctuate during a shift. Where the other resources can be assigned to sea side or land side exclusively, the ASC resources should be assigned to the sum of sea side and land side tasks. Under the condition that land side tasks are to a large extent unpredictable (and so are the containers to be asked for), this is an extra reason to formulate results-in-view in shift units rather than in hour units.
Besides that each ASC is restricted to its own part of the stacking area. For the unload flows left to the ASC-stack in figure 7.6. this is no problem: the control function is free to choose an ASC area at will. For the load flows right to the ASC-stack it is an extra complication. In trying to optimize move times, the control function is bound to the ASC area, where the container is stored. This may lead to a congestion of tasks for a specific ASC and influence task lead times. It is therefore concluded that there should be made a distinction between input moves and output moves.

Each of the functions in figure 7.6. should have the same result in view defined in order to prevent the creation of a bottle neck. Especially the sea side operation consists of three closely coupled functions with different resources. Having the results in view defined then the expected costs for each of the resources should be derived. In par. 7.5. this will be investigated further using simulation.

7.4.4. The resource flow

The use function aims to provide the required resources for the operation function. Manned resources (quay crane and straddle carrier) and personnel are assigned on a shift basis. There is no reason to use another time basis for the automatic resources. The results of assignment will therefore be expressed in terms of results per shift. The project program only defines results in view for assigned resources in terms of reliability. It is assumed here that 100% availability for assignment is required. To achieve this two things are required:

- the absolute numbers of available operational resources should be sufficient. The maintenance program and failure rates should fit to this.
- The numbers of required equipment should enable the achievement of operational results in view. The technical specifications should fit the operational requirements as mentioned in the preceding paragraph.

Assigned resources make up the labor and capital costs of the operation and form the basis for calculating the expected efforts. They will therefore be measured and reported to the terminal control function.
Restricting the use function to equipment and skipping personnel the use function is represented in figure 7.7.

During its life time, each equipment will "loop" between the status usable and in maintenance. Inside the use function these loops should be controlled in order to guarantee the required availability, which is an externally used intended result.

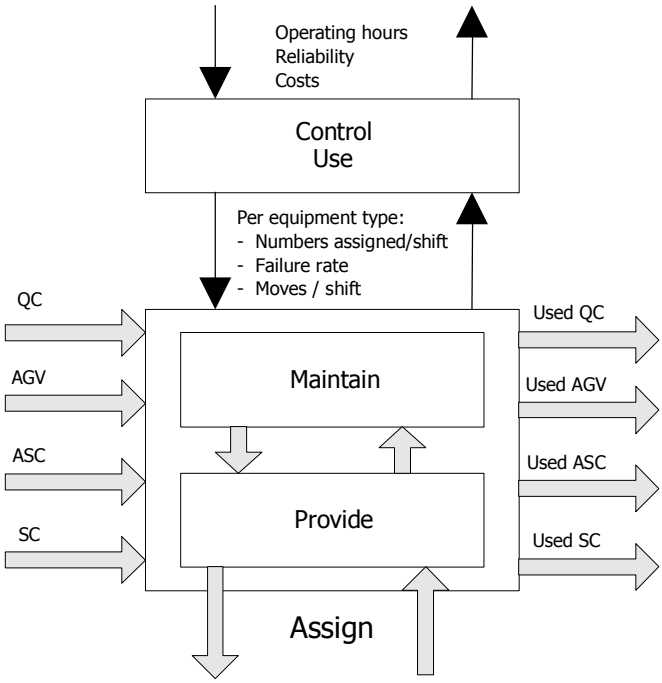


Figure 7.7: *The use function of the DSL terminal*

7.5. The use of simulation for productivity definitions

7.5.1. Definition of simulation goals

The availability requirements (and thus the expected efforts) have not been made explicit in the original project program. Only the results-in-view for quay cranes were defined, but with different time units than required for the control functions. Above that task lead times, transport, store and retrieve times still have to be defined as shown in the previous paragraphs. These data are required to answer the availability questions and to define technical requirements for the equipment. With these data it will be possible to judge the feasibility of the defined results in view. At the moment of appearance of the project program, these data were unknown, because ASC's and AGV's still had to be developed from scratch. Simulation however can be used to derive these data at the level of the functions of figure 7.6. Assuming the quay cranes are able to achieve the required performance, then the requirements should be translated to requirements for the AGV's and ASC's.

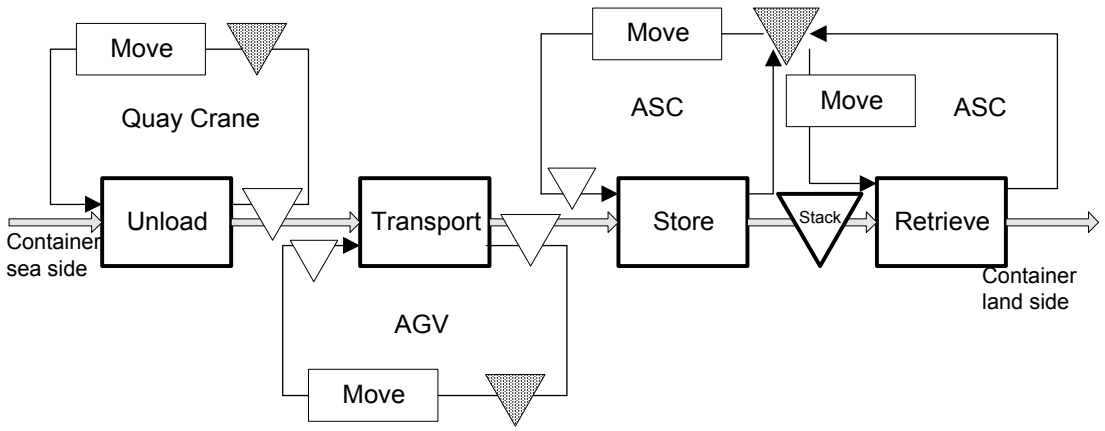


Figure 7.8: *The import process of the DSL terminal*

For illustration purposes the next study is restricted to the import functions and within these the operational processes of containers and the land side operation is taken into account until the point of retrieving a container from the stack. The resources used for each function are added revealing their repetitive processes by this. The resulting process is shown in figure 7.8.
In figure 7.8. triangles represent waiting times. The shaded triangles represent 'idle times' for the resources: they are waiting there for a task to be assigned. The white triangles are waiting times caused by an imperfect synchronization.

It is assumed that the assignment of resources to the operation has been taken care of and all quay cranes, AGV's and ASC's are available. In the following description it is assumed that tasks are known and available in time.

Each "container move" consists of two functions for each equipment: a "move" to the position to get a container and a perform of the actual function: "unload" (QC), "transport" (AGV), "store" and "retrieve" (ASC), which is finished by releasing the container. These two functions together are called one "cycle" from now on. AGV's don't have a facility to lift or put down a container and their operation is therefore strongly linked to the quay cranes and ASC's. The consequence is, that a third factor should be counted for in the definition of a cycle: the transfer of a container between AGV and the other equipment. With these factors the technical cycle time of equipment can be defined.
At land side the situation is different, because there are buffer positions provided where ASC's and straddle carriers can pick up or put down containers independently.

To formulate the equipment requirements for a feasible result in view of the system, the following global process description for an unloading process is given in table 7.2 (lines preceded with Qn refer to questions defined after the table).

Process of a Quay Crane *Repeat* *Q1 Work 'sample of QC's cycletime'* *Wait while no AGVs in AGVSWaiting* *Q2 Select AGV* *Q3 Load AGV in x seconds* *Resume AGV's process*	***Process of an AGV*** *Repeat* *Q4 Select QC* *Q5 Cycle is 'sample of AGV's cycletime'* *Drive 0.5 * Cycle *)* *Q6 Enter QC's AGVSWaiting* *Wait* *Q7 Select ASC* *Drive 0.5 * Cycle *)* *Q8 Enter ASC's AGVSWaiting* *Wait*
Process of an ASC *Repeat* *Wait while no AGVs in AGVSWaiting and no Landside jobs available* *Q9 Cycle = sample of ASC cycle time* *Q10 If AGVs in AGVSWaiting* *Q11 Select AGV* *Move to AGV in 0.5 * Cycle seconds *)* *Q12 Unload AGV in y seconds* *Resume AGV's process* *Store container in 0.5 * Cycle seconds *)* *Else* *Work D seconds* *Destroy Landside job*	***Process of Land Side*** *Repeat* *Wait(3600/90)* *Q13 Select ASC* *Create Landside job for ASC*

Table 7.2. Process descriptions for the MSS system

*) The duration of moving to the position where the AGV or ASC should get the container, is assumed to be half the cycle. This does not effect the results.

In this case every type of equipment (except a straddle carrier) has a process. The model assumes an undisturbed operation: there is always a sea side job for a quay crane. The results-in-view to be used are therefore the results per net operational hour of the project program. At land side it is assumed 90 containers per hour should be handled (their jobs arrive with a constant inter arrival time).
Straddle carriers are not modeled, because the transfer between straddle carrier and ASC is decoupled by means of a buffer. Only the work load resulting from the straddle carriers is modeled. The cycle times mentioned in table 7.2. exclude the transfer time at the transfer position.

The goal of the model is to gain insight in a feasible productivity; it is not the goal to optimize the operation already. Control algorithms are not modeled. Keeping this in mind, the process descriptions introduce 13 questions to the project team:

Q1. What is the cycle time of a quay crane unload move?
Q2. How does a quay crane select an AGV from the AGV's waiting to be loaded?
Q3. How long does it take to load an AGV by a quay crane (x seconds)?
Q4. When and how selects an AGV a quay crane to drive to?
Q5. What is the cycle time of an AGV, a cycle being the sum of driving times to quay crane and returning to an ASC?
Q6. Is there an upper limit to the number of AGV's waiting for a quay crane?
Q7. How does an AGV select an ASC to deliver the container to?
Q8. Is there an upper limit to the number of AGV's waiting for an ASC?
Q9. What is the cycle time of an ASC?
Q10. Is there a priority order between sea side and land side jobs to be handled by the same ASC?
Q11. How does an ASC select an AGV if there is more than one AGV waiting?
Q12. How long does it take to unload an AGV by an ASC (y seconds)?
Q13. How are ASC's selected for land side jobs?

The numbers of equipment together with their cycle times (Q1, Q3, Q5, Q9, Q12) mainly determine the results at the sea side of the system. For the other questions the most likely or 'best case' realization can be chosen (for the time being).

The (assumed) answers to the other questions are now:

Q6, Q8: there are no upper limits for the number of AGV's waiting for a QC or ASC.

Q2, Q11: both quay cranes and ASC's select AGV's in a First-In-First-Out order. This selection does not influence the results at this level of detail, AGV's nor containers have an identity here.

Q10: the process description shows that sea side jobs are selected before land side jobs, because an ASC first checks if AGV's are available. This will maximize the sea side performance.

Q7, Q13: Both AGV's and Land Side select an ASC according a uniform distribution.
A second alternative could be investigated where AGV's and Land Side select an ASC according the work load already assigned to an ASC. The selection is assumed to be made at the moment of arrival.
One should keep in mind, that the work load based selection is only possible for import containers. Export containers already are in the stack and thus the ASC to handle it, is determined beforehand.

Q4: This question is of great influence, because quay cranes are highly unpredictable. If an AGV selects a quay crane in an early stage of it's cycle than this can be a bad choice if the quay crane accidentally shows large cycles after this choice.
For the 'best case' alternative AGV's don't select a quay crane at all. They are assumed to arrive in a large pool of waiting AGV's, where each quay crane can select an AGV instantaneously. In this alternative the consequences of early (and thereby non-optimal) selection are excluded. A second alternative will be the situation where an AGV selects the destination quay crane at the moment of arrival in the quay crane area. Finally in the third alternative an AGV selects a quay crane at the moment of departure from the ASC.
Reality will be somewhere between the second and third alternative. The first alternative will only be used to find a starting point for other experiments.

The following assumptions will be made:

- loading an AGV by a quay crane will take 15 seconds
- QC cycle times for unloading will be 95 seconds for cases with 8 quay cranes and 75 seconds for cases with 6 quay cranes. Including the loading time of AGV's, this

means for 8 quay cranes a (maximum) average production of 260 containers per hour, for 6 quay cranes a production of 240 containers per hour. The 260 and 240 containers per hour agree with the requirements mentioned in the project program.

- Unloading an AGV by an ASC will take 15 seconds
- The cycle times of AGV's and ASC's are assumed to be uniformly distributed. The cycle times will be the result of the (still unknown) routing algorithm to be used and traffic delays.
- The cycle times of the quay cranes are assumed to be negative exponentially distributed.

A TOMAS model has been written for the description of table 7.2, in which the number of quay cranes, AGV's and ASC's and the cycle times can be varied. The processes in table 7.2. of an AGV and a QC are shown in TOMAS code below. The source code of the complete model is added in Appendix D.

Process of an AGV in TOMAS

```
Procedure TAGV.Process;
Var
    ASCNr: Integer;
    Cycle: Double;
    QC : TQC ;
Begin
    While TRUE Do
    Begin
        QC:=QCPriority.FirstElement;
        Inc(QC.AGVSAssigned);
        QCPriority.Sort;                //sorts QC's on AGVSAssigned
        Cycle:=AGVCycleTime.Sample;
        Hold(0.5 * Cycle);
        EnterQueue(QC.AGVSWaiting);
        Suspend;
        Hold(0.5 * Cycle);
        ASCNr:=Trunc(ASCSelection.Sample);
        EnterQueue(ASCS[ASCNr].AGVSWaiting);
```

```
            Suspend;
        End;
End;
```

Process of a quay crane in TOMAS

```
Procedure TQC.Process;
Var
        AGV: TAGV;
Begin

        While TRUE Do
        Begin
                Hold(QCCycleTime.Sample);
                While AGVSWaiting.Length = 0 Do
                        StandBy;
                AGV:=AGVSWaiting.FirstElement;
                AGVSWaiting.Remove(AGV);
                Hold(15);
                Dec(AGVSAssigned);
                QCPriority.Sort;
                AGV.Resume(TNow);
        End;
End;
```

7.5.2. Experiments

In the model 48 AGV's and 25 ASC's will be assumed available. These numbers of equipment were used during the early stages of process design. All experiments below can be repeated with other numbers in case of different availability figures. As long as these numbers are accepted, they become a standard result for the use function of figure 7.8.

Each experiment covers a period of 100 consecutive net operational hours, after an initial hour for operation start.

The first experiments are meant to derive an indication for the cycle times required to achieve a sea side productivity of 260 cnr / NOH with 8 quay cranes and 240 cnr / NOH with 6 quay cranes. First the required cycle time of AGV's is determined by setting the cycle time of ASC's to zero. For this the cycle times of AGV's used (including transfer times) are 300, 360, 420, 480, 540, 600 and 660 seconds. The results are shown in figure 7.9.

