Verification and simulation of the transport system architecture using formal specification techniques

Bessems, Peter J.J.

Award date:
1988

Verification and Simulation of the Transport System Architecture using Formal Specification Techniques.

by Peter J.J. Bessems

M.Sc. Thesis Report
Supervisors:
Prof.Dr.Ir. C.J. Koomen, Eindhoven University of Technology
Faculty of Electrical Engineering
Ir. J.J.A.J. Beukeboom, Nederlandse Philips Bedrijven BV
CFT/CAM-Centre
Eindhoven, The Netherlands
August 1988

Summary

This report shows some insights and experiences in the usage of Formal Description Techniques (FDT) for the specification of CAM-Architectures. A comparison between some relevant FDTs is given, which establishes LOTOS to be the most useful at the moment. With the aid of the very recent LOTOS software tools a simplified version of the already existing LOTOS specification of the Internal Transport System (ITS) Architecture has been verified. It appears that already in a very early stage of a design fundamental constraints can be checked on their consistency. Meanwhile, a new view on transport within the CAM-Reference Model is shown, and the strong analogy between the transport controller and other controllers on workstation level within the Reference Model. Based on this new view, together with existing Transport Controller System User Requirements, this report presents a set of functional requirements and a new LOTOS specification of a Transport Control System (TCS).

Contents

Preface

This document presents the M.Sc. Thesis as a final result of the graduation project to gain the grade of "Electrotechnisch Ingenieur" at the Eindhoven University of Technology, The Netherlands. The project was done within the CAM-Centre of the Centre for Manufacturing Technologies (CFT) of the Nederlandse Philips Bedrijven BV in Eindhoven.

The project was supervised by Prof.Dr.Ir. C.J. Koomen, professor at the Faculty of Electrical Engineering of the Eindhoven University of Technology, and Ir. J.J.A.J. Beukeboom, member of the CAM Concepts and Prototyping (CCP) group of the Department Technical Automation Means (TAM) within the CFT/CAM-Centre.

The aim of the graduation project was to verify and simulate the formal specifications of the Philips Internal Transport System (ITS). The motivation for the project was based on the need of the CFT/CAM-Centre to gain more insight and experience in the usage of formal specification techniques. Already some experiences with LOTOS as one of the formal specification techniques were presented, but the abstract syntax and the absence of computer support for verification appeared to be a great handicap within the acceptance of LOTOS as a mean for communication between designers and constructors of CAM products. Therefore an investigation was required on the possibilities to apply other formal specifications techniques, and on the usage in practical situations of formal specification techniques in general.

This document could be interesting for those who want to know more about formal specification techniques in general, the usage of LOTOS to specify CAM-Architectures, and experiences with the software support to verify LOTOS specifications.

Finally Mr. Sj.Sjoerdsma and Mr. M.Curley are acknowledged for the fruitful discussions I had with them about resp. applications of LOTOS and properties of the Transport System used within the CIMphony pilot.

Peter J.J. Bessems,
Eindhoven, Augustus 17, 1988.

Chapter 1

Introduction

1.1 Specification and verification using an FDT

Already for a long time the description of communication protocols or system architectures occurs mainly in an informal way. *Informal* means that no standarized description technique is used to specify these protocols or architectures. In the average the english language is used or the native language of the designer.
During the main part of the design process the designer describes each detaillingstep, from idea until the first realisation step, by informal means. Only when he starts the real implementation, i.e. the realisation of his concepts, he uses a more formal way of description such as a high or low level programming language.
This way of specifying and realising systems appears to have two awkward disadvantages.
The first is that only in a very late stage during the design process it is possible to verify whether or not some fundamental choices, already made in a very early stage of the design, are correct. During the first steps in the design process no formal verification is possible. So it can occur that already a lot of efforts (or money) is put in the development of a product when it appears that one of the first made very fundamental detaillingsteps in the design was incorrect, so some aspects of the product will be impossible to realize.
Undoubtedly it is much more efficient to minimize riscs of failure before starting the prototyping fase in a design process.
The second disadvantage of informal specification of systems is the possibility that a large amount of interpretations can be given to one specific description. Or, in the opposite, the behaviour of a system can be described informally in a lot of ways, meaning the same behaviour. So one can give a lot of meanings to one description, or a lot of descriptions to one meaning.
Therefore informal specifications does not improve the unity in interpretation of systems, and equivalence of systems are hard to prove.

The abovementioned disadvantages of informal specification emphasizes the need for Formal Description Techniques (FDTs). In the late seventies and the early eighties a lot of efforts were initiated to develop FDTs. Nowadays these efforts have resulted in some very powerful FDTs (see next chapter). Some of them already received the status of *Draft International Standard* (DIS) by the International Organisation for Standardization (ISO).

The aims of these FDTs are twofold:

1. With the aid of an FDT one should be able to verify detaillingsteps in a design already in a very early stage of the design process.

2. By standardization of description methods designers should be able to read and understand easily each others' specifications. It will be possible to prove equivalences in the behaviour of several separated systems, which improves the exchangebility of system modules.

The two aims are strongly related. Actually the rules to prove equivalence of systems are used to verify detaillingsteps in a design. With these rules one can prove whether the behaviour of a system after a detaillingstep is equivalent to the behaviour of the system before the detaillingstep.

As a derivation of the aims an FDT offers a lot of very useful qualities for the design of communication protocols or system architectures. Especially when the formalism is based on a algebraic structure. This algebraic formalism means that the behaviour of a system can be represented as an algebraic expression. In this case algebraic laws or relations between expressions can be defined or derived.
When the behaviour of systems is represented as an algebraic expression and algebraic laws are defined, verification is based on real mathematic fundaments which make an FDT much more powerful.

The real advantage of an FDT shows clearly when designing a communication protocol.
Communication protocols can be very complicated. Especially when a lot of concurrent processes have to communicate.
Without using an FDT, the verification of the protocol (e.g. if deadlocks can occur) can only be possible after a real implementation of the communicating processes. The protocols have to be worked out in all detailes, realized e.g. in a high level programming language, and implemented and simulated on a computer, before it can be proved whether the protocol causes deadlocks.
Using an FDT means that already during the design of the protocol the designer can prove the correctness of it. After each detaillingstep the new introduced protocol parts can be verified. This happens by verifying whether the composition of the new specified parts in the protocol behaves like the original specified protocol before the detaillingstep.
This way of specifying protocols suppresses the necessity to specify and implement all details before the protocol can be verified. Essential starting points of the protocol can be tested very early.

The next example (from [ST1]) of specifying and verifying a communication protocol gives a simple but very effective presentation of the great advantage of using an FDT.
The aim is to specify an interface between two systems, based on the handshake protocol.
As an FDT, Milners' CCS (*Calculus of Communicating Systems*, see [MIL80] and next chapter) is used. In CCS the behaviour of a system is represented as the behaviour of a finite state

machine. So a system is described as a sequence of possible input, internal or output actions which cause transition of the system from one state to another.
Normally an interface system performs a tranparent communication between two users.
So the specification S of a general interface module M in CCS should be (see fig. 1.1a):

S → M = m.in? : m.out! : M

This means that the original state of the interface module is M and after an input at gate m.in the module will present the input data at output gate m.out and will return to its original state.
As written, for the interface the handshake protocol will be used.
The next description is presented as an implementation I of the handshake mechanism (see fig. 1.1b):

I →

M1 = m1.in1? : m1.out! : m1.in2? : M1 and
M2 = m2.in? : m2.out1! : m2.out2! : M2

Now it should be verified whether the external behaviour of the implementation is equivalent to the behaviour of the original specification of an transperent interface. Then I is a correct implementation of S.
The equivalence of S and I ($S \approx I$) can be proved by verifying whether the parallel composition of the submodules M1 and M2, with hiding of the internal actions (I = (M1|M2)\P) shows the behaviour of S. (P={m1.in2,m1.out,m2.out2,m2.in})

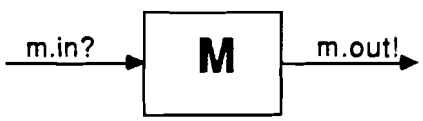


Fig.1.1a: Specification

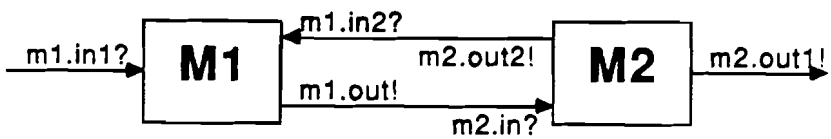


Fig.1.1b: Implementation

Interface based on a handshake protocol

For the verification Milners' expansion algorithm and CCS-laws are used:

$\mathbf{I} \rightarrow$

M = (M1|M2)\ P
= (m1.in1?:m1.out!:m1.in2?:M1 | m2.in?:m2.out1!:m2.out2!:M2)\P
input by M1:
= m1.in1: (m1.out!:m1.in2?:M1 | m2.in?:m2.out1!:m2.out2!:M2)\P
communication via m1.out and m2.in (this communication performs as an internal action and is therefore unobservable. This action is represented by τ):
= m1.in1?:τ:(m1.in2?:M1 | m2.out1!:m2.out2!:M2)\P
output by M2:
= m1.in1?:τ:m2.out1!:(m1.in2?:M1 | m2.out2!:M2)\P
communication via m1.in2 and m2.out2 (unobservable):
M = m1.in1? : τ : m2.out1! : τ : (M1|M2)\P

Applying the first CCS τ-law results in:

$\mathbf{I} \rightarrow$ M = m1.in1? : m2.out1! : (M1|M2)\P

Applying the CCS renaming law:

$\mathbf{I} \rightarrow$ M = m.in? : m.out! : M

So $\mathbf{S} \approx \mathbf{I}$

Now it is proved that the composite system shows the same behaviour as the original specification of the fully transparent interface.
The next step in the design of the handshake mechanism is realizing an implementation of the submodules M1 and M2.
Obviously, by first making a detaillingstep in the design ("top down") and then verifying the step by calculating the behaviour of the parallel composition ("bottom up"), the design process appears as a sequence of detaillingsteps of going down and up by specifying and verifying more details in the design.