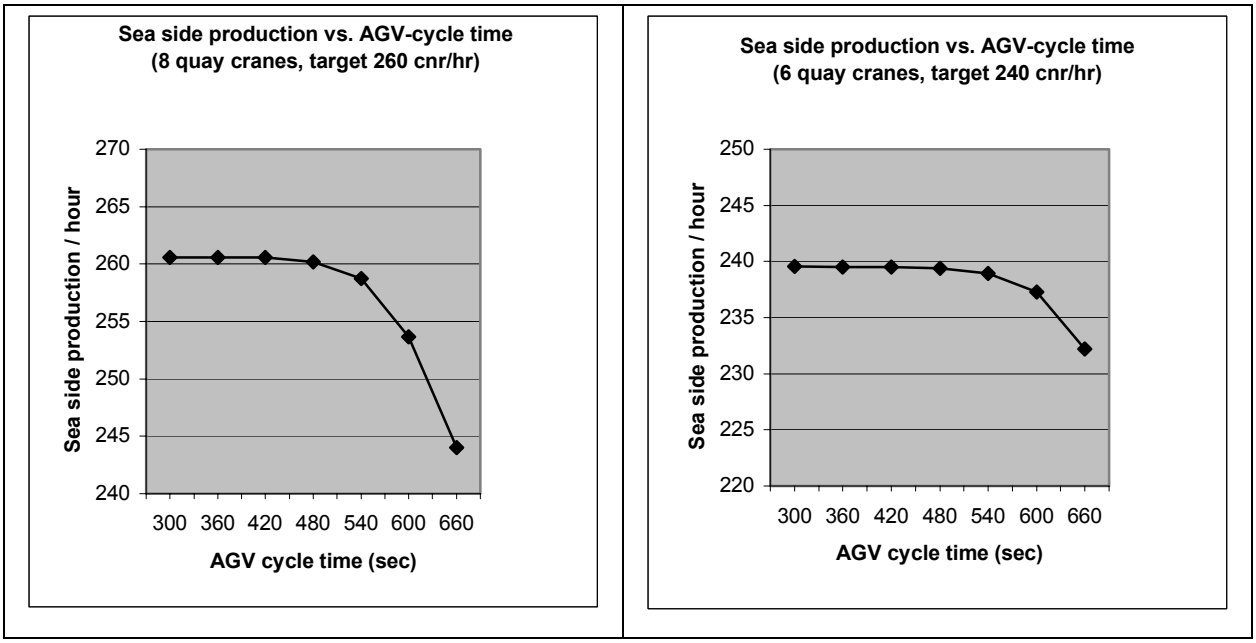


Figure 7.9: *Sea side production as a function of AGV cycle times*

Figure 7.9. shows that a maximum technical cycle time of 480 seconds is allowed for both an 8 quaycrane and a 6 quay crane operation. A longer cycle time will make the AGV-system the bottleneck of the system and decrease the resulting productivity.

In the next experiments the ASC cycle time (including transfer time) is gradually increased. The cycle times tested are 60, 120, 180 and 240 seconds. This will also influence the result of the AGV-system, because AGV's now may be waiting for an ASC. For that reason the experiments are performed with AGV cycle times of 360, 420 and 480 seconds. The results are shown in figure 7.10.

From figure 7.10. it is concluded that the required production is still feasible if the average ASC cycle time is around 120 seconds and the AGV cycle time less than 420 seconds. An ASC cycle time of 180 seconds combined with this AGV cycle time can be acceptable if process

control is able to use accidental advantageous circumstances (for example by a partly overlap of the ASC move function and the AGV transport function. In that case the operational cycle time of ASC's in fact decreases).

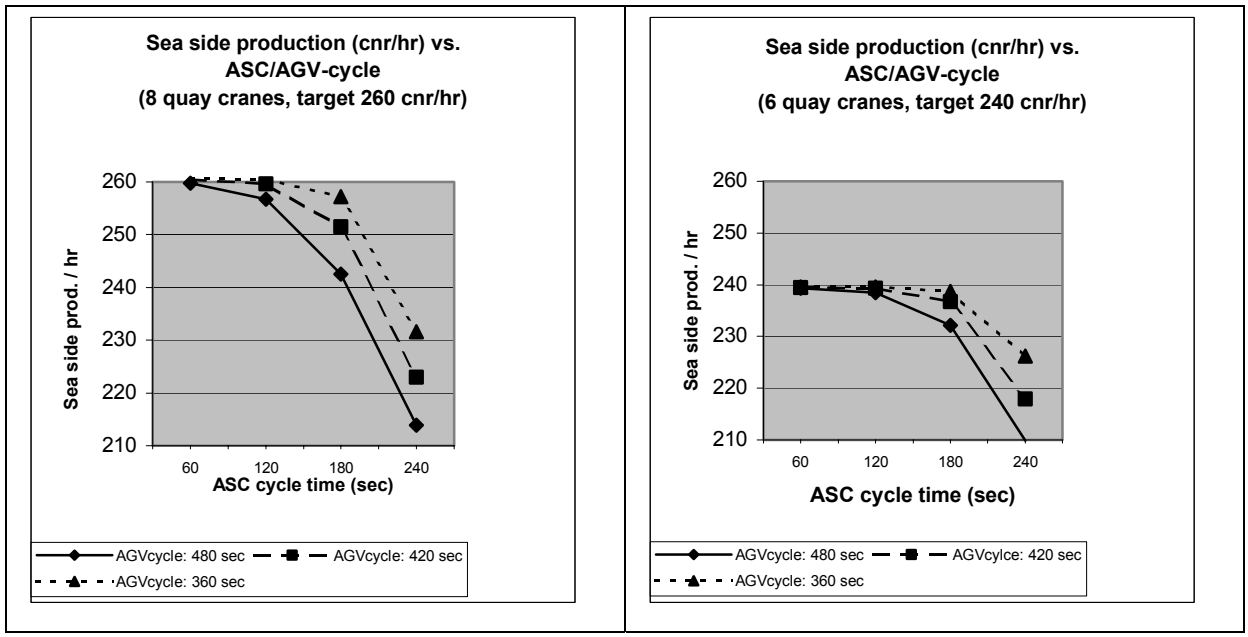


Figure 7.10: *Sea side production as a function of AGV and ASC cycle times*

Reviewing the control alternatives derived from table 7.2. the following 6 experiments were defined:

Experiment A: Uniform distributed selection of ASC's
A1. No selection of QC by AGV A2. Selection of QC by AGV at arrival at the quay A3. Selection of QC by AGV at departure from ASC
Experiment B: selection of ASC's based on work load
B1. No selection of QC by AGV B2. Selection of QC by AGV at arrival B3. Selection of QC by AGV at departure from ASC

Table 7.3. Experiment scheme

Experiment A1 has already been executed; it will serve as a reference experiment, and the other experiments will be executed with the following average cycle times: 180 seconds for an ASC and 360 seconds for an AGV. The results are shown in figure 7.11.

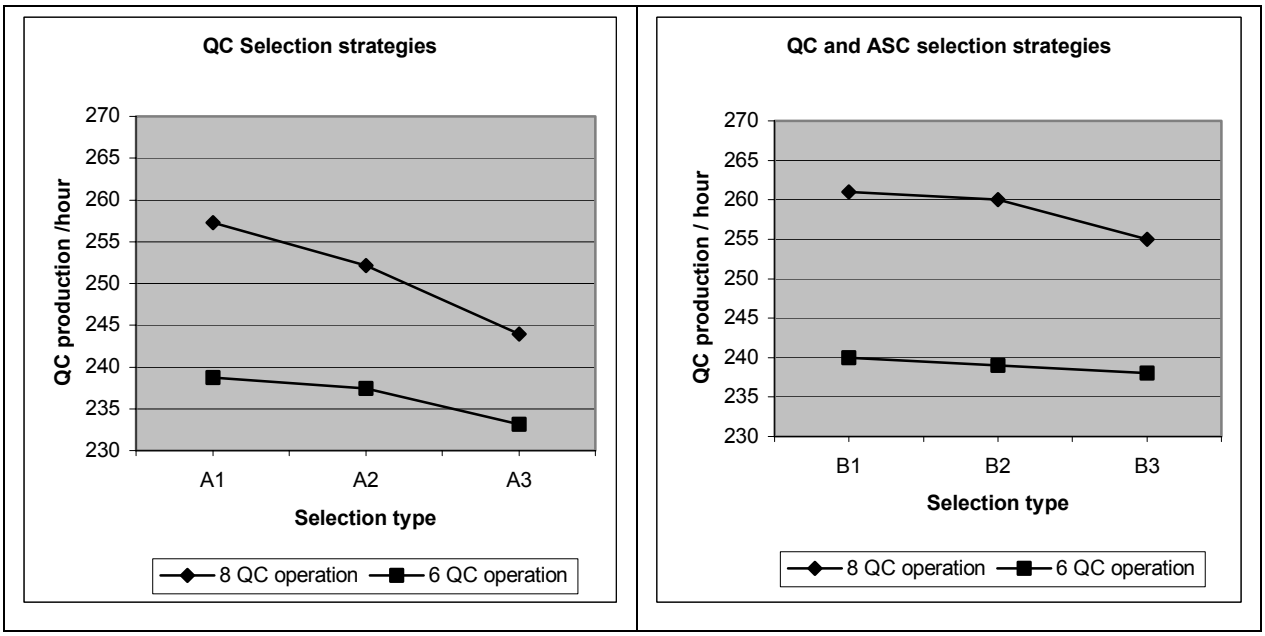


Figure 7.11: *The effect of QC and ASC selection strategies*

With an ASC selection based on the workload the graphs show that sea side productivity is less sensitive for the moment of QC selection. This workload based ASC selection however is only possible for an unloading operation. During a loading operation the loading sequence determines both the ASC and QC to be selected. Task scheduling can only influence the selection effectiveness by selecting a proper quay crane. But quay crane selection has to be done then even earlier than in the cases A3 and B3: at the start or during the move function of an AGV. To get an impression of the effects a final experiment has been executed where experiment A3 was adapted for the loading operation in such a way, that AGV's select a quay crane at the moment of departure from a quay crane. The experiments show that the production decreases to 235 and 227 cnr / NOH for an operation with 8 and 6 quay cranes respectively. Further decrease should be expected because of sequencing problems.

7.5.3. Results

The conclusions from the preceding experiments are (given the results-in-view for net operation hours in the project program):

- The AGV system should be capable of delivering an average "technical" AGV cycle time around 360 seconds, including transfer times at the QC's and the ASC's.
- The ASC system should be capable of delivering an average ASC cycle time around 180 seconds, including transfer times at the transfer positions.

- The required productions are assumed to be productions of unloading operations.
- The cycle times of the quay cranes are assumed to be technical cycle times, including transfer times.
- An ASC selection strategy based on workload has a positive effect on production and should be used to make the requirements feasible.

7.5.4. Using the model during the project

The process descriptions of table 7.2. form a proper basis for discussing these requirements at an interdisciplinary level. The requirements can be translated directly into a TOMAS simulation model. With this approach the intended results can be quantified up to the function level and they should be used by the project groups as objectives for the detailed design. For example, the AGV cycle times are objectives for the project groups developing the AGV's and the process control. If a (technological) driving speed of AGV's is decided for during some phase of design, then this automatically leads to a maximum allowed distance per AGV cycle, which then becomes an objective for process control. The same conclusion holds for the ASC cycle times. If at some moment during process design these requirements appear to be not feasible then the consequences for other groups should be evaluated by means of this model.

In reality at some moment the technical specification of AGV's defined a maximum driving speed of 3 m/sec straight forward and of 2 m/sec in curves. At that moment no traffic routing mechanism was decided on yet and looking at the layout of the quay area there was no reason to suspect that the cycle time of 360 sec would not be feasible. In a later phase however the routing algorithm was defined and showed long driving distances with usually six 90° curves for each cycle. In reality the measured maximum sea side production of terminal operations was around 210 containers/GOH for unloading operations.

If the technical data of routing distances and AGV speeds would have been applied to the global TOMAS model, this production could have been predicted. A raw calculation shows that the technical cycle time would be around 420 seconds and traffic delays were not included yet. With this knowledge the experiment B3 could have been executed again with an average AGV cycle time of 480 seconds. The results would have shown a sea side

production of 228 cnr/NOH with 6 quay cranes. If the condition of 22 NOH per day of 24 hours is applied, the measured 210 cnr/GOH match the simulation results.

During the project several detailed models were developed, for example an AGV traffic control model, an ASC system model and a Straddle Carrier model. They all worked stand alone, with assumed inputs at their system borders. Nowadays it is possible to connect each of these models to the global TOMAS model described above by means of the 'send' and 'receive' mechanism of chapter 5. By using the object oriented approach in connection with the process interaction approach, an object class (for example the AGV- or ASC-system) can be extracted from the global TOMAS model and can be implemented to any degree of detail. Each of these models makes use of the input flows as they are generated by the global model. In this way the results of detailed designs can be judged with respect to the high level requirements.

Finally, the same approach can be used to test the real system with respect to control algorithms (task scheduling, traffic control) and equipment.

7.6. Conclusions

In this chapter the interdisciplinary use of the PROPER model and process descriptions are illustrated in a complex design project and applied to the results of the function design phase (as they are formulated in the project program).

It has been shown that the PROPER model structures the theoretical productivity figures as presented in the project program. It thereby relates the figures of service, operation and technology to each other and to the overall terminal figures. It enables the definition of results in view for all project groups starting with process design. By using the PROPER model missing figures are detected and the suitability of other figures is reviewed; for example all intended results for shifts are not defined and no availability requirements were formulated.

It is also shown that time dependent behavior can be studied at a global level of detail, helping to quantify defined objectives and to derive objectives for service, transfer and resource use. The global descriptions can be detailed further according the function structure of the PROPER model. By doing so, the original requirements and product and order flow

characteristics are preserved, and adjustments can be made as soon as original objectives become infeasible.

In reality this systematic analysis and simulation approach was not applied; driving speeds were decided for on a technological basis only, and Simulations and Process Control were only trying to maximize production. The AGV traffic control had to be developed from scratch. Ideas started with so-called "free-ranging" AGV's (with short driving distances) and it took some time before a strict (and apparently non-optimal) routing control for AGV's was decided for. This routing control resulted in driving distances that –in combination with the driving speed- could not fulfill the requirements.
As a consequence, the real production stayed behind the requirements as formulated in the project program, mainly because the expectations were not realistic.

It is not proven that the use of the approach of this thesis would have improved the real results. It has been shown however that the use of approach of this thesis would have clarified in an early stage, what could be expected and proper measurements could have been taken to improve it.

Chapter 8

Conclusions and future research

"The important thing is not to stop questioning."
-Albert Einstein-

In the introduction to this thesis two questions were asked:

- "How can the concept of systems thinking be successfully used to construct a generic conceptual model for the design of logistic systems?"
- "How can the behavior of a logistic system be described in a conceptual interdisciplinary way and how can it be implemented in a software environment?"

The basic idea behind the questions was to enable communication, understanding and cooperation between different disciplines involved in the design of logistic systems in order to improve the correspondence between expectations and reality of the system to be designed.

As a starting point for development figure 8.1 was used, where conceptual modeling was applied to construct a simulation model.

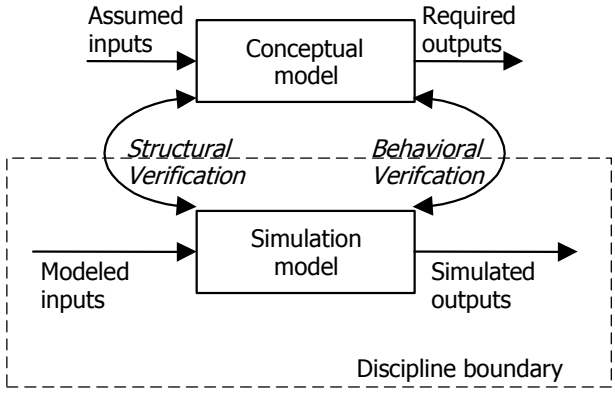


Figure 8.1: *Verification in the simulation of innovative systems*

It was stated that all disciplines use their own way of (conceptual) modeling a logistic system, based on a monodisciplinary perception and using different types of elements and

relations. As a consequence the verification of models has become a part of each discipline itself.