Hence, the above presented example shows how powerful an FDT can be during the design of communication protocols. After each design step, already from the first stage of the design process, the correctness of the implementation can be checked.
As already stated, one does not need to put a lot of efforts in the design before some very fundamental choices can be verified.

1.2 CAM-Architectures

All over the world companies are working towards a unified approach of factories and production units. Increasing demands on flexibility, integration, and standardisation pose serious questions of how to control a production process. Also within Philips, with a large variety of factories and production strategies, this unified approach is searched for.
The Reference Model for Production Control Systems, developed by the CFT CAM-centre in co-operation with Digital Equipment Corporation (DEC), is a framework for production control modules. It seeks to model the complexity to be found between on one side the number of required products that must be delivered at a certain time, and the sensors and actuators that directly influence the medium and the product on the other side. This enormous complexity is tackled by a hierarchical model, where each level has a distinct functionality. The functionality is general in the sense that it describes almost all kinds of production (batch, assembly, cyclic work, etc).
This so-called CAM Reference Model [CAM87] (see fig.1.2) aims at integration of 'islands of automation'. The integration between the modules of the hierarchical levels, is described in the standard CAM architecture. This architecture, a description of all application messages between the different modules and their temporal ordering, tries to give help in defining and achieving standards components in the area of production control. A specific production environment can be mapped on the Reference Model, and the standard architecture is used to define the relation between the control components. The Reference Model and its corresponding architecture can be considered as control module blueprints.
The transport control system is part of the control hierarchy. This document deals with the control architecture of the transport system (TS-Architecture) the Transport System Architecture (TS-Architecture).
The Transport System (or "Transport/Movement System") is presented in the Reference Model ([CAM87] p.11) as a system that can be commanded to transfer products, materials and tools from one location to another. Since product transformation operations take place within the domain of a Workstation and the capabilities of a Workstation is limited, products must be transported from one Workstation to another Workstation. The Cell/Line Controller decides which Workstations have to be visited by a product to undergo a certain set of operations.
Above the cell/line level transport issues are not encountered, as one deals then with planning functions, etc. This document therefore only deals with a certain aspect of the levels cell/line, workstation, and AM, namely the transport of products, materials, tools, etc. It describes the relations between those layers in a general sense, and it defines the architecture of the transport control system as a part of the general Reference Model for Production Control Systems.

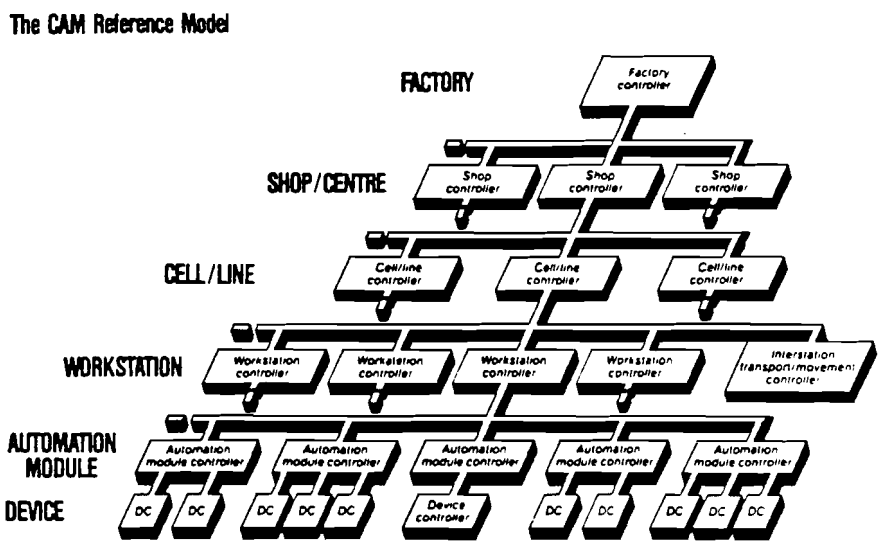


Fig.1.2 : The Philips/DEC CAM-Reference Model

Chapter 2

Specification techniques for CAM-Architectures

During the design of systems several specification techniques can be used. Actually two interpretations of the concept *specification* can be stated.
The first meaning of specification is the performance of a detaillingstep in a design process. The word specification is used for making a specification step as proposed by Koomen in [KOO88]. Specification means a step towards more specific details in a design, towards an implementation in software or hardware. Specification techniques in this sense are called Design Techniques or Development techniques. These techniques supports the designer in presenting more details in his design. Examples of these Design Techniques are Yourdon, DPDL, SADT, Cold-K, etc.
The second meaning of specification is description, so specifying a system only means describing a system. In this sense specification techniques are tools to describe a system. One of these tools is for instance a language. Whenever a standardized tool is used for the description of a system, the specification technique is called a Formal Description Technique (FDT) or a Formal Description Language. Examples of these FDTs are LOTOS, CCS, SDL, Estelle etc.
Of course these two kinds of specification techniques are strongly related. For many Design Techniques a formal description method is presented to describe specific stages in a design, and the formalism of Formal Description Techniques are often used to describe (and verify) detailling steps.
The scope of investigation is limited to the comparison of Formal Description Techniques. Comparison of Design Techniques has been done within the scope of other research projects.

For the design of CAM-Architectures many FDTs are available. However, only a few of them appeare to be interesting for further investigation.
One typical aspect of CAM-Architectures is that they are related to systems which can be devided in several subsystems which communicate parallel or sequentially.
The need to describe this concurrency between subsystems, especially the temporal ordering of their communication, makes it interesting to investigate those FDTs which represents processes as finite state machines.
Another argument which reduces the scope of FDTs for CAM-Architectures is that among

these FDTs some of them are strongly promoted by influential international organisations. Undoubtedly this quality offers a useful and powerful support to an FDT.
As a result of these considerations, only three FDTs for the specification of CAM-Architectures remain: LOTOS, Estelle (both ISO Standards) and SDL (CCITT-Standard).
Therefore these FDTs became subject to narrow investigation.
A fourth FDT (not standardized) is treated as well: CCS. The reason for this is that CCS was the first FDT which introduced an algebraic formalisms for verification of communication protocols. Later on, during the development of other FDTs, the algebraic fundation of CCS was used to add more fundamental verification tools to these FDTs.
Especially verification tools are very useful during the specification of CAM-Architectures because of the very complex temporal ordering of communication which can occur between CAM systems.
The next sections show a short review on CCS, LOTOS, Estelle and SDL and a subsequent conclusion of a comparison between them. For more details about the comparison is referred to [BES88].

2.1 CCS

In 1980 CCS (*Calculus of Communicating Systems*) was introduced by Robin Milner at the University of Edinburgh, Scotland, [MIL80].
The introduction of CCS was based on two priniples (from [MIL80]):
"The first is observation; we aim to describe a concurrent system fully enough to determine exactly what behaviour will be seen or experienced by an external observer. (...) This by no means prevents us from studying the structure of the system. Every interesting concurrent system is built from independent agents which communicate, and synchronized communication is our second central idea."
The first consideration resulted in the concept of observation equivalence ([MIL80]): *"two systems are indistinguishable if we cannot tell them apart without pulling them apart"*, and the second consideration resulted in the introduction of the parallel composition ([MIL80]): *"a binary operation which composes two independent agents, allowing them to communicate"*.
Milner introduced a calculus to describe communicating processes. He represented systems as behaviour expressions and derived rules to prove observation equivalence.
This calculus appeared to be a very strong tool for specifying and verifying communication protocols, and makes CCS a very powerful FDT.
Paradoxically this strong tool causes at the moment also some inconveniences. Verification by proving observation equivalence can be a rather time consuming business in case of complex systems. So without some computer support it even could be impossible. At the Eindhoven University of Technology some software support has been developed. The software performs some basic tools to analyze simple CCS specifications. Unless more features are added to the software tools (e.g. abstract data structures or a behaviour tree generator), CCS can not be used on its full power.
For a general introduction on CCS is referred to [KOO88].

2.2 LOTOS

LOTOS (Language of Temporal Ordering Specification) was specially developed for formal description of OSI standards. This FDT has been developed by an ISO Task Committee and received already the status of *Draft International Standard* [LOT87].
LOTOS is based both on CCS and on the abstract data type language ACT-ONE. CCS is used for process description, ACT-ONE is used for data type definitions. (ACT-ONE has been developed at the Berlin University of Technology, West-Germany, [EHR83]).
For a general introduction on LOTOS the reader is referred to [BRI85] and [SCO86].
To give an impression of the LOTOS syntax a LOTOS specification of the M1 module of the handshake interface in fig.1.1b (chapter 1; 'm1' is represented by 'a') is presented in fig.2.1.

```
process A[a.in1,a.in2,a.out] :=
          a.in1
        ; a.out
        ; a.in2; A[a.in1,a.in2,a.out]
endproc
```

Fig.2.1: LOTOS specification of the M1 module of the handshake interface, see also fig.1.1b

At the moment a lot of research is done on LOTOS. Within several companies, universities and joint research projects (e.g. the EC ESPRIT Project) more experience is gained with the use of LOTOS and many efforts are put in the definition and derivation of more algebraic laws and relations between behaviour expressions.
At the University of Twente a comprehensive package of software tools (the SEDOS LOTOS Toolset, see fig.2.2) can be obtained to support the analysis and verification of LOTOS specifications. A syntax checker, a static semantics checker, a symbolic executer, a communication tree builder and some tools for test sequence analysis are part of the software. At the moment some research is done to develop two rather missed tools: a test sequence generator which automatically generates (all) possible traces of input actions, and an interactive compiler which should transform a LOTOS specification into a C programme.

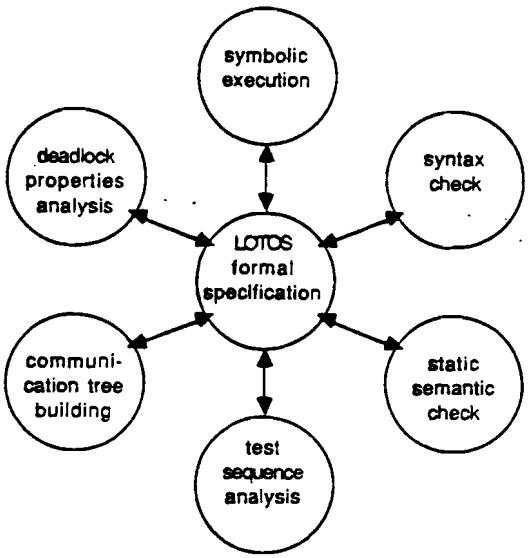


Fig.2.2: The SEDOS LOTOS Toolset

2.3 Estelle

Estelle (Extended State Transition Language) is developed by an ISO Task Committee in cooperation with the CCITT to describe data communication protocols. Estelle has been developed specially for OSI data transport protocols and received already the status of *Draft International Standard* [EST87]
An Estelle specification shows many similarities with a Pascal programme, especially ISO-Pascal, because of the modular structure.
For a general introduction on Estelle is referred to [LIN85]. Fig.2.3 gives an impression of the extensive syntax of Estelle. Again the M1 Module of the handshake interface (fig 1.1b; 'm1' is represented by 'a') is used for specification.
One strong aspect of Estelle is that it can be used to specify in detail the communication between systems. It offers a lot of possiblitities to describe many sorts of interfaces using specific type definitions for specific kinds of interfaces and special type names for some sorts of sequentially or parallel operating processes.
At the same time however, these many tools for specifying the communication behaviour of systems handicaps the use of Estelle. The comprehensive syntax definitions do not improve a fast and efficient use of the language, or increase its readability.
The comprehensive syntax also makes Estelle unsuitable for an algebraic substructure, which decrease the possibilities for formal verification. Nevertheless, the extensive overhead of definitions, declarations, initializations, etc. in the syntax are a great advantage in compiling a Estelle specification into a Pascal programme. An Estelle specification is very near to a real

implementation: it is almost equal to a Pascal programme.

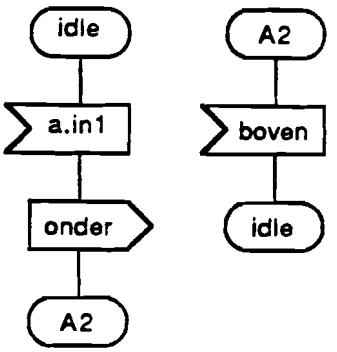
```
Module A process;
       (declaratie interaction points met instances, declaraties variabe'
        parameterlist);
        a.in1, a.in2: (channel type);
        a.out: (channel type);
end A;

trans
       (priority/provided/any/delay);
       from A
       to   A1
            when a.in1
                 begin (veranderingen binnen module A,
                        initialisatie intances, etc);
                 end;

trans
       (priority/provided/any/delay);
       from A1
       to   A2
            when none
                 begin  (veranderingen binnen module A,
                        initialisatie intances, etc);
                        output a.out;
                 end;

trans
       (priority/provided/any/delay);
       from A2
       to   A
            when a.in2
                 begin  (veranderingen binnen module A,
                        initialisatie intances, etc);
                 end;
```

Fig.2.3: Estelle specification of the M1 module of the handshake interface, see also fig.1.1b

2.4 SDL

The language SDL (Specification and Description Language) is developed by Study Group SG XI of the CCITT, a group who was charged with the set up of recommendations in the field of switching techniques and signalling within telephone communication. Since 1976 SDL belongs to the official "Recommendations" of the CCITT. The last changes have been made in 1984 [SDL84].
For a general introduction on SDL is referred to [ROC82].
As written, SDL was specially developed for use within telephone switching. However, because the most protocols within telephone switching are based on the OSI Communication Model, SDL is also suitable for other communicatiing systems.
Originally SDL was a graphic language (SDL-*Graphics*). Later on, a programming-language-like version of SDL was introduced (SDL-*Linear*).
Fig.2.4 shows an SDL specification of the M1 module of the handshake interface of fig1.1b ('m1' is represented by 'a'), both an SDL-Graphic representation and a SDL-Linear representation.
As appears, SDL specifications are easy to read. Especially the graphic method is very surveyable and very suitable for the functional design of communication protocols.
In the opposite, one disadvantage of SDL is that no possibilities exist to specify explicitly the composition of systems. So the behaviour of two composite processes can only be checked after implementation and simulation on a computer (software tools are available to simulate SDL specifications). At the moment some research is done on the introduction of CCS as an algebraic substructure for SDL. With the aid of CCS formal verification of the composition of systems will be possible within SDL.
Other research is done on a compiler which should be able to transform SDL specifications

into a Chill programme.
As far as data type definitions are concerned: since last year SDL has been extended with abstract data types.




```
PROCESS A

   STATE idle
      INPUT a.in1
         OUTPUT SEND onder TO B
      NEXTSTATE A2
   STATEND idle

   STATE A2
      INPUT boven FROM B
      NEXTSTATE idle
   STATEND A2

ENDPROCESS A
```

SDL-*Graphics* SDL-*Linear*

Fig.2.4: SDL-specification of the M1 module of
the handshake interface, see also fig.1.1b

2.5 Conclusions

After a detailled analysis of the formal description techniques CCS, LOTOS, Estelle and SDL (see [BES88]) it appeared that all of them are more or less suitable for the specification of CAM-Architectures. Especially the representation of processes as finite state machines, and the narrow relationship with communication protocols, emphasize the advantage of using these FDTs within a CAM systems development environment.
Nevertheless, each of the four FDTs has its own specific advantages and disadvantages.
During the analysis of them, some aspects emerge which were finally used as criteria to compare the languages.
These aspects are:

1. the complexity of the syntax,
2. the existence of an algebraic substructure,
3. the possibility to define data types,
4. the way how the composite behaviour of systems can be specified in detail
5. the existence of software support for verification
6. the difficulty to realize a specification in software or hardware.

The next considerations are the results of a comparison of CCS, LOTOS, Estelle and SDL, based on the abovementioned criteria.

The algebraic verification tools within CCS make that CCS can be a strong tool when designing system architectures or communication protocols. The property that CCS is used as a fundament to add algebraic verification techniques to other FDTs, shows how important CCS can be during a fundamental analysis of synchronization behaviours. But the absence of abstract data types and the only in a development stage being software tools for analyzing communication trees, are a disadvantage in the use of CCS in practice.

Estelle appeared to be especially dedicated to data communication protocols and offers very interesting tools to define interfaces in many details. Because of the comprehensive syntax and the property to describe a lot of details in declarations, definitions, initializations an Estelle specification is easy to implement in software (Pascal). This property also causes the biggest disadvantages of Estelle. The extensive syntax makes an Estelle specification difficult to read and unsuitable for a algebraic substructure for formal verification.

An SDL specification is very easy to read, especially a SDL-Graphic version. SDL can be a quit surveyable support for specifying the functional design of communication protocols. However, it is not possible to describe the composition of systems. The composite behaviour of systems can only be verified by simulation, not by algebraic derivation. It should be mentioned that this situation can change in the near future because CCS is introduced as an algebraic substructure for SDL.

LOTOS appears to cope with all the disadvantages of the other three languages. LOTOS has a rather simple syntax, uses an algebraic substructure, contains abstract data types and can be supported by an extensive software package with verification tools. These qualities make LOTOS to be a very powerful description technique.

The conclusion of the abovementioned considerations is obvious: at the moment LOTOS appears to be the best description technique for formal specification of CAM-Architectures. It has to be emphasized that this conclusion is made in the year 1988. Nowadays a lot of developments are going on to bring other FDTs in the same position as LOTOS at the moment. So a comparison after a few years from now could result in an other conclusion. It seems that all FDTs converge to a CSP-like description technique with data types and an algebraic substructure.
Fig. 2.5 presents a general overview of the advantages (+) and the disadvantages (–) of the FDTs based on the six criteria.