To unify these modeling approaches, the conceptual PROcess-PERformance (PROPER) model is developed. This model is an interdisciplinary model and presents a logistic system as a hierarchical structure of functions and processes, derived from the primary objectives of the system. A logistic system itself fulfills a function in its environment and all of its elements contribute to this function. In this way all elements are functions on their own, reflecting their objective. In order to judge the 'quality of functioning', performance criteria are defined in terms of effectiveness, efficiency and productivity. Each function is composed of a control function and a process, by which input flows are transformed into output flows. The major disciplines Information Technology, Organization (and Logistics) and Technology are reflected by the three flows: order flow, product flow and resource flow.

The model is compared to concepts in logistic practice. It is shown that the general Supply Chain Operations Reference (SCOR) model can be modeled by means of the PROPER model, so all existing logistic structures are covered by the model. The PROPER model however does not restrict itself to existing structures. By projecting the major fields of logistics (i.e. Purchase Logistics, Production Logistics and Physical Distribution) to the PROPER model, it is found that the field of Logistics is not flow oriented in terms of the PROPER model, but rather function oriented. The fields do cover the three flows of the model, but each of the fields deals with a specific functions in these flows only. There are no logistic approaches yet dealing with all functions of one flow completely. Further research is required to show if concepts like order flow logistics, product flow logistics or resource flow logistics can contribute to the overall field of logistics.

To position the use of the model in the design process two phases of design were distinguished:

- Function design, where the required functions (including their performance) are determined, and alternative structures are investigated to decide for a 'best feasible' configuration of functions.
- Process design, where each discipline translates the function requirements to physical designs.

The goal of function design is to determine the productivity in view. By interaction with process design the standard productivity for each design part is to be determined. In this way the expectations of each part of the system are made clear for the disciplines mentioned. Decisions made during function design are registered in terms of the PROPER and by doing so the model becomes the basis for creating 'shared memory', which can be used in future design projects.

The transition from function design to process design marks the border between interdisciplinary and mono- or multidisciplinary design steps. All disciplines will translate the required functions to their specific conceptual design methods.

The conclusion is that for conceptual modeling of a logistic system in an interdisciplinary way, the following requirements should be satisfied:

- a system should be defined in terms of its "function" to be fulfilled: define "why" activities inside the system are required.
- To fulfill the function the system is divided in partial functions contributing to the function of the system as whole. The elements of a system are therefore functions again and thus systems on their own.
- Within each function a clear distinction should be made between a control function and an operational function.
- With respect to the function of the system as a whole three interrelated aspects should be considered: orders, products and resources. If one of them is missing, the function of the system cannot be fulfilled.
- Each function is realized by a "process", a series of activities transforming input into output. The combination function – process is the basis for each single discipline to design the physical system.

The PROPER model represents a static structure by its nature and therefore only the structural verification is supported by it. In order to support design decisions, the time dependent behavior of the system should be modeled too and become a part of the function design phase. To preserve the interdisciplinary character (and thus supporting behavioral verification in an interdisciplinary way), the modeling of behavior is described in terms of natural language that can be read and understood by all disciplines. These linguistic descriptions are structured according the process structure of the PROPER model and are

directly connected to the functions in this way. Each function (or group of functions) has its own process description. Common attributes of functions and the process interactions between functions are defined to support the description of synchronous behavior. A process description is a sequence of events (moments in time); at each event the change of attribute values is described. Between two successive events a duration of time is defined by a "time consuming" expression. By this approach the attribute "process state" is introduced, which can take the values 'active', 'suspended', 'conditioned' or 'scheduled'. Process interactions introduce new events or interrupt the sequence of a process. A limited set of terms is required to define the type of interaction.

A process description is used to formulate design questions that cannot be derived from the static PROPER model alone. Answers to these questions can be found by using computer simulation. For this purpose the linguistic process descriptions should be translated in an easy way to a simulation software environment. It is shown that an object oriented modeling approach results in a semi-formal Process Description Language PDL, that is ready to be implemented in computer software. Preprogrammed simulation environments (so-called 'click-and-play' packages) however do not match the process interaction approach required to translate PDL descriptions directly. Also simulation languages do not fully comply to the requirements of PDL. A new simulation language TOMAS (Tool for Object oriented Modeling And Simulation) is developed that is completely based on process interaction and above that supports hierarchical modeling by means of a simple 'send' and 'receive' concept. The implementation of process interaction required an intervention in the usual sequential order of statement execution.

PDL and TOMAS are restricted to discrete systems. Attributes that change continuously cannot be modeled at this moment. For future applications it may become necessary to implement methods for this type of attributes.

The conclusions with respect to the conceptual modeling of time dependent behavior are:

- natural language is an interdisciplinary communication "tool" with rich expressions for describing time dependent behavior.
- By using natural language the verification of behavior descriptions becomes an interdisciplinary task.

- By adding a notion of "system time" processes can be described as being executed in parallel.
- The time notion results in a limited number of properties for a process definition, describing state, state transition and process interaction.
- Function and process descriptions can be modeled with an object oriented approach. This results in an unambiguous translation for software implementation.
- Implementation in (single processor based) software requires a mechanism that breaks through the traditional sequential execution of statements.
- For multi processor implementation the model is divided according the functions of the conceptual model and a send and receive mechanism for communication between functions is provided.
- Communication between functions at different aggregation strata is supported by the same send and receive mechanism.
- For design purposes the "system time" should be implemented in a multi processor environment in two ways.
    - System time is implemented as if the distributed model runs in a single processor environment. In this way reproducibility of simulation experiments is preserved.
    - System time is synchronized to the wall clock time. In this way real resources and real control algorithms can be tested in a virtual environment.

To illustrate the use of the approach a complex project for developing the first automated container terminal has been described. It is shown that the use of PROPER, PDL and TOMAS would have lead to a better definition of requirements and a better understanding of the terminal processes. The unexpected behavior and performance after the start of the real terminal operations would have been revealed by this approach in an early stage of the design project.

During the last two years the approach has been applied in a new research project FAMAS.MV2 for the intended extension of the Rotterdam port area called 'Maasvlakte 2' [Connekt, 2002], [Boer et al., 2002]. A first global model of the complete area has been constructed in TOMAS to support decisions on alternative layouts and use of the area. This model will be a reference model (and even a "simulated environment") for future detailed simulation models. Many simulation models are expected at different aggregation strata in the course of the project and to support a smooth interaction and connection between these

model a "backbone" structure was developed based on the TOMAS implementation of distributed simulation [Veeke, Ottjes, 2002] .

The use of the approach is not restricted to logistic systems. As shown in chapter 2, the modeling with function elements generally applies to purposive systems. Production systems and service systems are purposive systems too, and can therefore be modeled with this approach also. For the primary function at the level of the system as a whole, the same three flows should be distinguished.

In this thesis the design of control functions has not been described in detail. Creating a control structure is considered part of process design for the organizational discipline. Creating control algorithms is considered part of information technology and operations research. However, it has already been shown that control functions can be implemented in TOMAS models in the same object oriented way as "physical" processes (see for example [Ottjes, Veeke, 2002a] and [Ottjes, Veeke, 2002b]).

Finally, the approach of conceptual modeling in this thesis started with a basic systems approach in order to create a conceptual model of a logistic system. In this respect it differs from existing approaches that usually start from the mono disciplinary viewpoint. Especially in a technology oriented environment these approaches tend to use a hard systems approach, which is basically solution oriented. Starting with a soft systems approach adds the 'problem formulation' phase explicitly, without any restriction to specific disciplines. By this, conceptual modeling becomes an interdisciplinary task. In this thesis the approach focuses on three aspects: orders, products and resources. The human, environmental and financial aspects of logistic systems have not been investigated here. It is a great challenge to extend this research with these aspects.

References

Ackoff, R.L., 1961, "*Systems, Organizations and Interdisciplinary Research*", Systems Research and Design, ed. D.P. Eckman., Wiley, New York, pp. 26-42.

Ackoff, R.L., 1971, "*Towards a System of Systems Concepts*", Management Science, 17(11), pp. 661-671.

Beer S., 1985, "*Diagnosing the system for organisations*", John Wiley and Sons Ltd., Chichester, England

Bertalanffy, L. von, 1949, "*Zu einer allgemeinen Systemlehre*", Biologia Generalis, 195, pp. 114-129

Bertalanffy, L. von, 1973, "*General Systems Theory – Foundations, Development, Applications*", Harmondsworth, Penguin Books

Bikker, H.,1995, "*Organisatie, hoeksteen voor innovatie*", inaugural speech (in Dutch), Delft University of Technology.

Bikker, H., 2000, "*Analysis and Design of Production Organisations*", Lecture Notes, Delft University of Technology.

Boer C.A., Veeke H.P.M., Verbraeck, A., 2002, "*Distributed simulation of complex systems: Application in container handling"*, SISO European Interoperability Workshop, paper nr. 02E-SIW-029.pdf, Harrow, Middelsex UK

Booch, G., 1994, "*Object-Oriented Analysis and Design with Applications*", 2nd ed. Redwood City, The Benjamin Cummings Publishing Company Inc., California

Burback, R, 1998, "*Software engineering methodology: the Watersluice*", dissertation of Stanford University

CACI, 1996, *"MODSIM III The Language for Object-Oriented Programming*", Reference Manual

CACI, 1997, "*SIMSCRIPT II.5 Reference Handbook",* C.A.C.I, 2nd edition.

Cardelli, L., Wegner, P., 1985, "*On Understanding Types, Data Abstraction, and Polymorphism*", ACM Computing Surveys vol. 17 (4).

Chakravarti, Laha, and Roy, 1967, "*Handbook of Methods of Applied Statistics",* Volume I, John Wiley and Sons, pp. 392-394.

Checkland, P., 1981, "*Systems Thinking, Systems Practice*", John Wiley & Sons Ltd, New York, ISBN 0 471 27911 0

Connekt, 2002, "Maasvlakte Integrale Container Logistiek", Final Report FAMAS.MV2 – Project 2.1, ISBN 90 442 00097

Courtois, P., 1985, "*On Time and Space Decomposition of Complex Structures",* Communications of the ACM, vol. 28(6), p.596.

Christopher, M., 1998, "*Logistics and supply Chain management. Strategies for reducing costs and improving services*", London, FT Prentice Hall

Daellenbach, H.G., 2002,"*Hard OR, Soft OR, Problem Structuring Methods, Critical Systems Thinking:A Primer*", report University of Canterbury, Christchurch, New Zealand.

Damen, J.T.W., 2001, "*Service Controlled Agile Logistics*", Logistics Information Management, Vol. 14, nr.3, MCB Press, United Kingdom

Davenport, T., 1993, "*Process Innovation: Reengineering Work through Information Technology*", Harvard Business School Press, Boston

Doepker, P.E.,2001, "*Integrating the product realization process into the design curriculum*", Design Education for the 21 th Century, Volume 17, number 4/5, pp. 370-374

Dwyer, M.W., 1998, *the Entelics Research website*: http://www.entelics.com/cognitive.html

ECT, 1988, **"***Project programma MSS, ECT-Sea-Land Deltat Terminal***"**, internal report MSS 115, Europe Combined Terminals

Flood, Robert L., Jackson, Michael C., 1992, "*Critical Systems Thinking*", John Wiley & Sons, Chichester, England, ISBN 0-471-93098-9

Fujii, S. et al., 1999, "*Synchronization Mechanisms for Integration of Distributed Manufacturing Simulation Systems*", Simulation 72:3, pp.187-197, ISBN 0037-5497/99

Galatescu, A., 2002, "*Reifying Model Integration Abilities from Natural Language*", Journal of Conceptual Modeling, Issue 24

Gall, J., 1986, "*Systemantics: How Systems Really Work and How They Fail***"**, 2nd ed. Ann Arbor, MI: General Systemantics Press.

Gans, O.B. de, 1999, "*PROSIM Modeling Language Tutorial",* 3d edition, PROSIM B.V, Zoetermeer

Gheluwe van, 2000, "*Multi-formalism modelling and simulation*", D.Sc. Thesis, Faculty of Sciences, Ghent University

Grover, V., Jeong, S.R., Kettinger, W.J. & Teng, J.T.C. ,1995, "*The Implementation of Business Process Reengineering*", Journal of Management Information Systems, 12(1), pp. 109-144.

Ham, R.Th van der, 1992, "*Must Simulation Software*", User and Reference Manual version 5.50, Upward Systems, Rijswijk

Holweg, M., 2001, "*Systems Methodology in manufacturing and Logistics Research, A Critical Review*", Lean Enterprise Research Centre, Cardiff Business School, United Kingdom

Jackson M. C., 1991, "*Systems Methodology for the Management Sciences*", Plenum Press, USA.

Jacobson, I., Booch, G., Rumbaugh, J., 1999, "*The Unified Software Development Process*", Addison-Wesley Publishing Company.

Jonas, W., 1997, "*Viable structures and Generative tools – an approach towards designing design*", The European Academy of Design, Stockholm, 1997.

Kiniry, J.R., 1998, "*The Specification of Dynamic Distributed Component Systems*", M.S. Thesis, California Institute of Technology, Pasadena, CA.

Kirkerud, B., 1989, "*Object-Oriented Programming with SIMULA",* Addison-Wesley

Kirkwood, C.W., 2002 , "*System Dynamics*", website of Arizona State University: *http://www.public.asu.edu/~kirkwood/sysdyn/SDRes.htm*

Konda, S., Monarch, I., Sargent, P., Subrahmanian, E.,1992, "*Shared Memory in Design: A unifying Theme for Research and Practice*", Research in Engineering Design, 4(1), p.23-42

Kramer, N.J.T.A., Smit, J. de, 1991, "*Systeemdenken*", Stenfert Kroese, ISBN 90 207 2008 2

Lambert, D. M., 1997, "*Fundamentals of Logistics Management*", The Irwin/McGraw-Hill series in Marketing

Laws, A.*,* 1996, *"Soft Systems Methodology",* course notes Business Systems Analysis CMSCB3001, School of Computing & Mathematical Sciences, John Moores University, Liverpool, United Kingdom

de Leeuw, A.C.J., 1982, *"Organisaties: management, analyse, ontwerp en verandering",* Assen, Van Gorcum.

Little, J.D.C., 1961, "*A proof of the formula L = λ * W*", Operations Research, Vol. 9, pp. 383-387

Lodewijks, G., 1991, "*Object geörienteerd programmeren in Turbo Pascal en C++*", computer assignment (in Dutch), Delft University of Technology, report 91.3.TT.2886

Lodewijks, G., 1996, "*Dynamics of Belt systems*", thesis Delft University of Technology, ISBN 90-370-0145-9.

Macaulay L. A., 1996, "*Requirements Engineering*", Springer-Verlag London Ltd.

McGinnis, L.F., 2001, "*Industrial Logistics Systems Design: A Conceptual and Computational Foundation*", Keck Virtual Factory Lab, School of Industrial and Systems Engineering, Georgia Institute of Technology.

Meyer, B., 1988, "*Object-Oriented Software Construction*", Prentice Hall.