.	CCS	LOTOS	Estelle	SDL
syntax	+	+	– –	++
verification	+	+	–	–
data types	–	+	–	+
composition	+	+	++	–
software tools	+	++	–	–
realization	–	–	+	–

Fig.2.5 : Advantages and disadvantages of
CCS, LOTOS, Estelle and SDL (anno 1988)

Finally, it should be remarked that the developments on FDTs are very recent. Only in the late seventies the first proposals for standardized description techniques appeared, and only in 1980 Milners' CCS gave the first drive to algebraic verification of specifications.
This is the main reason why FDTs are not yet used within a large range of research and development projects. Many specifications of systems or products are still made informally.
Another reason why FDTs are still not yet used by many designers is that FDTs are seen as too abstract and too difficult to read. Designers rather like to use their own specific informal description method which is easier for them to read and easier to implement in software or hardware. Although all of them admit that a uniform way of specifying different system architectures would improve the efficiency during integration of system parts.
So as long as no means exist to decrease the high abstraction level of an FDT, and no means exist for automatic transformation of specifications into software or hardware, FDTs will only be used by a small group of designers.

Chapter 3

TS-Architecture vs.1 : ITS

The transport system as defined in the CAM Reference Model [CAM87] is a system that can be commanded to transfer products, materials and tools (hereafter "products") fromone location to another.
As an appendix to the Reference Model the *Philips Reference Model of Internal Transport Systems* (ITS) is presented [CAM87].
Fig. 3.1 shows the internal structure of the ITS. The model resembles the well-known Open Systems Interconnection (OSI) Model of data communications. The basis of the choice for this model was the notion that data transport systems and product transport systems have much in common.
Each "tower" in the Model belongs to one Node (dicision point) in the transport system (see also fig.3.2).

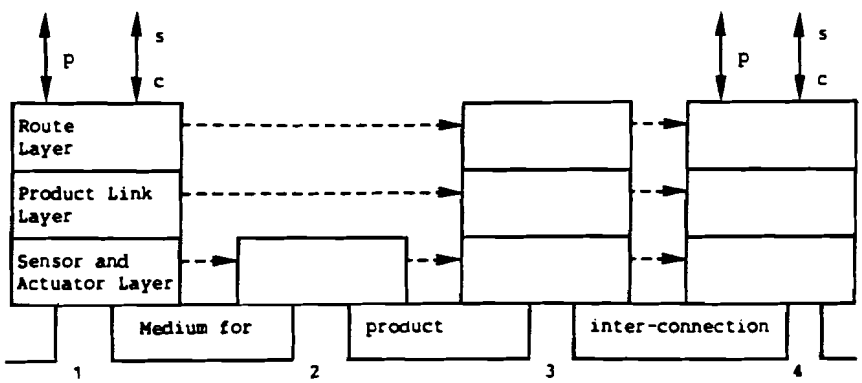


Fig.3.1: The Reference Model of the Internal Transport System (ITS)

p : product exchanges with the transport system
s : status exchanges with the transport system
c : command exchanges with the transport system
1,2,3,4 : Nodes within the transport system

The *Medium for Product Inter-connection* allows a passive conduction of products and possible product data, by transition of forces on products and stimulation of sensors. It is the physical layer of the transport system.
The *Sensor and Actuator Layer* offers a transport over a "standard displacement". This standard displacement is the most elementary displacement of a product (and product data) that can be executed by the transport system. Examples of standard displacements are lifting, rotating and transferring displacements.

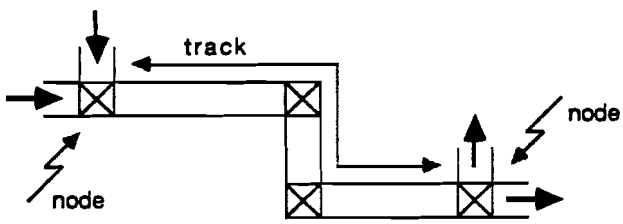


Fig.3.2: Definitions of Track and Node

The functions of the *Product Link Layer* detect the loss of products, prevent collision between products, check the number of products on the track and command the standard displacement offered by the Sensor and Actuator service. The Product Link Layer controls a sequence of one or more standard displacements which a product must undergo to be transported along a track (see fig.3.2). A *Track* connects two *Nodes*. A Node is a point at which there is a choice of directions that a product can be dispatched, and/or where a product can be stopped because it is at its destination, or because other products take precedence.
The functions of the *Route Layer* determine a route to a given destination. They can control the transport along a sequence of tracks. This as distinct from the Product Link Layer which can only control the transport along one track, i.e. between two adjacent Nodes. The Route Service can be summarized as a transparent transport of products (and product data) between two users.

In [BLO87] Blonk and Biemans show the first version of an Arcitecture for the transport system, which was based on the CAM Reference Model. For this TS-Architecture vs.1 the above described Philips ITS Model was used.
In [BLO87] both an informal (english) and a formal (LOTOS) specification of the ITS-Architecture is presented.
Blonk and Biemans were not able to verify their formal specifications, because the verification would have been too time-consuming without some computer support.
However, since January 1988 a software package exists for the verification of LOTOS specifications (the SEDOS LOTOS Toolset, see fig.2.2). In the next sections a simplified version is derived from the LOTOS Route Service specifications of the ITS, and the results of the verification with the SEDOS LOTOS Toolset are presented.

3.1 Informal specification ITS

For convenience only the specification of the Route Service is used for verification. The Route Layer is the only Layer which services interfaces with external systems (Workstations and Cell). The implementation of the Product Link Services and the Sensor/Actuator Services would not interfere the synchronization within these interfaces in an important way.

The Route Service Specification presented by Blonk/Biemans [BLO87] was the first version (TS-Architecture vs.1) based on the ITS Reference Model.
As already stated, Blonk/Biemans were not able to verify their specifications. Here, a simplified specification of the Route Service is created to obtain a useful input for verification with the SEDOS LOTOS Toolset. For this simplified specification also some new ideas from Blonk are used, based on personal correspondence.

The simplified specification of the Route Service has the following characteristics:

- no product data can be attached to a product
- only direct addressing is possible
- only one product can be sent with one command
- only primitives for the sending and receiving of a product are available (no report, sequencing, reset, etc. primitives)
- products can neither be lost nor spontaneously appear
- there are no command access points

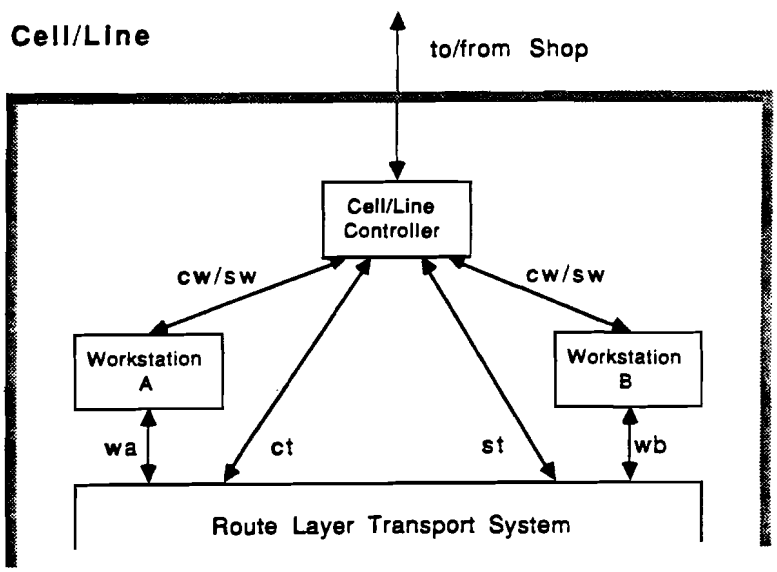


Fig.3.3: The ITS within its environment

Fig. 3.3 shows the Transport System within its environment in the Reference Model. The Workstations A and B exchange products with the Transport System, and the Cell/Line Controller exchanges commands and status with the Transport System, the Workstations and the Shop Controller.

For the simplified LOTOS specifications the ITS will be represented as a process with four communication gates (see fig.3.4). The The Route Service is specified between a sender A and a receiver B. The process has the gates ct and st to exchange commands and status with the Cell, and gates wa and wb to exchange products with A and B respectively.

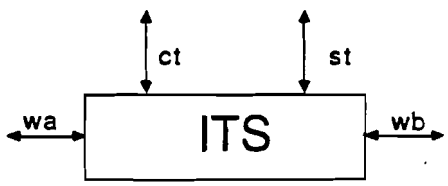


Fig.3.4: Representation of the ITS as a process
with four communication gates.

In the Route Service Specification this process has been implemented as the parallel composition of five concurrent processes (see fig.3.5) which are synchronized via three gates (sync1, sync2 and sync3).

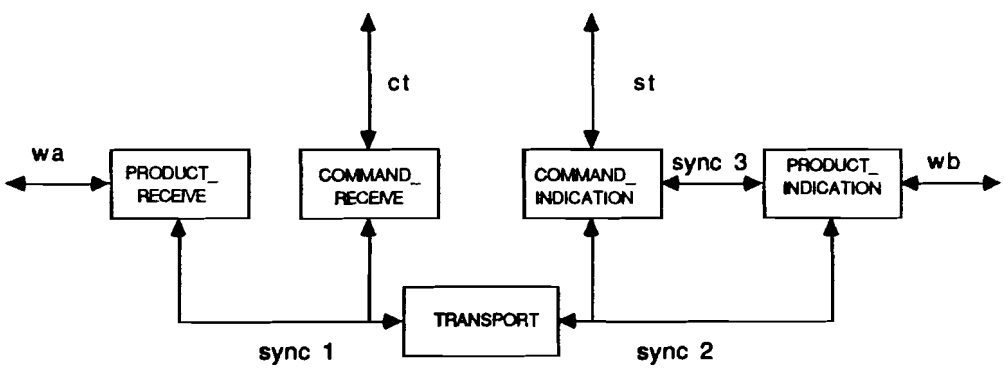


Fig.3.5: Implementation of the ITS as the parallel
composition of five concurrent processes.