Ottjes, J.A., Hogendoorn, F.P.A., 1996, "*Design and Control of Multi-AGV Systems, Reuse of simulation software*", Proceedings of the 8th European Simulation Symposium (ESS 1996), Genua, ISBN 1-56555-099-4

Ottjes, J.A., Veeke, H.P.M., 2000a,"*Production Scheduling of Complex Jobs with Simulation*", Proceedings of the Business and Industry Simulation Symposium (ASTC 2000), Washington D.C., ISBN 1-56555-199-0

Ottjes, J.A., Veeke, H.P.M., 2000b, "*Project Management with Simulation: A critical view on the critical path*", Proceedings of the ICSC Symposia on Intelligent Systems & Applications (ISA 2000),Wollongong, Australia, ISBN 3-906454-24-X

Pahl, G., Beitz, W.,1996, "*Engineering design: A Systematic Approach*", 2nd edition, Springer, London, England

Pugh, S., 1990, "*Total Design*", Addison-Wesley Publishing Co.

Rosenblueth, A., Wiener, N. , Bigelow, J., 1943, "*Behaviour, purpose and teleology*", Philosophical Science 10(1), pp.18-24.

Rosnay, J. de, 1988, "*Macroscoop*", L.J.Veen B.V. ISBN 90 204 1864 5

Salvendy, G. (ed.),1982, "*Handbook of industrial engineering*", Purdue University, John Wiley & sons

SCC, 2002, *The Supply Chain Council* website: www.supply-chain.org

Shlaer, S., Mellor,S., 1988, "*Object-Oriented Systems Analysis: Modeling the World in Data*", Prentice-Hall, Inc

Simon, H., 1982, "*The Sciences of the Artificial*", Cambridge, MA : MIT Press.

Stewart G., 1997, "*Supply chain operations Reference model (SCOR)*", Logistics Information Management, vol. 10, number 2.

Strachey, C., 1967, "*Fundamental Concepts in programming languages*", Lecture Notes for International Summer School in Computer Programming, Copenhagen

Swanstrom, E., 1998, "*Creating Agile Organizations with the OOCL Method*", John Wiley & Sons, Inc.

Vaan, M.J.M. de, 1991, "*Business Logistics, Logistiek voor ondernemers en managers*", Kluwer Bedrijfswetenschappen, ISBN 90-267-1756-3

Veeke, H.P.M.,1982, "*Process simulation as a management tool*", Proceedings of the IASTED International Symposium Applied Modeling and Simulation, Paris.

Veeke, H.P.M., Ottjes J.A., 2000, "*Objective Oriented Modeling*", Proceedings of Industrial Systems 2000 (IS 2000), Wollongong, Australia, The International Computer Science Convention ICSC.

Veeke, H.P.M., Ottjes, J.A., 2001*,"Applied distributed discrete process simulation*", Proceedings of the 15th European Simulation Multiconference (ESM 2001), Prague, ISBN 1-56555-225-3

Veeke, H.P.M., et al., 2002, "A simulation architecture for complex design projects", Proceedings of the 14th European Simulation Symposium (ESS 2002), Dresden, ISBN 3-936150-21-4

Veld, J. in 't, 2002, "*Analyse van organisatieproblemen, een toepassing van denken in systemen en processen*", 8th edition, Stenfert Kroese, ISBN 90-207-3065-7

Visser, H.M., Goor, A.R., 1996,"*Werken met logistiek*", Educatieve Partners Nederland, ISBN 90-11-05356-7

Whitmore, R., 1998, "*Systems Applications in the Theory and Practice of Resource Development: A Review of the Literature*", Department of Resource Development, Michigan State University, Occasional Paper RDOP-98-1.

Wiener, N., 1948, "*Cybernetics*", MIT Press, NewYork.

Wigal, C.M., 2002, "*Utilizing Systems Thinking to Improve Problem Solving Practices*", technical paper ASEE SE section Annual Conference, Florida

Williams, B., 2002, "*Evaluation and Systems Thinking*", work in progress on website 'users.actrix.co.nz/bobwill'

Zeigler, B.P., 1976, "*Theory of Modeling and Simulation"*, John Wiley and Sons, New York

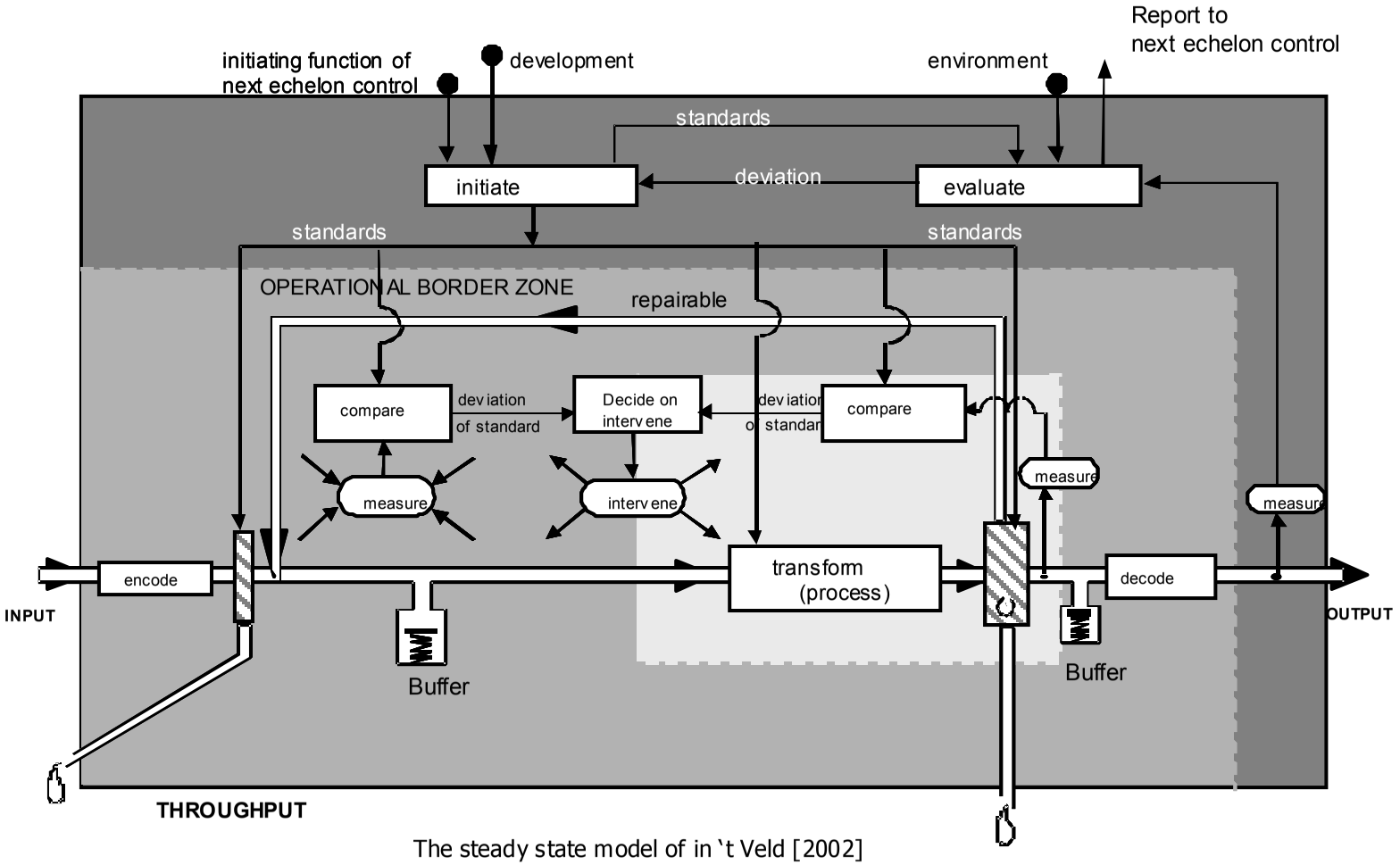
Zeigler, B.P., Praehofer, H., Kim, T.G., 2000, "*Theory of Modeling and Simulation"*, New York, Academic Press

List of abbreviations

AGV = Automatic Guided Vehicle
ASC = Automatic Stacking Crane
BPR = Business Process Redesign
DESS = Differential Equation Specification System
DEVS = Discrete Event System Specification
DSL = Delta Sea-Land
DTSS = Discrete Time System Specification
ECT = Europe Combined Terminals
FSM = Formal System Model
GOH = Gross Operational Hour
GST = General Systems Theory
IP = Instruction Pointer
MIT = Moment In Time
MSS = Majority Sea to Sea
MTS = Multi Trailer System
NOH = Net Operational Hour
OOA = Object Oriented Analysis
OOC = Object Oriented Computing
OOD = Object Oriented Design
OOP = Object Oriented Programming
PDL = Process Description Language
PDS = Product Design Specification
PROPER = PROcess-PERformance
QC = Quay Crane
SC = Straddle Carrier
SCC = Supply Chain Council
SCM = Supply Chain Management
SCOR = Supply Chain Operations Reference
SSM = Soft Systems Methodology
TCP/IP = Transmission Control Protocol / Internet Protocol
TOMAS = Tool for Object oriented Modeling And Simulation
VSM = Viable System Model

Appendix A

The steady state model



The steady state model of in 't Veld [2002]

Appendix B

A multi component DEVS

The Generator

A generator has no inputs, but when started in phase "active" it generates outputs with a specific period. The generator has two basic state variables: phase and **σ**. The generator remains in the phase "active" for the period, after which the output function generates a "job" for output and the internal transition function resets the phase to "active" and **σ** to "period".

$$DEVS_{generator} = <X, S, Y, \delta_{int}, \delta_{ext}, \lambda, P>$$

where

$X = \varnothing$

S = {"passive", "active"}

Y = {job} *(processing time)*

δ_{int}(phase, **σ**) = ("active", period)

λ("active", **σ**) = job

P(phase, **σ**) = **σ**

The processor

The generator is connected to a processor, which results in a model of a simple workflow situation. The processor takes some time to do the jobs of the generator. In this simple case only the times to do the jobs are represented.

$$DEVS_{processor} = <X, S, Y, \delta_{int}, \delta_{ext}, \lambda, P>$$

where

X = {job} *(processing time)*

S = {"idle", "busy"}

Y = {job} *(processing time)*

δ_{int}(phase, **σ**, job) = ("idle", processing time, job)

δ_{ext}(phase, **σ**, job, e , x) =

("busy", processing time, job) if phase = "idle"

(phase, **σ** - e, x) otherwise

λ("busy", **σ**, job) = job

P(phase, **σ**, job) = **σ**

If the DEVS is passive (idle) when it receives a job, it stores the job and goes into phase "busy". If it is already busy it ignores incoming jobs. After a period "processing time" it outputs the job and returns to "passive".

Appendix C

TomasProcess

A kernel module for discrete process simulation

1. Introduction

TOMASPROCESS contains the basic services for the process oriented discrete event simulation.

The description is divided into three parts:

1. Services for process control
2. The class TomasProcessElement
3. General supporting routines.

A precompiled version is available for Delphi 5 (TomasProcess.dc5), Delphi 6 (TomasProcess.dc6) and Delphi 7 (TomasProcess.dc7).

TomasProcess can be included in any user-built simulation environment. In this platform an example of such a platform is given by the TOMAS source code, which offers added functionality such as user friendly simulation control, queues and distributions.

2. Services for process control

TOMASPROCESS uses a time-axis to sort events. At any moment during a simulation always one and only one process is the current process. When a process reaches a point where the status of the process doesn't change for some amount of time, the process needs a service to communicate this to the sequencing mechanism. The following services are available to do this (the current time of the simulation is assumed to be Now).

Procedure HoldProcess(T: Double);
: this service suspends the process for T time units. The process will become current again at Now + T.

Procedure StandByProcess;
: StandbyProcess suspends the process until some condition is met.
The usual programming structure to achieve this is:
While Condition = FALSE **Do** StandbyProcess;

Procedure SuspendProcess;
: This service suspends the process for indefinite time. Some external process is required to make this process current again (see Scheduleprocess of TomasProcessElement).

Procedure FinishProcess;
: By this service the current process is finished. The owner of the process (a TomasProcessElement) remains present in the system.

Procedure FinishProcessAndDestroy;
: FinishProcessAndDestroy finishes the current process and destroys the owner of the process.

Because the sequencing mechanism runs in a user environment, where interaction or intermediate results are required, we need to have services for controlling the sequencing mechanism itself. This can be done with:

Procedure StartProcessSimulation;

This service gives control to the sequencing mechanism and starts the simulation with a process being scheduled there.

Procedure InterruptProcessSimulation;

By this service you are able to interrupt the simulation (i.e. stopping the simulation clock) and do whatever is needed in your own environment.

Procedure ResumeProcessSimulation;

ResumeProcessSimulation resumes the simulation after it has been interrupted.

A special service can be defined to take actions whenever the simulation state changes.
For this purpose the following service is available:

Procedure SetStatusChange(ChangeProc: ChangeProcedure);

Specify here which procedure should be called whenever the state changes. In this procedure you can decide on what to show on the screen, when to interrupt the simulation etc.
In your statuschange procedure you can check what kind of change takes place. To investigate this the following functions are available:

Function TracingMessage: Boolean returns TRUE if there is a message useful for tracing purposes.
Function TraceLine: String returns the tracing message. Typical trace messages from the TomasProcess environment concern the change of current process and status of connections with a server-program.
Function ServerQuestion: Boolean to show there is a question from a Server program.
Function ServerMessage: Boolean to show there is a message from a Server Program.
Se also chapter 4 of the general supporting routines

Finally the status of the simulation itself is specified by the type 'SimulationStatus', which can take the following values:

S_INIT:	simulation not started yet.
S_FINISHED:	simulation has finished
S_INTERRUPTED:	simulation is interrupted
S_RUNNNG:	simulation is running
S_STEPRUNNING:	simulation is running but will be interrupted after each event.

The SimulationStatus can be retrieved by Function GetSimulationStatus: SimulationStatus;

3. The class TomasProcessElement

Every process is owned by a TomasProcessElement. This general class contains all services to inform the sequencing mechanism of it's presence and it's process description.
The class has the following properties and services:

Name: The name of the element
Status: The current status of the element.
This field is of type ElementStatus and can take the following values

TC_DATA:	The element is inactive
TC_PASSIVE:	The element is suspended
TC_INTERRUPTED:	The element is interrupted
TC_TIMESCHEDULED:	The element is scheduled for a specific moment
TC_STATESCHEDULED:	The element is scheduled for a condition
TC_CURRENT:	The element is the current process.

ArrivalTime: The creation time of the element
EventTime: The next event time of the element's process. Returns NEVER if the element is not scheduled.

Constructor Create(Name: String; CTime: Double);
creates an instance of TomasProcessElement with name Name at time CTime.

Destructor Destroy;
destroys the instance.

Procedure Process;
is a virtual service representing the process description of the element class.
You must override this service if the element owns a process. If you don't override, but still activates the element a warning message is given.

Procedure ScheduleProcess(T: Double);
a service to initiate or resume the process of the element at time T.

Procedure RemoveProcess;
a service to cancel or finish the process when the element is not current.

Procedure InterruptProcess;
A service to interrupt the process. If interrupted the process can be resumed finishing it's current hold or standby period.

Procedure CancelProcess;
A service to cancel the hold, standby of suspended state of the element. To restart the process you need to schedule it again.

Function Scheduled: Boolean;
returns TRUE if the element has a next event time or condition.