The five processes deal with the exchange of a product from a sending Workstation A (the process PRODUCT_RECEIVE), the exchange of commands with the Cell (COMMAND_-RECEIVE), the transport of the product from a (address of Workstation A) to b (address of Workstation B) (the process TRANSPORT), the exchange of status with the Cell (COMMAND_INDICATION) and the exchange of a product with the receiving Workstation B.
Fig.3.6 shows the data types which are necessary to define the representation of the messages within the simplified ITS-Architecture.

PRIMITIVES	PARAMETERS
R_PRODUCTrequest R_PRODUCTindication	product
R_PRODUCT_SENDrequest	source_address destination_address required service quality required options number of products commander_data
R_PRODUCT_SENDacknowledge	source_address provided service quality
R_PRODUCT_ARRIVALindication	destination_address source_address number of products commander_data

Fig.3.6: Route Service primitives and parameters.

3.2 formal specification ITS

The simplified LOTOS specifications of the Route Service are presented in Appendix A.
In this specification all original data type definitions are presented. Their consistancy and static semantic have been checked and made errorless.
The actual transport of the product is represented as an internal action in the process TRANSPORT. It has to be implemented within a next detaillingstep in the design, which also implements Product Link Services. In the present specifications the process TRANSPORT contains a set which represents the products in the system. Sending a product means inserting a product to the set, delivery of a product means removing a product from the set.

3.3 Results of verification with SEDOS LOTOS Toolset

The SEDOS LOTOS Toolset appeared to be an interesting support for checking and verifying LOTOS specifications.
With this Toolset first the comprehensive part of data type definitions for the Route Service vs.1 has been checked on syntax, semantic and consistency.
A Standard Library of data types is implemented in the ISO/DIS 8807 LOTOS Standard. So the original data type definitions of the Route Service have been rewritten to make them consistent with the definitions in the Standard Library and to offer the possibility to link them with the Standard Library.

When the data type definitions were checked, debugged and installed errorless and consistently, the syntax and semantic of the Route Service process description part has been tested.
Because the used specification describes a very simple architecture, no much difficulties

showed up during the debugging fase.
(Note: Only one rather complicated problem arose. The way how a "Set" is defined in LOTOS is not very useful. A lot of renamings and actualisations of formal data types are necessary, and all types of elements of the set and their ordering have to be known on forehand.)

Fig.3.7 shows a possible path in the behaviour tree of the ITS specifications.

```
0    START
1    ca          !r_product_sendreq(a, b, no_qos, empty_reqs, 0, no_comdata)
2    wa          !num_1
3    i(sync1)    !r_product_sendreq(a, b, no_qos, empty_reqs, 0, no_comdata)
                 !r_product(num_1)  !no_wsdata
4    wa          !num_5
5    ca          !r_product_sendreq(a, b, no_qos, empty_reqs, 0, no_comdata)
6    i(sync1)    !r_product_sendreq(a, b, no_qos, empty_reqs, 0, no_comdata)
                 !r_product(num_5)  !no_wsdata
7    i           { rsobject(r_product(num_5), b, a, 0, 0, empty_reqs,
                   no_comdata, no_wsdata) }
8    i(sync2)    !rsobject(r_product(num_5), b, a, 0, 0, empty_reqs,
                  no_comdata, no_wsdata)
9    wb          !b  !r_productind(r_product(num_5))
10   i           { rsobject(r_product(num_1), b, a, 0, 0, empty_reqs,
                   no_comdata, no_wsdata) }
11   i(sync3)    !product_delivered
12   cb          !r_product_arrivalind(b, a, 0, no_comdata)
13   i(sync2)    !rsobject(r_product(num_1), b, a, 0, 0, empty_reqs,
                  no_comdata, no_wsdata)
14   wb          !b  !r_productind(r_product(num_1))
15   i(sync3)    !product_delivered
16   cb          !r_product_arrivalind(b, a, 0, no_comdata)
```

Fig.3.7: A possible path in the behaviour tree of the ITS, incl. internal actions.

So fig.3.7 shows that two products (num_1 (first) and num_5) are transported. In case of num_1 the command was received before the product was arrived, in case of num_5 the product was arrived before a command was received. However, for both products first the product was delivered to the receiving Workstation before the Cell was indicated about the arrival of the products at their destination. This was exactly as expected.

The LOTOS Toolset also offers the opportunity to show the behaviour of composit systems with hiding of all internal (unobservable) actions. Fig.3.8 shows this behaviour based on the path of fig.3.7

```
0    START
1    ca  !r_product_sendreq(a, b, no_qos, empty_reqs, 0, no_comdata)
2    wa  !num_1
4    wa  !num_5
5    ca  !r_product_sendreq(a, b, no_qos, empty_reqs, 0, no_comdata)
9    wb  !b  !r_productind(r_product(num_5))
12   cb  !r_product_arrivalind(b, a, 0, no_comdata)
14   wb  !b  !r_productind(r_product(num_1))
16   cb  !r_product_arrivalind(b, a, 0, no_comdata)
```

Fig.3.8: A possible path in the behaviour tree of the ITS, only observable actions.

Fig.3.8 shows the external behaviour (architecture) of the ITS process in fig.3.4. The behaviour appears as expected.
Nevertheless, this is only one path of the tree. For real verification all possible paths in the behaviour tree should be analysed. Actually, real verification should take place by representing the full behaviour tree as a behaviour expression, representing the expected behaviour of the ITS as an expression and verifying whether both expressions are equivalent.
Because the present behaviour tree is very large, and the Toolset does not generate automatically test sequences or behaviour expressions from trees, this verification was impossible to realize. Verification was done by only checking the "risky" paths in the tree.
However, full verification stays possible by verifying all path in the behaviour tree.
Now it is showed that already in a very early stage of the design (which the simplified specification presents) verification is possible.
Thereby, it has to be mentioned, that verifying the synchronisation within the communication appeared to be the most time-consuming activity. The value passing almost always was correct after verification.

Chapter 4

Reconsiderations

The first approach to an implementation of the ITS-Architecture was made within the CIMphony project set up by the CFT. To gain more experience in the fully integrated control of a production facility, the project realized a facility which consists of four Workstations, a Cell Controller, several interchangable warehouses and a transport system. The facility, called the *CIMphony Pilot Production Facility*, can assemble several types of shavingheads for different types of Philips shavers.
The facility control is fully automized, based on the CAM-Reference Model control hierarchy. From one terminal all characteristic requests for production (e.g. how many shaving heads have to be produced of a specific type) can be offered to the facility, and all required data about the production progress (e.g. test data) can be received. The main target of the transport system is to transport trays with product parts from one Workstation (or Warehouse) to another Workstation (or Warehouse).
The CIMphony experiment showed a main disadvantage of the fully distributed transport control as presented in [BLO87]. The "router", implemented in the CIMphony Pilot, appears to be not optimal: it requires intensive data communication with the Cell Controller. For the transport of the products passive labels (bar codes) are used on the trays. These passive labels only contain the identity of the tray, not its destination or other information about its transport. So each Node has to communicate with the Cell about the destination of a tray. Obviously, with a lot of trays and a lot of Nodes the communication between the transport control system and the Cell controller will be extensive. The communication with the Cell will be even more when some monitoring of the transport is required. Then an overall view facility, e.g. for line balancing or traffic control, has to be implemented in the Cell Controller because an overall view of the transport can hardly be obtained with a fully distributed control. In this case it would be more efficient to implement these monitoring function in a "router" within the transport control system.
At two other places within Philips controllers for transport systems are developed.
The first one is the Machine Fabriek Alkmaar, part of the Division Industrial & Electro-acoustic Systems (I&E) where the commercial product VTS (Variable Transport System) is manufactured. This transport system is modular based with respect to the mechanical and electrical characteristics. In Alkmaar the need exists for a more modular approach also of the control part of the VTS. They also wanted to develop controllers which were possible to connect with other controllers within an integrated factory control system.

Another place within Philips where controllers for transport systems are developed is the Hasselt plant, part of the Division Consumer Electronics (CE). The CE Division at the presence produces its own transport controllers, based on the CAM-Reference Model. Nevertheless, these controllers are not suitable to connect with the controllers of the I&E VTS. Therefore, CE wanted to agree on a common interface so that they are able to connect their controllers with the I&E VTS controllers.

The experiences with the ITS-Architecture within the CFT, together with the experiences of I&E and CE on transport controllers, emphasized the need for a corporate approach towards transport control systems, and a strong reconsideration of the existing views on transport control.
This was actually done by a cooperation group of CFT, CE and I&E representatives.
The reconsiderations made by this group resulted in the set up of detailed user requirements, and a new view of transport within the CAM-Reference Model. The results are published in [BEU88].

4.1 User requirements

This section presents a very short overview of the general user requirements set up by Beukeboom et al. [BEU88]. It is presented here because these user requirements were used to define the functional requirements (next chapter) of a new transport control system architecture. For more details is referred to the original document.

The requirements are derived from four representative projects that FAS Alkmaar has done in the last years. The requirements are subdivided in four groups:

- *Routeing Functions*
  More general routeing features are required in case of parallel Workstations. *Parallel* means that a certain operation on a product can be executed on several locations (Workstations) with the same performances, so a choice has to be made for on of these locations.
  Other required features on routeing functions are dynamic routeing and priority handling. Dynamic routeing means that the route of a product may change just before or during the transport, based on changes within the production system (disfunction of a Workstation), changes within the dynamic properties of the transport system (track occupation, collision avoidance, synchronisation of throughput) or changes of product status (as a result of treatment (testing) by a Workstation).
  Priority handling offers the possibility to introduce precedence rules at (complicated) intersections of tracks (i.e. Nodes).

- *Product Recognition*
  For some (parts of) transport systems no product recognition is required, for others it is. In a simple flow Line that works with batches or with ranges of identities, the products need not to be identified individually and commands need not be given to every product. Of course some supervisory level must ensure that the right products

enter at the right place.
When more features have to be implemented (sequencing, error handling, priority handling, etc.) labels can be attached to a carrier or a product. These labels may be *passive* or *active*. Passive labels (e.g. bar codes) are rigidly attached to the carrier or product and their information is fixed. Active labels offers the possible to read and write data frequently in a label physically attached to a carrier or a product. This data can contain information about the status of the product (test results) or can also give information to the transport control system which is necessary to derive the destination of the product. With labels, one gets in principle a flexible control. For every individual product a specific route can be set out and (parts of) the execution can be delegated to the local controllers. Active labels even provide possibilities for fully distributed control.

- *Control System Requirements*
  These requirements deal with the physical controllers within the transport system. Facilities are required to link the controllers with other systems. Full integration and exchangebility with other CAM components are desired to obtain automated data collection and control.
  On the other hand, also stand-alone operation of the transport System is required. This can occur in a situation when the transport control system is the only controller for a complete production Line.
  Finally error detection and recovery request typical qualities of the transport controllers. Error recovery from a temporary power failure or communication failure should be possible.

- *Requirements from the Operator/User*
  Some requirements have been set up specially about the interfaces of the transport control system with the *Production Preparation System* (PES), the *Production Evaluation System* (PES) [HEH87] and a possible user/operator of the system. (see fig 4.1)

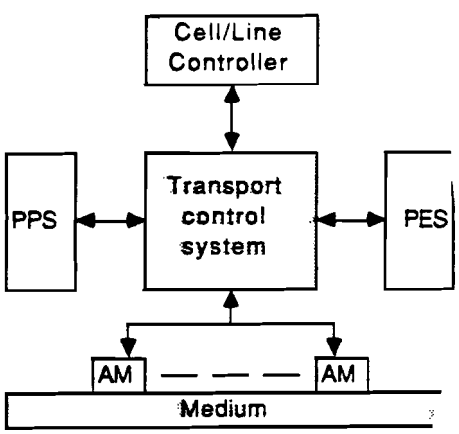


Fig.4.1: Interfaces with the Transport Control System

The PPS provides data to the transport system concerning how the commands from an user/operator or controller must be carried out. The PES receives information about the transport, evaluates this information and possibly delivers the results of these evaluations to the PPS or user/operator. For more details of the PPS and the PES is referred to [HEH87].

4.2 Analogy Transport System and Workstation

In [BEU88] Beukeboom et al. distinguish three possible users of the transport system: the Workstation Controller, the Cell/Line Controller or the Inter-Cell Controller. All of them present their own specific commands to the transport system.
A workstation Controller as commander of the transport system sends commands like: *"move the present object from one location (*track*) to another location (*track*)"*.
A Cell Controller as a commander sends commands like: *"transport product of type P from A (*Workstation*) to B (*Workstation*)"*.
And the commands from an Inter-Cell Controller are like: *transport N products of type P from A to E via B,C and D.*
In [BEU88] it is shown that these commands are related. An Inter-Cell command actually represents a sequence of Cell commands, so Cell commands can be derived from Inter-Cell commands. The same appears for the commands from a Cell: they represent a sequence of Workstation commands, so the Workstation commands can be derived from a Cell command. If the Inter-Cell commands are treated to be sent to the Transport Control System as sequences of seperated Cell commands (this does not influence the general functionality of the router; it does not really matter whether the decomposition of the Inter-Cell commands takes place in the Inter-Cell Controller or in the router), and *Composite Nodes* as described in [BEU88] are treated as a set of individual Nodes, a simplified representation of Beukebooms Layering of the Transport Control System ([BEU88] p.38) can be shown (see fig.4.2).

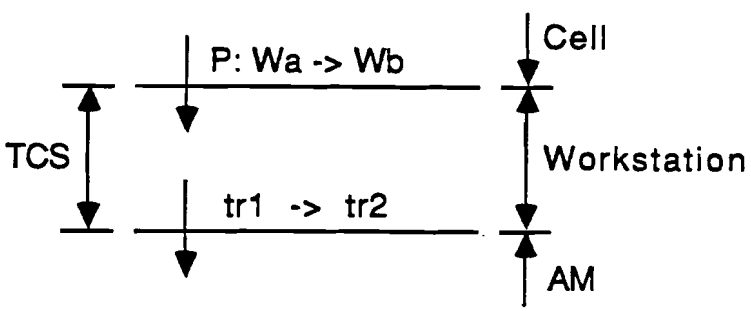


Fig.4.2: The TCS within the CAM-Reference Model

The simplification is made to show that, in principle, the TCS has the functionality of the controllers at Workstation level within the CAM-Reference Model. Commands from higher

layers are product type dependent, while commands to lower layers are product independent. Because the Workstation level in the Reference Model is the lowest level which has to keep product information, the TCS can be placed on Workstation level within the control hierarchy, see fig.4.2).
This seems quite obvious, but it represents a new view on transport so it deviates from the way transport is modelled in [CAM87].

Many of the general user requirements (parallel Workstations, dynamic routeing, sequencing, synchronation of through-puts, etc) demand for monitoring functions within the TCS, functions which keep an overall view of the transport of products and take dicisions about routes.
In a fully distributed control, without using active labels or implementing a lot of intelligence and global knowledge in every Node, these functions have to be implemented in the Cell Controller. The experiences with the CIMphony project (see the beginning of this chapter) proved that this is a very disadvantageous option.
On the other hand, a fully centralized control of the TCS causes an extensive data communication between the Nodes and the central controller, and is too much dependent on real time performances of the transport (the exchange of information between a Node and central controller about the destination of a product only can occur at the moment a product is actually presented at the Node.)
The abovementioned considerations about fully distributed and central control created the notion of partially distributed control. In general one can say that the monitoring functions will be implemented central and executing functions will be implemented local. This causes a great advantage: no intensive data communication is necessary between Cell and "router" or between router and Nodes, the control is not too much dependent on real time performances of the transport, and no strong intelligence or global knowledge have to be implemented in the Nodes.

Chapter 5

TS-Architecture vs.2 : TCS

In chapter 4 a short overview is given of the reconsiderations which have been made by Beukeboom et al. in [BEU88] about the fully distributed ITS-Architecture and the existing transport control systems realised by MFA, I&E and CE within Philips. These reconsiderations resulted in the set up of detailed user requirements, and a new view of transport within the CAM-Reference Model control architecture. The results are also described in [BEU88], and a short overview is given in chapter 4 of this report.
The reconsideration, the user requirements and the new view of transport within the CAM-Reference Model leads eventually to the description of a new transport system control architecture. This new version (vs.2) of the TS-Architecture will be reffered to as the Transport Control System (TCS) Architecture.
This chapter presents a LOTOS implementation of the functionality and the architecture of the TCS in analogy with the functionality of controllers at workstation level in the CAM-Reference Model. The functional requirements are based on the user requirements presented in [BEU88].

Fig.5.1a and Fig.5.1b show the main differences in view on transport control between the ITS-Architecture and the TCS-Architecture.

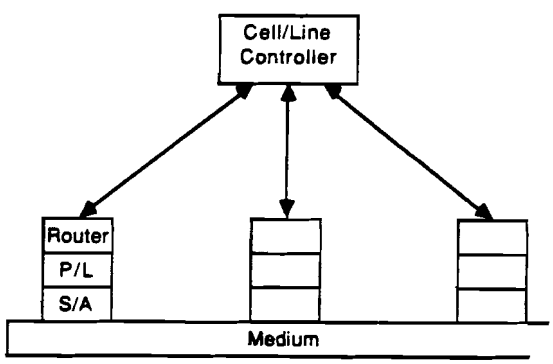


Fig.5.1a: ITS concept of fully distributed control of transport. The TS control is a separated part within the CAM-Reference Model.

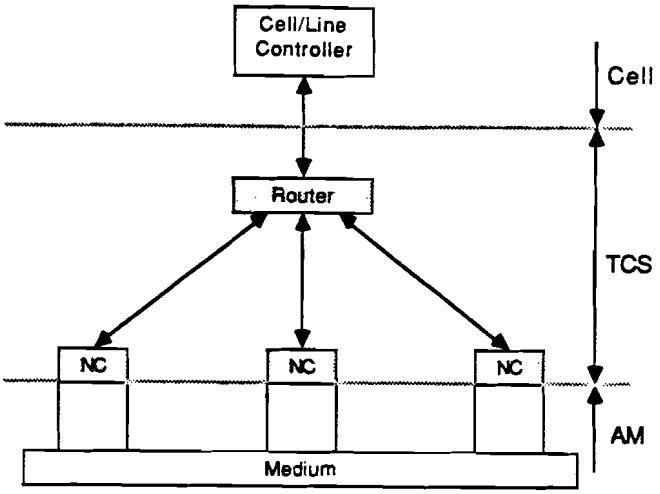


Fig.5.1b: TCS view on transport control.
The TS controller is a controller at Workstation level within the CAM-Reference Model

As shown the control architecture of the TCS is split into two levels: the ROUTER and the Node Controllers (NC). Both levels contain their own specific functionality. Some TCS functional requirements are realised by the ROUTER, others are realised by the Node Controllers. So some control functions of the TCS are centralised, others are distributed. The advantadges of a partially distributed control are described in chapter 4. The partially distributed control appears to have much more in common with practical situation than the ITS control.