4. General supporting routines

To gain control over the sequencing mechanism some additional services have been added. The major goal of the first group of services is to support the verification phase. With these services you can interrupt the simulation or looking at it event by event. The services are:

Procedure SetBreakTime(T: Double);
This service informs the sequencing mechanism that the simulation msut be interrupted at T. If T < Now then this service has no effect.

Function GetBreakTime: Double;
Returns the currently set BreakTime. If no Breaktime is set, it returns -1.

Procedure SetKeyPressed(C: Char);
This service offers a way to interrupt the simulation by pressing the spacebar.
It will be extended to more key characters in future.

Procedure SetStopTime(T: Double);
Informs the sequencing mechanism to stop the simulation and to return to the statement directly following the StartProcessSimulation statement.
If no StopTime is set, the simulation ends only, when there are no events available.

Function GetStopTime: Double;
Returns the currently set StopTime. If no StopTime is set it returns -1.

Procedure StartStepMode;
Forces the sequencing mechanism to wait after each event for a PerformNextEvent call. So after each event the user is able to look in detail to the state changes.

Procedure StopStepMode;
Disables the StepMode.

Procedure PerformNextEvent;
Tells the sequencing mechanism to perform the next event on the event chain.

Procedure ResetTomasProcess;
Clears the sequencing mechanism, closes a connection with a server-program (if present) and resets time to zero.

Procedure SetProcessingInterval(Pint: Integer);
By default TomasProcess checks every event if there are Windows messages to be processed (otherwise interaction with the application would be impossible). This checking however takes about 50% of the time needed by the sequencing mechanism. In cases with thousands of events you can speed up the model by decreasing this amount of checking. Just specify the number of events after which TomasProcess should check for Windows messages.

The second group of routines provides information on variables of TomasProcess. These are:

Function GetNow: Double;
Returns the current time of the simulation.

Function CurrentProcessElement: TomasProcessElement;
Returns the current TomasProcessElement (and Nil if the simulation isn't running).

Function TracingMessage: Boolean;

Returns TRUE is TomasProcess has prepared a message, which could be useful for the tracing of events. These messages inform the environment about the current element and the connection status (see next group of routines)

Function TraceLine: String;
This function returns the data string, mentioned above, and clears it.

Finally the last group of routines is concerned with connecting models via a **TCP/IP interface**. TomasProcess supports communication between models. There must be a TomasServer at some specified ServerAddress (IP-address). TomasServer synchronizes the different models to one time-axis. TomasProcess knows two types of communication:

- messages; they are sent or received to and from the TomasServer and processing continues.
- Questions; they are sent to the TomasServer and program execution is suspended until an answer is received.

Procedure GoSetModelName(Name: String);
By entering a unique modelname, communication can be made unambiguous.

Procedure SetServer(Srv: String);
This service sets the serveraddress of TomasServer; if no ServerAddress is set, pure local standalone simulation is assumed.
A valid ServerAddress fullfills all conditions of a normal IP-address. If TomasServer is located at the same machine, 'localhost' can be specified (but in general 127.0.0.1 refers also to this machine).

Function GetServer: String;
Returns the ServerAddress, set by SetServer.

Function ServerMessage: Boolean;
Returns TRUE if a message of TomasServer is received. This service will normally be called in the StatusChange procedure.

Function ServerQuestion: Boolean;
Returns TRUE is a question of TomasServer is received. This service will normally be called in the StatusChange procedure.

Procedure QuestionParms(var Question, Source: String; Var ID: integer);
In case of a question received, this procedure fills the field Question with the contents of the question, the field Source with the sender of the question and ID with the unique ID of the question. A reply should provide this ID (see DoSendReplyTo);

Function MessageLine: String;
Returns the message received of TomasServer and clears it. By setting up conventions on the messagecontents a very flexible way of communication can be developed.

Procedure SendModelMessage(Model, Msg: String);
Sends a message 'Msg' to model 'Model'. The layout of the resulting message string will be: '$$tomodel: ' + Model +'//' + Msg + #13#10.

Procedure DoSendQuestionTo(Model: String; Var Question: String);

Sends a question 'Question' to model 'Model' and waits for an answer. The reply is received in the Question field.

Procedure DoSendReplyTo(Model,Reply: String; ID: Integer);
Sends a reply 'Reply' to model 'Model' as an answer to the question with the identity ID.

Procedure DoStartLogging;
Starts the logging of all messages sent and received in a log file. The log file is named to the model name with extension '.log'.

Procedure DoStopLogging;
Finishes the logging of messages.

Starting a simulation run automatically connects to the TomasServer if a server address is specified and disconnects automatically at the end of the simulation. However, many control programs aren't simulation models at all, but only algorithms that are called by a simulation model to make decisions. If a control program is implemented in a separate application, one should be able to connect and disconnect from the server explicitly. Therefore the following services are implemented:

Procedure CheckServer;
Checks if a connection is made; if not it connects to the specified Server.

Procedure DisconnectFromServer;
Disconnects from the Server.