Like the ITS, the TCS wil be represented in the formal specifications as a process with command and status gates (resp. ct and st) to the Cell Controller and for each Workstation a gate to exchange products (wa, wb, wc, etc.). See Fig.5.2.

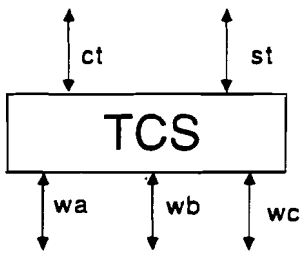


Fig.5.2: Representation of the TCS as a process with five communication gates.

In the formal specification the architecture of the TCS will be represented as the parallel composition of the ROUTER and the Node Controllers with hiding of the interactions between them. See fig.5.3. (To represent the set of products which can be present on a track between two Nodes, the parallel composition contains processes called "TRACK" as well).
The specification of the TCS wil be something like:

```
hide "internals" in

  ROUTER
||
  ( NODE(one)
  ||
    TRACK
  ||
    NODE(two)
  ||
    TRACK
  ||
    NODE(three)
  )
```

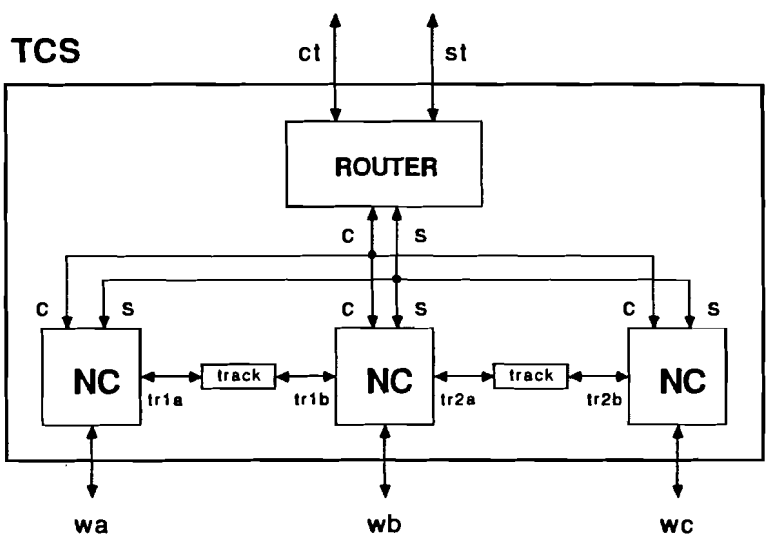


Fig.5.3: The TCS as the parallel composition
of ROUTER, Node Controllers and Tracks.

The main activities of the ROUTER are to decompose tasks from the Cell, and to process status data from the Nodes.
The main activities of a Node Controller are to combine a received product with a (possible not yet) received command for that product, and to control the Automation Modules (AM) to move the product to the required track.
The general commands from the Cell/Line controller to the ROUTER are like "transport a product of type X from Workstation A to Workstation B".
The general commands from the ROUTER to the Node Controllers are like "move a product of type X from track 1 to track 3".
The general commands from Node Controller to the AM-layer are like "move lift from track

1 to track 3".
Fig.5.4 presents the general commands which pass through the two layers in the transport control.

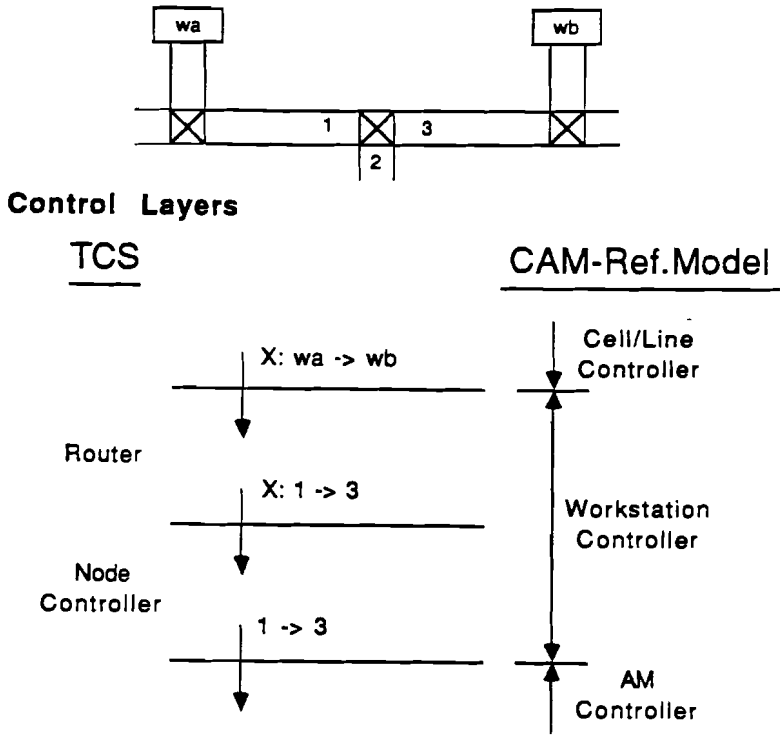


Fig.5.4: Control layers within the TCS.

As mentioned, the main activities of the ROUTER are to decompose tasks from the Cell and to process status data from the Nodes.
In [CAM87] the general model of a controller within the CAM-Reference Model is defined. Already a short view on this controller shows the strong equivalence of the functionality of this controller with the functionality of the ROUTER. Therefore this general model of a controller is used to specify the behaviour of the ROUTER.

In chapter 4 also the place of the transport system within the CAM-Reference Model has been discussed.
The commands from the superior layer of the TCS are product type dependent, the commands from the TCS to the lower layer are product type independent. This means that the TCS can be placed on workstation level within the CAM-Reference Model. The workstation level in the control hierarchy of the production organisation is the lowest level which has to keep product information.
Fig.5.4 shows also the analogy with the control layers at the workstation environment in the CAM-Reference Model.
Because of this analogy the functionality of controllers at workstation level presented by v.d.Padt [PAD88] and the already existing LOTOS specification of a workstation controller presented by Biemans/Sjoerdsma [BIE86] have been used during the specification of the TCS-Architecture. Thereby the ROUTER specifications are made in such a way, that an implementation of an interface with the *Production Preparation* and the *Production Evaluation*

systems [HEH87] can be easily realised.
Fig.4.1 showed the position of the TCS between the Cell/Line layer, the Automation Module layer, the Production Preparation System and the Production Evaluation System.

5.1 Functional requirements

Both of the control layers in the TCS architecture have their own specific functional requirements.

5.1.1 functional requirements ROUTER

- The ROUTER receives commands from the Cell controller. A command from the Cell controller exists of a send request to send a specific number of products of a special type with a specific priority from one workstation to another workstation. To execute a command from the Cell the ROUTER splits this command into several route commands which are dedicated each to one specific Node. If necessary, the ROUTER also can give a command to all Nodes together. The ROUTER can send a number of route commands to a Node without waiting for the execution of each of these route commands by the Node. So because the ROUTER can contain several commands from the Cell, a Node Controller can contain several route commands.

- When a command is finished, an indication is given to the Cell

- The Cell controller can order to cancel a given command. The ROUTER cancels all the route commands in the Nodes which belong to this command, when execution has not started.

- The Cell can stop the total transport temporally until a new start command is received. This feature is desirable e.g. in case of tray congestion on tracks. After a stop command is received by the ROUTER, the ROUTER will stop all the Nodes. Only if a stop-acknowledge is received from all the Nodes, a stop-acknowledge will be returned to the Cell controller. Between a stop and a start command communication between the ROUTER, the Cell and the Nodes remains possible.

- The ROUTER can respond to a status request. The subject of the status request can be a product type or a Node identifier. The first one will result in an indication to the Cell about the current number and places of the products with the given type in the tranport system. The second one will result in an indication to the Cell of the phisical status of a specific Node.

- The ROUTER can receive a reset command from the Cell, to reset the complete command tables in all the Nodes and in the ROUTER.

- The ROUTER can send an alarm to the Cell, e.g. based on an alarm from a Node.

- The ROUTER can determine whether or not the order in a sequence of products is changed at delivery, or whether spontaneous (dis)appearence of products occur. In these cases an indication is given to the Cell.