5. The source code of TomasProcess

```
unit TomasProcess;

interface

  Uses
    Classes,
    Windows,
    Forms,
    WSocket,
    SysUtils,
    Dialogs,
    Controls;

Const
  NEVER: Double = 1.7E+307;
  ProcessVersion: String = 'V1.00.5';

Type
  ChangeProcedure = Procedure;
  SimulationStatus =
      (S_INIT,S_FINISHED,S_INTERRUPTED,S_RUNNING,S_STEPRUNNING);
  ElementStatus = (TC_DATA,TC_CLAIMING,TC_PASSIVE,TC_INTERRUPTED,
   TC_TIMESCHEDULED,TC_STATESCHEDULED,TC_CURRENT);
```

```
{TomasProcessElement is the basic class of TomasProcess.
 Active Elements should override the process-service.
 Remember to put this service in the 'published' section
of your Element-definition.}
TomasProcessElement = class
  Private
    FName: String;
    FReturnAddress: Pointer;
    FLocData: Pointer;
    FStatus: ElementStatus;
    FArrivalTime: Double;
    FEventTime: Double;
  Public
    Claims: TList;
    Property Name: String read FName write FName;
    Property Status: ElementStatus read FStatus write FStatus;
    Property Arrivaltime: Double read FArrivalTime write FArrivalTime;
    Property EventTime: Double read FEventTime write FEventTime;
  Published
    Constructor Create(S: String; CTime: Double);
    Destructor Destroy; override;
    Procedure Process; virtual;
    Procedure ScheduleProcess(T: Double);
    Procedure RemoveProcess(Stop: Boolean);
    Procedure InterruptProcess;
    Procedure CancelProcess;
    Procedure CancelResume;
    Function Scheduled: Boolean;
  End;



Procedure HoldProcess(T: Double);
Procedure StandByProcess;
Procedure SuspendProcess;
Procedure FinishProcess;
Procedure FinishProcessAndDestroy;


Procedure StartProcessSimulation;
Procedure ResumeProcessSimulation;
Procedure InterruptProcessSimulation;
Procedure StartStepMode;
Procedure StopStepMode;
Procedure PerformNextEvent;
Function CurrentProcessElement: TomasProcessElement;
Function GetSimulationStatus: SimulationStatus;

Procedure GoSetSimulationSpeed(Speed: Integer);
Procedure SetStatusChange(ChangeProc: ChangeProcedure);
Procedure SetBreakTime(T: Double);
Function GetBreakTime: Double;
Procedure SetStopTime(T: Double);
Function GetStopTime: Double;
Function GetNow: Double;
Function TracingMessage: Boolean;
Function TraceLine: String;
Function CurrentChange: Boolean;
```

```
Procedure ResetTomasProcess;
Procedure DoStartLogging;
Procedure DoStopLogging;
Procedure SetProcessingInterval(PInt: Integer);
Function GetLogging: Boolean;
Procedure SetKeyPressed(C: Char);

Procedure CheckServer;
Procedure DisconnectFromServer;
Function ServerMessage: Boolean;
Function ServerQuestion: Boolean;
Procedure QuestionParms(Var Question,Source: String; Var ID: Integer);
procedure SendModelMessage(Model,Msg: String);
Function MessageLine: String;
Procedure GoSetModelName(Name: String);
Procedure SetServer(Srv: String);
Function GetServer: String;
Procedure DoSendQuestionTo(Model: String; Var Question: String);
Procedure DoSendReplyTo(Model,Reply: String; ID: Integer);
Function ConnectedToServer: Boolean;
Procedure DoSetSynchronization(OnOff: Boolean);

Implementation

Type

  TQuestion = class
    private
      FID: Integer;
      FQuestion: String;
      FTarget: String;
    public
      property Question: String read FQuestion;
      property Destination: String read FTarget;
      property Source: String read FTarget;
    End;

  TReply = class
    private
      FID: Integer;
      FSource: String;
      FAnswer: String;
    public
      property Answer: String read FAnswer;
      property Source: String read FSource;
      property Destination: String read FSource;
    End;

  TTomasStatus = class
    private
      TS_SimulationStatus: SimulationStatus;
      TS_NextStep: Boolean;
      TS_CurrentElement: TomasProcessElement;
      TS_ServerAddress: String;
      TS_Now: Double;
      TS_Tracing: String;
      TS_Message: String;
      TS_CurrentChange: Boolean;
      TS_Question: TQuestion;
      TS_SimulationSpeed: Integer;
    published
```

```
      Constructor Create;
    End;

  LocalData = ^LocalDataRecord; {structure created by: hold, }
  LocalDataRecord =             {Suspend or standby        }
    Record
      LD_Length: LongInt;
      LD_Data: Array[1..64000] Of Byte;
    End;


  Sequel = ^SequelRecord;
  SequelRecord =
    Record
      SR_Next: Sequel;
      SR_Pred: Sequel;
      SR_Element: TomasProcessElement;
      SR_Time: Double;
      SR_StandBy: Boolean;
      SR_LocalData: Localdata;
    End;

  TModelSocket = class(TWSocket)
    published
      Constructor Create(AOwner: TComponent); override;
      procedure RegisterConnection(Sender: TObject; Error: Word);
      procedure CheckServerAnswer(Sender: TObject; Error: Word);
    End;

Const
  ServerMessages : Array[0..10] of string =
      ('','$$!','$$?','$$hello:','$$continue:','$$stop:',
'$$simtime:','$$time:','$$clients:','$$from:','$$unknown:');

Var
  TomasStatus: TTomasStatus;
  SequentialMechanism: Sequel;
  StandByMechanism: Sequel;
  FutureEvent: Sequel;
  SB_Successor: Sequel;
  CurrentSequel: Sequel;
  CurElement: TomasProcessElement;
  TheElement: TomasProcessElement;
  DestroyRoom: TList;
  QuestionList: TList;
  QuestionID: Integer;
  ReplyList: TList;
  NoReplyReceived: Boolean;
  ReplyRecognized: Boolean;
  StackPosition: Pointer;
  JumpAddress: Pointer;
  NrOfBytes: LongInt;
  InitialStack: Pointer;
  ReturnAddress: Pointer;
  LocBytes: Integer;
  LocData: LocalData;
  StartEAX: Integer;
  StartEBX: Integer;
  StartECX: Integer;
  StartESI: Integer;
  StartEDI: Integer;
```

```
  StartEBP: Integer;
  SimulationStopTime: Double;
  GoOnModel: Boolean;
  SimulationBreakTime: Double;
  SimulationKeyTime: Double;
  SimulationReturn: Pointer;
  StatusChange: ChangeProcedure;
  InStepMode: Boolean;
  ModelSocket: TWSocket;
  ModelNumber: String;
  ModelName: String;
  WaitMsg: String;
  MessageReceived: Boolean;
  Buffer: Array[0..2047] Of Char;
  Logging: Boolean;
  LoggingStarted: Boolean;
  LogFile: TextFile;
  LogCount: Integer;
  EventCounter: Integer;
  ProcessingInterval: Integer;
  Synchronization: Boolean;

Procedure SelectNextCurrent; forward;
Function CompInSequence(Comp: TomasProcessElement): Boolean; forward;


Procedure LogMessage(Msg,MsgType: String);
Var
  Log2File: TextFile;
  S: String;
Begin
  If Logging Then
  Begin
    Inc(LogCount);
    If LogCount >= 1000 Then
    Begin
      Reset(LogFile);
      While LogCount > 500 Do
      Begin
        ReadLn(LogFile,S);
        Dec(LogCount);
      End;
      AssignFile(Log2File,ModelName + '.lg2');
      ReWrite(Log2File);
      While Not(EOF(LogFile)) Do
      Begin
        ReadLn(LogFile,S);
        WriteLn(Log2File,S);
      End;
      CloseFile(LogFile);
      CloseFile(Log2File);
      Erase(LogFile);
      Rename(Log2File,ModelName + '.log');
    End;
    Append(LogFile);
    WriteLn(LogFile,FormatFloat('0.00',GetNow) + MsgType + Msg);
    CloseFile(LogFile);
  End;
End;

Procedure DoSetSynchronization(OnOff: Boolean);
```

```
Begin
  Synchronization:=OnOff;
End;

Constructor TTomasStatus.Create;
Begin
  inherited;
  TS_SimulationStatus:=S_INIT;
  TS_CurrentElement:=Nil;
  TS_ServerAddress:='';
  TS_Now:=0;
  TS_Tracing:='';
  TS_Message:='';
  TS_CurrentChange:=FALSE;
  TS_Question:=Nil;
  TS_SimulationSpeed:=100;
End;

Procedure GoSetSimulationSpeed(Speed: Integer);
Begin
  If Speed < 0 Then Speed:=0;
  If Speed > 100 Then Speed:=100;
  TomasStatus.TS_SimulationSpeed:=Speed;
End;


Procedure SetKeyPressed(C: Char);
Begin
  If C = ' ' Then
  Begin
    SimulationKeyTime:=TomasStatus.TS_Now;
  End;
End;

Function TracingMessage: Boolean;
Begin
  Result:=(TomasStatus.TS_Tracing<>'');
End;

Function TraceLine: String;
Begin
  TraceLine:=TomasStatus.TS_Tracing;
  TomasStatus.TS_Tracing:='';
End;

Function CurrentChange: Boolean;
Begin
  Result:=TomasStatus.TS_CurrentChange;
End;

Function ServerMessage: Boolean;
Begin
  Result:=(TomasStatus.TS_Message <> '');
End;

Function ServerQuestion: Boolean;
Begin
  Result:=(TomasStatus.TS_Question <> Nil);
End;

Procedure QuestionParms(Var Question,Source: String; Var ID: Integer);
```

```
Begin
  Question:=TomasStatus.TS_Question.Question;
  Source:=TomasStatus.TS_Question.Source;
  ID:=TomasStatus.TS_Question.FID;
  TomasStatus.TS_Question.Free;
  TomasStatus.TS_Question:=Nil;
End;


Procedure TomasSendStr(S: String);
Var
  k: Integer;
  i: Integer;
Begin
  LogMessage(S,' sent: ');
  i:=0;
  k:=ModelSocket.SendStr(S);
  While (k = -1) And (i<10) Do
  Begin
    k:=ModelSocket.SendStr(S);
    inc(i);
  End;
  If (k = -1) And (i=10) Then
  Begin
    If Logging Then
    Begin
      LogMessage('SendStr from model '+ ModelName,' Error: ');
    End
    Else
    Begin
      ShowMessageDlg('Error sending message: '+ S);
    End;
  End;
End;

Procedure SendModelMessage(Model,Msg: String);
Begin
  If Model = '$$server' Then
  Begin
    TomasSendStr('$$' + Msg + #13#10);
  End
  Else
  Begin
    TomasSendStr('$$toclient: ' + Model + '!!' + Msg + #13#10);
  End;
End;

Function MessageLine: String;
Begin
  MessageLine:=TomasStatus.TS_Message;
  TomasStatus.TS_Message:='';
End;

Procedure SetStatusChange(ChangeProc: ChangeProcedure);
Begin
  StatusChange:=ChangeProc;
End;

Procedure GoSetModelName(Name: String);
Begin
  ModelName:=Name;
```

```
End;

Procedure SetServer(Srv: String);
Begin
  TomasStatus.TS_ServerAddress:=Trim(Srv);
End;

Function GetServer: String;
Begin
  Result:=TomasStatus.TS_ServerAddress;
End;

Procedure SetBreakTime(T: Double);
Begin
  SimulationBreakTime:=T;
End;

Function GetBreakTime: Double;
Begin
  Result:=SimulationBreakTime;
End;

Procedure SetStopTime(T: Double);
Begin
  SimulationStopTime:=T;
End;

Function GetStopTime: Double;
Begin
  Result:=SimulationStopTime;
End;

Function GetSimulationStatus: SimulationStatus;
Begin
  Result:=TomasStatus.TS_SimulationStatus;
End;

Procedure StartStepMode;
Begin
  TomasStatus.TS_SimulationStatus:=S_STEPRUNNING;
  TomasStatus.TS_NextStep:=FALSE;
  InStepMode:=TRUE;
End;

Procedure StopStepMode;
Begin
  InStepMode:=FALSE;
  If TomasStatus.TS_SimulationStatus > S_FINISHED Then
  Begin
    TomasStatus.TS_SimulationStatus:=S_RUNNING;
    TomasStatus.TS_NextStep:=TRUE;
  End;
End;

Procedure PerformNextEvent;
Begin
  TomasStatus.TS_NextStep:=TRUE;
End;

Function GetNow: Double;
Begin
```

```
  Result:=TomasStatus.TS_Now;
End;


Function CurrentTimeString: String;
Var
  S: String;
Begin
  Str(TomasStatus.TS_Now:10:2,S);
  Result:=S;
End;


Constructor TomasProcessElement.Create(S: String; CTime: Double);
Begin
  Inherited Create;
  FName:=S;
  FStatus:=TC_DATA;
  FArrivalTime:=CTime;
  FEventTime:=Never;
  FLocData:=Nil;
  Claims:=TList.Create;
End;

Destructor TomasProcessElement.Destroy;
Begin
  Claims.Clear;
  Claims.Free;
  inherited Destroy;
End;

Procedure TomasProcessElement.Process;
Begin
  ShowMessage(CurrentTimeString + ' Warning: ' + Name
   + ' has no process defined');
  SelectNextCurrent;
End;

Function TomasProcessElement.Scheduled: Boolean;
Begin
  Result:=CompInSequence(Self);
End;

Procedure DoContinue;
Begin
  CurrentSequel^.SR_Element.FStatus:=TC_Current;
  CurElement:=CurrentSequel^.SR_Element;
  TomasStatus.TS_CurrentElement:=CurElement;
  If NOT(InStandBy) Then
  Begin
    If TomasStatus.TS_Now <> CurrentSequel^.SR_Time Then
    Begin
      TomasStatus.TS_Now:=CurrentSequel^.SR_Time;
      FutureEvent:=CurrentSequel^.SR_Next;
    End;
  End;
  If FutureEvent = CurrentSequel Then
    FutureEvent:=CurrentSequel^.SR_Next;
  CurElement.FEventTime:=Never;
  ReturnAddress:=CurElement.FReturnAddress;
  TomasStatus.TS_CurrentChange:=TRUE;
```

```
  StatusChange;
  TomasStatus.TS_CurrentChange:=FALSE;
  CurrentSequel^.SR_Pred^.SR_Next:=CurrentSequel^.SR_Next;
  CurrentSequel^.SR_Next^.SR_Pred:=CurrentSequel^.SR_Pred;
  If CurrentSequel^.SR_LocalData<>Nil Then
  Begin
    LocBytes:=CurrentSequel^.SR_LocalData^.LD_length;
    LocData:=Addr(CurrentSequel^.SR_LocalData^.LD_Data[1]);
    asm
      mov  esi,LocData
      mov  ecx,LocBytes
      mov  edi,InitialStack
      sub  edi,LocBytes
      mov  esp,edi
      rep  movsb
    end;
    FreeMem(CurrentSequel^.SR_LocalData,
      CurrentSequel^.SR_LocalData^.LD_Length+4);
    CurrentSequel^.SR_Localdata:=Nil;
    LocData:=Nil;
  End
  Else
  Begin
    asm
      mov esp,InitialStack
      mov ebp,StartEBP
      push edi
      push esi
      push ebx
      push ebp
    End;
  End;
  Dispose(CurrentSequel);
  asm
    pop  ebp
    pop  ebx
    pop  esi
    pop  edi
    mov eax,CurElement
    mov edx,ReturnAddress
    jmp edx
  End;
End;

Procedure AskToContinue;
Begin
  TomasSendStr('$$continue:' + FormatFloat('0.00',CurrentSequel^.SR_Time)
   + #13#10);
  GoOnModel:=FALSE;
  While NOT(GoOnModel) Do
  Begin
    Application.ProcessMessages;
  End;
  DoContinue;
End;

Procedure NotifyServer;
Begin
  If (TomasStatus.TS_ServerAddress <> '')
   And (TomasStatus.TS_CurrentElement = Nil)
   And (TomasStatus.TS_SimulationStatus > S_Finished) Then
```

```
  Begin
    CurrentSequel:=SequentialMechanism^.SR_Next;
    If (CurrentSequel^.SR_Time >= SimulationStopTime) Then
    Begin
      TomasSendStr('$$newevent:' + FormatFloat('0.00',SimulationStopTime)
       + #13#10);
    End
    Else
    Begin
      TomasSendStr('$$newevent:'
       + FormatFloat('0.00',CurrentSequel^.SR_Time) + #13#10);
    End;
  End;
End;

Procedure WaitForFinish;
Begin
  If ModelSocket.State <> wsClosed Then
  Begin
    TomasSendStr('$$continue:' + FormatFloat('0.00',SimulationStopTime)
     + #13#10);
    GoOnModel:=FALSE;
    While NOT(GoOnModel) Do
    Begin
      Application.ProcessMessages;
    End;
    SelectNextCurrent;
  End;
End;

Procedure SelectNextCurrent;
Var
   InStandBy: Boolean;
   CurrentSequel: Sequel;
Begin
  TomasStatus.TS_CurrentElement:=Nil;
  While DestroyRoom.Count > 0 Do
  Begin
    TheElement:=DestroyRoom.Items[0];
    DestroyRoom.Remove(TheElement);
    TheElement.Destroy;
  End;
  If StandByMechanism^.SR_Next = StandByMechanism Then
  Begin
    InStandBy:=FALSE;
    CurrentSequel:=SequentialMechanism^.SR_Next;
    SB_Successor:=Nil;
  End
  Else
  Begin
  If NOT(InStandBy) Then
    Begin
      CurrentSequel:=StandByMechanism^.SR_Next;
      InStandBy:=TRUE;
      SB_Successor:=CurrentSequel^.SR_Next;
    End
    Else
    Begin
      If SB_Successor = StandByMechanism Then
      Begin
        InStandBy:=FALSE;
```

```
      SB_Successor:=Nil;
      CurrentSequel:=SequentialMechanism^.SR_Next;
    End
    Else
    Begin
      CurrentSequel:=SB_Successor;
      SB_Successor:=CurrentSequel^.SR_Next;
    End;
  End;
  If InStandBy Then
  Begin
    CurrentSequel^.SR_Time:=TomasStatus.TS_Now;
  End;
End;
If ((SimulationBreakTime<>-1) And
 (CurrentSequel^.SR_Time >= SimulationBreakTime)) Or
 (SimulationKeyTime<>-1) Then
Begin
  If SimulationKeyTime <> -1 Then
  Begin
    TomasStatus.TS_Now:=SimulationKeyTime;
    SimulationKeyTime:=-1;
  End
  Else
  Begin
    TomasStatus.TS_Now:=SimulationBreakTime;
    SimulationBreakTime:=-1;
  End;
  TomasStatus.TS_SimulationStatus:=S_INTERRUPTED;
  StatusChange;
End;
If TomasStatus.TS_SimulationStatus = S_STEPRUNNING Then
Begin
  TomasStatus.TS_NextStep:=FALSE;
End;
Dec(EventCounter);
If EventCounter = 0 Then
Begin
  Application.ProcessMessages;
  EventCounter:=ProcessingInterval;
End;
While ((TomasStatus.TS_SimulationStatus < S_RUNNING) And
 (TomasStatus.TS_SimulationStatus <> S_FINISHED)) OR
  NOT(TomasStatus.TS_NextStep) Do
Begin
  Application.ProcessMessages;
End;
If (CurrentSequel^.SR_Time >= SimulationStopTime) Then
Begin
  If (TomasStatus.TS_ServerAddress<>'') And Synchronization
   And (TomasStatus.TS_Now < SimulationStopTime) Then
  Begin
    asm
      pop  edx     {destroy ebx}
      pop  edx     {destroy returnaddress}
    End;
    WaitForFinish;
  End;
  If TomasStatus.TS_SimulationStatus >= S_RUNNING Then
  Begin
    If SimulationStopTime >= 0 Then
```

```
      Begin
        TomasStatus.TS_Now:=SimulationStopTime;
      End;
      TomasStatus.TS_SimulationStatus:=S_FINISHED;
      StatusChange;
      asm
        pop edx   {pop ebx}
        pop edx   {pop returnaddress}
        mov esp,InitialStack
        mov ebp,StartEBP
        mov ebx,StartEBX
        mov esi,StartESI
        mov edi,StartEDI
        mov eax,STARTEAX
        mov ecx,STARTECX
        mov edx,SimulationReturn
        jmp edx
      End;
    End
    Else
    Begin
      Exit;
    End;
  End;
  If TomasStatus.TS_SimulationSpeed < 100 Then
    Sleep(100 - TomasStatus.TS_SimulationSpeed);
  If (TomasStatus.TS_ServerAddress <> '') And Synchronization Then
  Begin
    AskToContinue;
  End
  Else
  Begin
    DoContinue;
  End;
End;

Procedure EnterSequence(T: Double; Comp: TomasProcessElement);
Var
  NewSequence: Sequel;
  NextSequence: Sequel;
Begin
  New(NewSequence);
  NewSequence^.SR_Element:=Comp;
  NewSequence^.SR_Time:=T;
  NewSequence^.SR_StandBy:=FALSE;
  NewSequence^.SR_LocalData:=LocData;
  LocData:=Nil;
  Comp.FStatus:=TC_TIMESCHEDULED;
  Comp.FEventTime:=T;
  If FutureEvent^.SR_Time > T Then
  Begin
    NextSequence:=FutureEvent;
    If T > GetNow Then
      FutureEvent:=NewSequence;
  End
  Else
  Begin
    NextSequence:=FutureEvent^.SR_Next;
    While (NextSequence^.SR_Time<=T) Do
    Begin
      NextSequence:=NextSequence^.SR_Next;
```

```
    End;
  End;
  NewSequence^.SR_Next:=NextSequence;
  NewSequence^.SR_Pred:=NextSequence^.SR_Pred;
  NewSequence^.SR_Pred^.SR_Next:=NewSequence;
  NextSequence^.SR_Pred:=NewSequence;
  NotifyServer;
End;

Procedure EnterStandBySequence(Comp: TomasProcessElement;
 NewStandBy: Boolean);
Var
  NewSequence: Sequel;
Begin
  New(NewSequence);
  NewSequence^.SR_Element:=Comp;
  NewSequence^.SR_LocalData:=LocData;
  NewSequence^.SR_StandBy:=TRUE;
  NewSequence^.SR_Time:=GetNow;
  LocData:=Nil;
  Comp.FStatus:=TC_STATESCHEDULED;
  If NewStandBy Then
  Begin
    NewSequence^.SR_Next:=StandByMechanism;
    NewSequence^.SR_Pred:=StandByMechanism^.SR_Pred;
    NewSequence^.SR_Pred^.SR_Next:=NewSequence;
    StandByMechanism^.SR_Pred:=NewSequence;
  End
  Else
  Begin
    NewSequence^.SR_Next:=SB_Successor;
    NewSequence^.SR_Pred:=SB_Successor^.SR_Pred;
    SB_Successor^.SR_Pred:=NewSequence;
    NewSequence^.SR_Pred^.SR_Next:=NewSequence;
  End;
  Comp.FEventTime:=GetNow;
  NotifyServer;
End; {EnterStandBySequence}


Procedure TomasProcessElement.ScheduleProcess(T: Double);
Begin
  If FLocData<>Nil Then
  Begin
    LocData:=FLocData;
    FLocData:=Nil;
  End
  Else
  Begin
    LocData:=Nil;
    FReturnAddress:=MethodAddress('Process');
  End;
  If T >= GetNow Then
  Begin
    EnterSequence(T,Self);
  End
  Else
  Begin
    EnterStandBySequence(Self,TRUE);
  End;
End;
```

```
Procedure LeaveSequence(Comp: TomasProcessElement);
  Var
    NextSequence: Sequel;
Begin
  NextSequence:=StandbyMechanism^.SR_Next;
  While (NextSequence<>StandByMechanism)
   And (NextSequence^.SR_Element <> Comp) Do
  Begin
    NextSequence:=NextSequence^.SR_Next;
  End;
  If NextSequence = StandByMechanism Then
  Begin
    NextSequence:=SequentialMechanism^.SR_Next;
    While (NextSequence^.SR_Element <> Comp) Do
    Begin
      NextSequence:=NextSequence^.SR_Next;
    End;
  End;
  If NextSequence = FutureEvent Then
    FutureEvent:=NextSequence^.SR_Next;
  NextSequence^.SR_Pred^.SR_Next:=NextSequence^.SR_Next;
  NextSequence^.SR_Next^.SR_Pred:=NextSequence^.SR_Pred;
  Comp.FStatus:=TC_DATA;
  Comp.FEventTime:=Never;
  If NextSequence^.SR_LocalData <> Nil Then
  Begin
    FreeMem(NextSequence^.SR_LocalData,
     NextSequence^.SR_LocalData^.LD_Length+4);
  End;
  NextSequence^.SR_LocalData:=Nil;
  Dispose(NextSequence);
  NotifyServer;
End;

Procedure TomasProcessElement.RemoveProcess(Stop: Boolean);
Begin
  If (SB_Successor <> Nil) And (SB_Successor^.SR_Element = Self) Then
  Begin
    SB_Successor:=SB_Successor^.SR_Next;
  End;
  LeaveSequence(Self);
  If Stop Then
  Begin
    FStatus:=TC_DATA;
  End
  Else
  Begin
    FStatus:=TC_PASSIVE;
  End;
End;   {RemoveProcess}

Procedure TomasProcessElement.CancelResume;
Var
  ELocData: LocalData;
Begin
  If Assigned(FLocData) Then
  Begin
    ELocdata:=FLocData;
    FreeMem(ELocData,ELocData^.LD_Length+4);
```

```
    FLocData:=Nil;
  End;
  FStatus:=TC_DATA;
End;

Procedure TomasProcessElement.InterruptProcess;
Var
  NextSequence: Sequel;
Begin
  If (SB_Successor <> Nil) And (SB_Successor^.SR_Element = Self) Then
  Begin
    SB_Successor:=SB_Successor^.SR_Next;
  End;
  NextSequence:=StandbyMechanism^.SR_Next;
  While (NextSequence<>StandByMechanism)
   And (NextSequence^.SR_Element <> Self) Do
  Begin
    NextSequence:=NextSequence^.SR_Next;
  End;
  If NextSequence = StandByMechanism Then
  Begin
    NextSequence:=SequentialMechanism^.SR_Next;
    While (NextSequence^.SR_Element <> Self) Do
    Begin
      NextSequence:=NextSequence^.SR_Next;
    End;
  End;
  NextSequence^.SR_Pred^.SR_Next:=NextSequence^.SR_Next;
  NextSequence^.SR_Next^.SR_Pred:=NextSequence^.SR_Pred;
  FStatus:=TC_INTERRUPTED;
  FEventTime:=Never;
  If NextSequence^.SR_LocalData <> Nil Then
  Begin
    FLocData:=NextSequence^.SR_LocalData;
    NextSequence^.SR_LocalData:=Nil;
  End;
  Dispose(NextSequence);
  NotifyServer;
End;

Procedure TomasProcessElement.CancelProcess;
  Var
    NextSequence: Sequel;
Begin
  If (SB_Successor <> Nil) And (SB_Successor^.SR_Element = Self) Then
  Begin
    SB_Successor:=SB_Successor^.SR_Next;
  End;
  NextSequence:=StandbyMechanism^.SR_Next;
  While (NextSequence<>StandByMechanism)
   And (NextSequence^.SR_Element <> Self) Do
  Begin
    NextSequence:=NextSequence^.SR_Next;
  End;
  If NextSequence = StandByMechanism Then
  Begin
    NextSequence:=SequentialMechanism^.SR_Next;
    While (NextSequence^.SR_Element <> Self) Do
    Begin
      NextSequence:=NextSequence^.SR_Next;
    End;
```

```
  End;
  If FutureEvent = NextSequence Then
    FutureEvent:=NextSequence^.SR_Next;
  NextSequence^.SR_Pred^.SR_Next:=NextSequence^.SR_Next;
  NextSequence^.SR_Next^.SR_Pred:=NextSequence^.SR_Pred;
  FStatus:=TC_PASSIVE;
  FEventTime:=Never;
  If NextSequence^.SR_LocalData <> Nil Then
  Begin
    FLocData:=NextSequence^.SR_LocalData;
    NextSequence^.SR_LocalData:=Nil;
  End;
  Dispose(NextSequence);
  NotifyServer;
End;

Function CompInSequence(Comp: TomasProcessElement): Boolean;
Var
  NextSequence: Sequel;
Begin
  NextSequence:=StandbyMechanism^.SR_Next;
  While (NextSequence<>StandByMechanism)
   And (NextSequence^.SR_Element <> Comp) Do
  Begin
    NextSequence:=NextSequence^.SR_Next;
  End;
  If NextSequence = StandByMechanism Then
  Begin
    NextSequence:=SequentialMechanism^.SR_Next;
    While (NextSequence<>SequentialMechanism)
     And (NextSequence^.SR_Element <> Comp) Do
    Begin
      NextSequence:=NextSequence^.SR_Next;
    End;
  End;
  Result:=(NextSequence <> SequentialMechanism)
End;


Procedure HoldProcess(T: Double);
Label
  RetHold;
Begin
  asm
    push edi
    push esi
    push ebx
    push ebp
    mov ecx,InitialStack
    mov edx,esp
    sub ecx,esp
    mov NrOfBytes,ecx
  End;
  GetMem(LocData,NrOfBytes+4);
  LocData^.LD_Length:=NrOfBytes;
  StackPosition:=Addr(LocData^.LD_Data[1]);
  asm
    mov  eax,esi
    mov  edx,edi
    mov  edi,StackPosition
    mov  esi,esp
```

```
    mov  ecx,NrOfBytes
    rep  movsb
    mov  ecx,OFFSET RetHold
    mov  JumpAddress,ecx
    mov  esi,eax
    mov  edi,edx
  End;
  TomasStatus.TS_CurrentElement.FReturnAddress:=JumpAddress;
  EnterSequence(TomasStatus.TS_Now + T,TomasStatus.TS_CurrentElement);
  asm
    mov esp,InitialStack
  End;
  SelectNextCurrent;
  asm
    RetHold:
  End;
End;

Procedure StandByProcess;
Label
  RetStandBy;
Begin
  asm
    push edi
    push esi
    push ebx
    push ebp
    mov ecx,InitialStack
    mov edx,esp
    sub ecx,esp
    mov NrOfBytes,ecx
  End;
  GetMem(LocData,NrOfBytes+4);
  LocData^.LD_Length:=NrOfBytes;
  StackPosition:=Addr(LocData^.LD_Data[1]);
  asm
    mov  eax,esi
    mov  edx,edi
    mov edi,StackPosition
    mov esi,esp
    mov ecx,NrOfBytes
    rep movsb
    mov  ecx,OFFSET RetStandBy
    mov  JumpAddress,ecx
    mov  esi,eax
    mov  edi,edx
  End;
  If TomasStatus.TS_CurrentElement.FReturnAddress=JumpAddress Then
  Begin
    EnterStandBySequence(TomasStatus.TS_CurrentElement,FALSE);
  End
  Else
  Begin
    TomasStatus.TS_CurrentElement.FReturnAddress:=JumpAddress;
    EnterStandBySequence(TomasStatus.TS_CurrentElement,TRUE);
  End;
  asm
    mov esp,InitialStack
  End;
  SelectNextCurrent;
  asm
```

```
    RetStandBy:
  End;
End;

Procedure SuspendProcess;
Label
  RetSuspend;
Begin
  asm
    push edi
    push esi
    push ebx
    push ebp
    mov ecx,InitialStack
    mov edx,esp
    sub ecx,esp
    mov NrOfBytes,ecx
  End;
  GetMem(LocData,NrOfBytes+4);
  LocData^.LD_Length:=NrOfBytes;
  StackPosition:=Addr(LocData^.LD_Data[1]);
  TomasStatus.TS_CurrentElement.FLocData:=LocData;
  asm
    mov  eax,esi
    mov  edx,edi
    mov edi,StackPosition
    mov esi,esp
    mov ecx,NrOfBytes
    rep movsb
    mov  ecx,OFFSET RetSuspend
    mov  JumpAddress,ecx
    mov  esi,eax
    mov  edi,edx
  End;
  TomasStatus.TS_CurrentElement.FReturnAddress:=JumpAddress;
  TomasStatus.TS_CurrentElement.FEventTime:=Never;
  LocData:=Nil;
  asm
    mov esp,InitialStack
  End;
  SelectNextCurrent;
  asm
    RetSuspend:
  End;
End;

Procedure FinishProcess;
Begin
  TomasStatus.TS_CurrentElement.FStatus:=TC_DATA;
  SelectNextCurrent;
End;

Procedure FinishProcessAndDestroy;
Begin
  TomasStatus.TS_CurrentElement.FStatus:=TC_DATA;
  DestroyRoom.Add(TomasStatus.TS_CurrentElement);
  SelectNextCurrent;
End;

Procedure StartProcessSimulation;
Label
```

```
  LSimulationReturn;
Begin
  asm
    mov InitialStack,esp
    mov StartEBP,ebp
    mov StartESI,esi
    mov StartEDI,edi
    mov StartEAX,eax
    mov StartEBX,ebx
    mov StartECX,ecx
    mov edx,offset LSimulationReturn
    mov SimulationReturn,edx
  End;
  TomasStatus.TS_Now:=0;
  If NOT(InStepMode) Then
  Begin
    TomasStatus.TS_SimulationStatus:=S_RUNNING;
  End;
  TomasStatus.TS_NextStep:=TRUE;
  CheckServer;
  SelectNextCurrent;
  asm
    LSimulationReturn:
  End;
End;

Procedure ResumeProcessSimulation;
Begin
  If InStepMode Then
  Begin
    TomasStatus.TS_SimulationStatus:=S_STEPRUNNING;
    TomasStatus.TS_NextStep:=TRUE;
  End
  Else
  Begin
    TomasStatus.TS_SimulationStatus:=S_RUNNING;
  End;
End;

Procedure InterruptProcessSimulation;
Begin
  TomasStatus.TS_SimulationStatus:=S_INTERRUPTED;
End;

Function CurrentProcessElement: TomasProcessElement;
Begin
  Result:=TomasStatus.TS_CurrentElement;
End;

Procedure CheckServer;
Begin
  If (TomasStatus.TS_ServerAddress<>'')
   And (ModelSocket.State <> wsConnected) Then
  Begin
    ModelSocket.Addr:=TomasStatus.TS_ServerAddress;
    ModelSocket.Proto:='tcp';
    ModelSocket.Port:='52623';
    TomasStatus.TS_Tracing:='Connection called';
    StatusChange;
    ModelSocket.Connect;
    GoOnModel:=FALSE;
```

```
    While (ModelSocket.State <> wsConnected) Or NOT(GoOnModel) Do
    Begin
      Application.ProcessMessages;
    End;
  End;
End;

Function ConnectedToServer: Boolean;
Begin
  Result:=(ModelSocket.State = wsConnected);
End;

Constructor TModelSocket.Create(AOwner: TComponent);
Begin
  inherited Create(AOwner);
  OnSessionConnected:=RegisterConnection;
  OnDataAvailable:=CheckServerAnswer;
End;

Procedure TModelSocket.RegisterConnection(Sender: TObject; Error: Word);
Begin
  If Error <> 0 Then
  Begin
    TomasStatus.TS_Tracing:='Connection failed';
  End
  Else
  Begin
    TomasStatus.TS_Tracing:='Connected to Server';
  End;
  StatusChange;
End;

Procedure TModelSocket.CheckServerAnswer(Sender: TObject; Error: Word);
Var
  S: String;
  STot: String;
  Sm: String;
  Len: Integer;
  P: Integer;
  MsgNr: Integer;
  HL: Integer;
  TSequel: Sequel;
  NewReply: TReply;
  NewQuestion: TQuestion;

  Function GetMsgNr(Msg: String): Integer;
  Var
    i: Integer;
  Begin
    i:=10;
    While (I>0) And (Pos(ServerMessages[i],Msg)=0) Do
    Begin
      Dec(i);
    End;
    HL:=Length(ServerMessages[i]);
    Result:=i;
  End;

Begin
  Len := Receive(@Buffer[0], sizeof(Buffer) - 1);
  While (Len > 0) Do
```

```
Begin
  While (Len > 0) AND (Buffer[Len-1] in [#13, #10]) Do
  Begin
    Dec (Len);
  End;
  Buffer[Len] := #0;
  STot := Trim(StrPas(Buffer));
  While STot <> '' Do
  Begin
    P:=Pos(#13#10,STot);
    If P = 0 Then
    Begin
      S:=STot;
      STot:='';
    End
    Else
    Begin
      S:=Copy(STot,1,P-1);
      STot:=Copy(STot,P + 2, Length(STot) - P - 1);
    End;
    MsgNr:=GetMsgNr(S);
    Case MsgNr Of
      0:              {unknown answer; send it to receiveproc}
      Begin
        TomasStatus.TS_Message:=S;
        LogMessage(S,' received: ');
        StatusChange;
      End;
      1:              {! = reply}
      Begin
        LogMessage(S,' received: ');
        P:=Pos('!!',S);
        NewReply:=TReply.Create;
        NewReply.FID:=StrToInt(Copy(S,HL+1,P-HL-1));
        S:=Copy(S,P+2,Length(S) - P - 1);
        P:=Pos('!!',S);
        NewReply.FSource:=Copy(S,1,P-1);
        NewReply.FAnswer:=Copy(S,P+2,Length(S) - P - 1);
        ReplyList.Add(NewReply);
        NoReplyReceived:=FALSE;
      End;
      2:              {? = question}
      Begin
        LogMessage(S,' received: ');
        NewQuestion:=TQuestion.Create;
        P:=Pos('!!',S);
        NewQuestion.FID:=StrToInt(Copy(S,HL+1,P-HL-1));
        S:=Copy(S,P+2,Length(S) - P - 1);
        P:=Pos('!!',S);
        NewQuestion.FTarget:=Copy(S,1,P-1);
        NewQuestion.FQuestion:=Copy(S,P+2,Length(S) - P - 1);
        TomasStatus.TS_Question:=NewQuestion;
        StatusChange;
      End;
      3:              {hello}
      Begin
        P:=Pos('!!',S);
        ModelNumber:=Copy(S,HL+1,P - HL - 1);
        LogMessage(S,' received: ');
        If Synchronization Then
        Begin
```

```
    TomasStatus.TS_Now:=StrToFloat(Trim(Copy(S,P + 2,
     Length(S)- P -   1)));
    TSequel:=SequentialMechanism^.SR_Next;
    While TSequel <> SequentialMechanism Do
    Begin
      If TSequel^.SR_Time <> NEVER Then
      Begin
        TSequel^.SR_Time:=TSequel^.SR_Time + TomasStatus.TS_Now;
      End;
      TSequel:=TSequel^.SR_Next;
    End;
  End;
  TomasSendStr('$$name:' + ModelName + #13#10);
  GoOnModel:=TRUE;
End;
4:              {continue}
Begin
  If Synchronization Then
    TomasStatus.TS_Now:=StrToFloat(Trim(Copy(S,HL+1,
     Length(S)-HL)));
  LogMessage(S,' received: ');
  GoOnModel:=TRUE;
End;
5:              {stop}
Begin
  SimulationStopTime:=StrToFloat(Trim(Copy(S,HL+1,Length(S)-HL)));
  LogMessage(S,' received: ');
  GoOnModel:=TRUE;
  SelectNextCurrent;
End;
6:              {Simtime}
Begin
  If Synchronization Then
    TomasStatus.TS_Now:=StrToFloat(Trim(Copy(S,HL+1,
     Length(S)-HL)));
  LogMessage(S,' received: ');
  TomasStatus.TS_Message:=S;
  StatusChange;
End;
7:              {time}
Begin
  TomasStatus.TS_Message:=S;
  LogMessage(S,' received: ');
  StatusChange;
End;
8:              {clients}
Begin
  TomasStatus.TS_Message:=S;
  LogMessage(S,' received: ');
  StatusChange;
End;
9:              {from}
Begin
  P:=Pos('!!',S);
  Sm:=Copy(S,HL+1,P - HL -1);
  S:=Copy(S,1,HL) + Copy(S,P+2,Length(S) - P - 1);
  If Synchronization Then
    TomasStatus.TS_Now:=StrToFloat(Trim(Sm));
  TomasStatus.TS_Message:=S;
  LogMessage(S,' received: ');
  StatusChange;
```

```
        End;
        10:             {unknown}
        Begin
          TomasStatus.TS_Message:=S;
          LogMessage(S,' received: ');
          StatusChange;
        End;
      End;
    End;
    Len := Receive (@Buffer[0], sizeof (Buffer) - 1);
  End;
End;

Procedure DisconnectFromServer;
Begin
  If (TomasStatus.TS_ServerAddress<>'')
   And (ModelSocket.State = wsConnected) Then
  Begin
    ModelSocket.Close;
    GoOnModel:=TRUE;
  End;
End;


Procedure DoStartLogging;
Begin
  If LoggingStarted Then
  Begin
    If FileExists(ModelName + '.log') Then
    Begin
      Append(LogFile);
    End
    Else
    Begin
      AssignFile(LogFile,ModelName + '.log');
      ReWrite(LogFile);
      LogCount:=0;
    End;
    WriteLn(LogFile,'***' + FormatFloat('0.00',GetNow)
     + ' resumes logging');
    CloseFile(LogFile);
  End
  Else
  Begin
    AssignFile(LogFile,ModelName + '.log');
    ReWrite(LogFile);
    WriteLn(LogFile,'***' + FormatFloat('0.00',GetNow) + ' start logging');
    CloseFile(LogFile);
    LogCount:=0;
    LoggingStarted:=TRUE;
  End;
  Logging:=TRUE;
  Inc(LogCount);
End;

Procedure DoStopLogging;
Begin
  If Logging Then
  Begin
    Append(LogFile);
    WriteLn(LogFile,'***' + FormatFloat('0.00',GetNow) + ' stop logging');
```

```
    CloseFile(LogFile);
    Inc(LogCount);
  End;
End;

Function GetLogging: Boolean;
Begin
  Result:=Logging;
End;

Procedure SetProcessingInterval(PInt: Integer);
Begin
  ProcessingInterval:=PInt;
  EventCounter:=PInt;
End;

Procedure DoSendQuestionTo(Model: String; Var Question: String);
Var
  NewQuestion: TQuestion;
  TheReply: TReply;
  AnswerToThisQuestion: Boolean;
  I: Integer;
Begin
  Inc(QuestionID);
  If Model = '' Then
  Begin
    Model:='server';
  End;
  NewQuestion:=TQuestion.Create;
  NewQuestion.FID:=QuestionID;
  NewQuestion.FTarget:=Model;
  NewQuestion.FQuestion:=Question;
  QuestionList.Add(NewQuestion);
  TomasSendStr('$$?' + IntToStr(NewQuestion.FID) + '!!'
   + NewQuestion.FTarget + '!!' + NewQuestion.FQuestion + #13#10);
  AnswerToThisQuestion:=FALSE;
  TheReply:=Nil;
  While NOT(AnswerToThisQuestion) Do
  Begin
    While NoReplyReceived Do
    Begin
      Application.ProcessMessages;
    End;
    I:=0;
    While (I<=ReplyList.Count - 1) And Not(AnswerToThisQuestion) Do
    Begin
      TheReply:=ReplyList.Items[i];
      If TheReply.FID=NewQuestion.FID Then
      Begin
        AnswerToThisQuestion:=TRUE;
      End
      Else
      Begin
        Inc(i);
      End;
    End;
  End;
  If AnswerToThisQuestion Then
  Begin
    QuestionList.Remove(NewQuestion);
    NewQuestion.Free;
```

```
    Question:=TheReply.FAnswer;
    ReplyList.Remove(TheReply);
    TheReply.Free;
    NoReplyReceived:=TRUE;
  End;
End;

Procedure DoSendReplyTo(Model,Reply: String; ID: Integer);
Begin
  TomasSendStr('$$!' + IntToStr(ID) + '!!'
   + Model + '!!' + Reply + #13#10);
End;

Procedure ResetTomasProcess;
  Var
    TheSequel: Sequel;
Begin
  TheSequel:=SequentialMechanism^.SR_Next;
  While TheSequel <> SequentialMechanism Do
  Begin
    TheSequel^.SR_Pred^.SR_Next:=TheSequel^.SR_Next;
    TheSequel^.SR_Next^.SR_pred:=TheSequel^.SR_Pred;
    Dispose(TheSequel);
    TheSequel:=SequentialMechanism^.SR_Next;
  End;
  DestroyRoom.Clear;
  SimulationStopTime:=Never;
  SimulationBreakTime:=-1;
  SimulationKeyTime:=-1;
  ModelSocket.Close;
  TomasStatus.TS_SimulationStatus:=S_INIT;
  TomasStatus.TS_CurrentElement:=Nil;
  TomasStatus.TS_Now:=0;
  TomasStatus.TS_Tracing:='';
  TomasStatus.TS_Message:='';
End;  {ResetTomasProcess}


Procedure InitializeTomasProcess;
Begin
  New(SequentialMechanism);
  SequentialMechanism^.SR_Next:=SequentialMechanism;
  SequentialMechanism^.SR_Pred:=SequentialMechanism;
  SequentialMechanism^.SR_Element:=Nil;
  SequentialMechanism^.SR_Time:=NEVER;
  SequentialMechanism^.SR_StandBy:=FALSE;
  SequentialMechanism^.SR_Localdata:=Nil;
  New(StandByMechanism);
  StandByMechanism^.SR_Next:=StandByMechanism;
  StandByMechanism^.SR_Pred:=StandByMechanism;
  StandByMechanism^.SR_Element:=Nil;
  StandByMechanism^.SR_Time:=NEVER;
  StandByMechanism^.SR_StandBy:=TRUE;
  StandByMechanism^.SR_Localdata:=Nil;
  FutureEvent:=SequentialMechanism;
  InStandBy:=FALSE;
  SB_Successor:=Nil;
  DestroyRoom:=TList.Create;
  QuestionList:=TList.Create;
  ReplyList:=TList.Create;
  QuestionID:=0;
```

```
  NoReplyReceived:=TRUE;
  ReplyRecognized:=FALSE;
  TomasStatus:=TTomasStatus.Create;
  SimulationStopTime:=Never;
  SimulationBreakTime:=-1;
  SimulationKeyTime:=-1;
  InStepMode:=FALSE;
  ModelSocket:=TModelSocket.Create(Nil);
  ModelName:='TomasModel';
  WaitMsg:='';
  GoOnModel:=FALSE;
  Logging:=FALSE;
  LoggingStarted:=FALSE;
  ProcessingInterval:=1;
  EventCounter:=1;
  Synchronization:=TRUE;
End;

Procedure FinishTomasProcess;
Begin
  ResetTomasProcess;
  Dispose(SequentialMechanism);
  DestroyRoom.Free;
  TomasStatus.Free;
  ModelSocket.Free;
End;

Initialization
 InitializeTomasProcess;

Finalization
  FinishTomasProcess;

end.
```

Appendix D

TOMAS model for the DSL terminal

```
unit ProdUnit;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Tomas, TomasSeeds;

Const
  NrASC = 25;
  NrAGV = 48;
  NrQC = 8;
  ASCCycleLB = 0; {seconds}   //ASCCycletime uniformly distributed
  ASCCycleUB = 0; {seconds}
  AGVCycleLB = 510; {seconds} //AGVCycleTime uniformly distributed
  AGVCycleUB = 630; {seconds}
  QCCycle = 110;  {seconds}   //QCCycleTime exponentially distributed

type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

  TAGV = class;
  TASC = class;
  TQC = class;

  TAGV = class(TomasElement)
    QC: TQC;
    published
    Procedure Process; override;
    End;

  TLandSide = class(TomasElement)
    ASCSelector: TUniformDistribution;
    published
    Procedure Process; override;
    End;

  TASC = class(TomasElement)
    AGVSWaiting: TomasQueue;
    LSWaiting: Integer;
    published
    Procedure Process; override;
    End;

  TQC = class(TomasElement)
    AGVSWaiting: TomasQueue;
    AGVSAssigned: Integer;
    published
    Procedure Process; override;
    End;

  TMonitor = class(TomasELement)
    published
    Procedure process; override;
    End;
```

```
  TPriorityQueue = class(TomasQueue)
    published
    Procedure AddSortedQC(QC: TQC);
    End;

var
  Form1: TForm1;
  ASCSelection: TUniformDistribution;
  AGVCycleTime: TUniformDistribution;
  ASCCycleTime: TUniformDistribution;
  QCCycleTime: TExponentialDistribution;
  ASCS: Array[1..NRASC] Of TASC;
  QCPriority: TPriorityQueue;
  NrOfAGVCycles: Integer;
  LandSide: TLandSide;
  Monitor: TMonitor;

implementation

{$R *.DFM}

Procedure TPriorityQueue.AddSortedQC(QC: TQC);
Var
  NextQC: TQC;
Begin
  NextQC:=FirstElement;
  While (NextQC<>Nil) And (NextQC.AGVSAssigned <= QC.AGVSAssigned) Do
  Begin
    NextQC:=Successor(NextQC);
  End;
  If NextQC = Nil Then
    AddToTail(QC)
  Else
    AddBefore(QC,NextQC);
End;

Procedure TAGV.Process;
Var
  ASCNr: Integer;
  Cycle: Double;
Begin
  While TRUE Do
  Begin
    QC:=QCPriority.FirstElement;
    Inc(QC.AGVSAssigned);
    QCPriority.Remove(QC);
    QCPriority.AddSortedQC(QC);
    Cycle:=AGVCycleTime.Sample;
    Hold(0.5 * Cycle);
    EnterQueue(QC.AGVSWaiting);
    Suspend;
    Hold(0.5 * Cycle);
    ASCNr:=Trunc(ASCSelection.Sample);
    EnterQueue(ASCS[ASCNr].AGVSWaiting);
    Suspend;
  End;
End;

Procedure TLandSide.Process;
Var
  ASCNr: Integer;
```

```
Begin
  While TRUE Do
  Begin
    Hold(3600/90);
    ASCNr:=Trunc(ASCSelection.Sample);
    Inc(ASCS[ASCNr].LSWaiting);
    If ASCS[ASCNr].Status = Passive Then
      ASCS[ASCNr].Resume(TNow);
  End;
End;

Procedure TASC.Process;
Var
  AGV: TAGV;
  Cycle: Double;
Begin
  While TRUE Do
  Begin
    While (AGVSWaiting.Length + LSWaiting = 0) Do
      StandBy;
    Cycle:=ASCCycleTime.Sample;
    If AGVSWaiting.Length > 0 Then
    Begin
      Hold(0.5 * Cycle);
      AGV:=AGVSWaiting.FirstElement;
      AGVSWaiting.Remove(AGV);
      Hold(15);
      AGV.Resume(TNow);
      Hold(0.5 * Cycle);
    End
    Else
    Begin
      Hold(Cycle);
      Dec(LSWaiting);
    End;
  End;
End;

Procedure TQC.Process;
Var
  AGV: TAGV;
Begin
  While TRUE Do
  Begin
    Hold(QCCycleTime.Sample-15);
    While AGVSWaiting.Length = 0 Do
      StandBy;
    AGV:=AGVSWaiting.FirstElement;
    AGVSWaiting.Remove(AGV);
    Hold(15);
    Dec(AGVSAssigned);
    QCPriority.Remove(Self);
    QCPriority.AddSortedQC(Self);
    AGV.Resume(TNow);
    Inc(NrOfAGVCycles);
  End;
End;

Procedure TMonitor.Process;
Var
  OutFile: TextFile;
```

```
Begin
  AssignFile(OutFile,'Production.txt');
  Rewrite(OutFile);
  WriteLn(OutFile, Date);
  CloseFile(OutFile);
  While TRUE Do
  Begin
    Hold(3600);
    Append(OutFile);
    WriteLn(OutFile,NrOfAGVCycles);
    CloseFile(OutFile);
    NrOfAGVCycles:=0;
  End;
End;

Procedure InitializeModel;
Var
  i: Integer;
  AGV: TAGV;
  ASC: TASC;
  QC: TQC;
Begin
  ASCSelection:=TUniformDistribution.Create(Seed(1),1,NrASC + 1);
  AGVCycleTime:=TUniformDistribution.Create(Seed(2),AGVCycleLB,AGVCycleUB);
  ASCCycleTime:=TUniformDistribution.Create(Seed(3),ASCCycleLB,ASCCycleUB);
  NrOfAGVCycles:=0;
  For i:=1 To NrASC Do
  Begin
    ASC:=TASC.Create('ASC');
    ASC.AGVSWaiting:=TomasQueue.Create('WASC_' + IntToStr(i));
    ASC.LSWaiting:=0;
    ASC.Start(TNow);
    ASCS[I]:=ASC;
  End;
  For i:= 1 To NrAGV Do
  Begin
    AGV:=TAGV.Create('AGV');
    AGV.Start(TNow);
  End;
  LandSide:=TLandSide.Create('LandSide');
  LandSide.ASCSelector:=TUniformDistribution.Create(Seed(4),1,NrASC + 1);
  WaitingQCs:=TomasQueue.Create('WaitingQCs');
  QCCycleTime:=TExponentialDistribution.Create(Seed(5),QCCycle);
  QCPriority:=TPriorityQueue.Create('QCs');
  For i:= 1 To NrQC Do
  Begin
    QC:=TQC.Create('QC');
    QC.AGVSWaiting:=TomasQueue.Create('WQC_' + IntToStr(i));
    QC.AGVSAssigned:=0;
    QCPriority.AddToTail(QC);
    QC.Start(TNow);
  End;
  LandSide.Start(TNow);
  Monitor:=TMonitor.Create('Monitor');
  Monitor.Start(TNow);
End;

Initialization
  InitializeModel;

end.
```

About the author

Hermanus Petrus Maria Veeke was born on June 23th, 1952 in Breda, The Netherlands. After finishing his primary education at the Onze Lieve Vrouwe Lyceum in Breda in 1970, he started his study at the Hogere Informatica Opleiding in Eindhoven and received his engineering degree with a simulation study of the performance of the IBM 360 processor family in 1974. Then he started his working career as a systems analyst at the Computing Centre of the Delft University of Technology.

In 1977 he changed to the Faculty of Mathematics at this university and was assigned the position of scientific programmer. The same year he started his study Mathematics and specialized in Operations Research. He graduated in June 1983 under supervision of prof. dr. ir. P. Wesseling and R.W. Sierenberg on the subject of combined discrete continuous simulation. In the mean time he changed to the Faculty of Design, Engineering and Production and joined the section Industrial Organization of Prof. ir. J. in 't Veld, where he specialized in the application of the systems approach and simulation to analyze and solve organizational problems.

From 1985 until 1988 he joined the consultancy Sierenberg & De Gans in Waddinxveen. He was a co-developer of the simulation language Personal PROSIM and was assigned consultancy projects at Heineken in Zoeterwoude and Melkunie in Woerden. Since 1989 he is a self-employed consultant with two major activities: consultancy and software development. As a consultant he was involved in many innovation projects. Most of these projects are concerned with the development of (more or less) automated container terminals e.g. with ECT in Rotterdam, the European IPSI-project and projects of the TRAIL Research School. As a software developer he developed a system for simulation based scheduling and project planning (BART which stands for Bottleneck Avoiding Resource Tool). In 1994 he started the development of the production supporting software system EOS (Electronic Order System) for Carrier Bedrijfskoeling BV in Culemborg, which is based on the functional systems approach and which is operational and growing until now. Finally he developed the free source simulation environment TOMAS as described in this thesis.

Since 1995 he rejoined the section Industrial Organization (now managed by Prof. ir. H.Bikker) on a part-time basis. Above that, he is involved with postgraduate courses such as the Master of Business Management course of TSM in Twente and the Management of

Transportation and Logistics course of TIAS in Tilburg. Together with dr. ir. J. Ottjes of the Department Marine and Transport Technology, he continued the research in the field of process interaction simulation. This cooperation generated about twenty conference papers since 1999 and finally resulted in a 'best paper' award at the European Concurrent Engineering Conference this year.

He is married to Mies Hermans since 1979 and together they have four children Bart, Jeroen, Max and Maartje.