5.1.2 functional requirements Node Controllers

- A Node contains a table of commands each dedicated to a special product type. The table might be empty.
- When a command is finished, the finishing will be indicated to the ROUTER and the command will be removed from the command table.
- Products at different tracks can apply simultaneously for movement at one Node, although a Node only can transport one product at the time. The order in dispatching the products is based on their priority.
- If no command is present in the set which fits to a just arrived product type, or the arrived product type is unknown, the Node will ask the ROUTER for a command. If the product type is known by the ROUTER, it will give a command to the Node after all, otherwise it will give a command for a default route.
- A Node will be able to receive a stop command, finish the movement, save the tables, and return a stop acknowledge. Then the Node waits for a start command before it continues executing route commands. Communication with the ROUTER remains possible.
- No difference exist in the delivery of a product to a track or to a Workstation.
  (Note: Special product data from the sending workstation can be presented to the receiving Workstation by a Node, the ROUTER, the Cell or by active labels. In case the TCS (i.e. a Node or the ROUTER) has to present the product data to the receiving workstation, the TCS can be seen as a fully transparent system because the TCS only receives and delivers the data and does not use or change it. Therefore this communication between two workstations through the TCS does not influence the synchronisation behaviour within the TCS and will be treated as being outside the scope of the current architecture. Also the handshake communication between the TCS and a Workstation to indicate the reception or arrival of product exceeds the scope of the present architecture.)
- If products with the same product type are indistinguishable, a command in the command table may contain a number which gives the quantity of products that have to be transported under this command.
- A cancel command can be received from the ROUTER to remove a specific command from the command table.
- A reset command can be received from the ROUTER to clear the complete command table, and to bring the AMs in a predefined state.

- A Node can respond to a status request, it can indicate information about its current fysical state to the ROUTER.
- In case of disfunction a Node can send an alarm signal to the ROUTER which indicates the reason of disfunction.

5.2 Informal specification TCS

Refering to Fig.4.1 first the messages between the ROUTER and the Cell/Line controller, the Production Preparation System, the Production Evaluation System and the Node Controllers will be specified. The messages are derived from the functional requirements as presented in section 5.1.

5.2.1 Messages

Messages between ROUTER and Cell/Line controller

1. *Send Request*
   The Send Request is sent by the Cell to request the transport of a specific number of products from one workstation to another.
   Parameters:
   - The Source Address is the address related to the workstation from which the products have to be transported.
   - The Destination Address represents the address of the workstation which has to receive the products.
   - The Product Type indicates the type of the products that have to be transported.
   - The Priority of the product which can be used e.g. when more products arrive simultaneously at one Node.
   - Quantity gives the number of products that have to be transported with this Send Request.
2. *Transport Completed*
   When a Send Request has been executed, so the last product has arrived at the receiving Workstation, the ROUTER will send a Transport Completed to the Cell.
   Parameter:
   - The Transport Completed message contains the type of the product to indicate which Send Request has been executed.
3. *Reset Command*
   The Reset Command is used to disrupt the execution of a all Send Requests in the transport system in case of emergency. Disruption means that no new products will be accepted from sending Workstations, the remaining products in the transport system

will be delivered to the required Workstations, all Node Tables will be cleared, the send requests will be deleted and the TCS will remain in a predefined state.
No parameters.

4. *Reset Acknowledge*
When the disruption after receiving a reset command has been finished, the ROUTER will send a Reset Acknowledge to the Cell.
No parameters.

5. *Cancel Command*
The Cancel Command is used to disrupt the execution of a specific Send Request. No products will be accepted from the sending Workstation, the remaining products in the transport system will be delivered to the receiving Workstation, the Route Commands belonging to the Send Request will be removed from the Node Tables concerned, and the Send Request will be deleted. Parameters:

   - The Cancel Command contains the type of the product which Send Request should be disrupted.

(Note: the Reset Command can be implemented as a special application of the Cancel Command)

6. *Cancel Acknowledge*
When the disruption after receiving a Cancel Command has been finished, the ROUTER will send a Cancel Acknowledge to the Cell.
Parameters:

   - The type of the product which Send Request has been disrupted.

7. *Stop Command*
This message commands the transport system to stop transportation and hold the system in the next possible well defined state. Transportation can continue after receiving a Start Command.
No parameters.

8. *Stop Acknowledge*
When a Stop Command has been executed by the ROUTER, i.e. the ROUTER has received a Stop Acknowledge from all Nodes, the ROUTER will return a Stop Acknowledge to the Cell. During the time that the ROUTER is waiting for all acknowledges, other communication between ROUTER and Nodes remains possible: the ROUTER still has to be able to receive e.g. Movement Completed messages.
No parameters.

9. *Start Command*
The Start Command (re-)starts the transport system. The system should be in a defined state.
No parameters.

10. *Start Acknowledge*
    When the ROUTER, after receiving a Start Command from the Cell, (re-)started all Nodes successfully, it will return a Start Acknowledge to the Cell.
    No parameters.

11. *Reject Indication*
    If the system can not execute a Send Request it will send a Reject Indication to the Cell.
    Parameters:

    - The Reject Indication contains a parameter Reason which gives the reason of rejection (e.g. memory overflow, no capability, no route available, etc.)

12. *Product Status*
    The ROUTER can inform the Cell (solicited or unsolicited) about the position of products with a specific type in the system: the amount (optional: sequence) of products in each track, the amount (sequence) of products that already have arrived at the receiving workstation, an indication wether the transport has been completed, etc. This information is sent to the Cell by the message Product Status.
    Parameters:

    - A parameter Product Type representing the type of the product.
    - A parameter Type of Information which indicates the sort of the information. This can be used within the Product Status Request to indicate what information is required.
    - A parameter Product Information which contains the information.

13. *Product Status Request*
    The Cell can send a Product Status Request to the ROUTER to ask for information about a product type (Product Status).
    Parameters:

    - The parameter Product Type.
    - The parameter Type of Information which indicates the sort of information which is required.

14. *Facility Status*
    The ROUTER can inform the Cell about service qualities, capacities, capabilities, extra features, small disfunctions etc. of the transport facility. This information is put in the message Facility Status.
    Parameters:

    - The parameter Type of Information which indicates the sort of the information. This can be used within the Facility Status Request to indicate what information is required.
    - The parameter Facility Information which contains the required information.

15. *Facility Status Request*
The Cell can send a Facility Status Request to receive some information about the transport facility (Facility Status).
Parameters:

- The parameter Type of Information which indicates the sort of information which is required.

16. *Alarm*
In case of a disfunction somewhere in the transport system, the ROUTER can send an Alarm to the Cell.
Parameter:

- The Alarm message contains information about the Origin of the disfunction.

MESSAGES	PARAMETERS
Send Request	Source Adress Destination adress Product Type Priority Quantity
Transport Completed	Product Type
Reset Command	
Reset acknowledge	
Cancel Command	Product Type
Cancel Acknowledge	Product type
Stop Command	
Stop Acknowledge	
Start Command	
Start Acknowledge	
Reject Indication	Reason
Product Status	Product Type Type of Information Product Information
Product Status Request	Product Type Type of Information
Facility Status	Type of Information Facility Information
Facility Status Request	Type of Information
Alarm	Origin

Fig.5.5: Messages between ROUTER and Cell/Line Controller.

Messages between ROUTER and Node Controllers

1. *Route Command*
   The Route Command is a command to a specific Node to transport a certain number of products of a special type and a certain priority to a specific track.
   Parameters:
   - The Node Identity of the Node to which the Route Command is sent.
   - The Product Type of the products which have to be moved under this command.
   - The Quantity of products which have to be moved
   - The number of the track from which the products are received.
   - The number of the track to which the products have to be moved.
2. *Movement Completed*
   When a Node has executed a Route Command, so the movement has finished and the product is delivered to a track, the Node will send a Movement Completed to the Router.
   Parameter:
   - The Product Type of the products which Route Command has been executed.
3. *Reject Indication*
   If the Node can not execute a Route Command (e.g. if the command contains a number of a track which is not connected to the Node), it will send a Reject Indication to the ROUTER.
   Parameter:
   - The Reject Indication contains a parameter Reason which indicates the reason of rejection.
4. *Reset Command*
   The Reset Command is used to clear the complete table of Route Commands in a Node. It is possible to send the reset to all Nodes together.
   No parameters.
5. *Reset Acknowledge*
   After receiving a Reset Command from the ROUTER, the Node Controller will finish the movement (if it is moving a product), clear the Node Route Command table and return a Reset Acknowledge to the ROUTER.
   No parameters.
6. *Cancel Command*
   This command can be used to remove a specific Route Command from the Route Command table. If a Route Command is in execution, the movement will be finished first.
   Parameter:

- The parameter Node Information which contains the required information.

14. *Node Status Request*
The ROUTER can send a Node Status Request to ask a Node about its current state.
Parameter:

    - The Node Identity of the Node concerned.
    - The parameter Type of Information which indicates the sort of the information which is required.

15. *Alarm*
The ROUTER can receive an Alarm signal from a Node in case of disfunction.
Parameters:

    - The Alarm contains the Origin of the Alarm

MESSAGES	PARAMETERS
Route Command	Node Identity Product Type Quantity Source Gate Destination Gate
Movement Completed	Product Type
Reject Indication	Reason
Reset Command	
Reset Acknowledge	
Cancel Command	Product Type
Cancel Acknowledge	Product Type
Stop Command	
Stop Acknowledge	
Start Command	
Start Acknowledge	
Route Command Request	Product Type
Node Status	Node Identity Type of Information Node Information
Node Status Request	Node Identity Type of Information
Alarm	Origin

Fig.5.5: Messages between ROUTER and Node Controllers.

Messages between ROUTER and Production Preparation System

1. *Recipe*
   A Recipe can be send (solicited or unsolicited) from the PPS to the ROUTER to insert a new Recipe or to change an existing Recipe. A Recipe consists of a set of Route Commands, each intended for a specific Node.
   No parameters.

2. *Recipe Request*
   This message is send from the ROUTER to the Production Preparation System (PPS) to ask for a Recipe which belongs to a specific Send Request. Parameter:

   - The Send Request for which the Recipe is required.

MESSAGES	PARAMETERS
Recipe	
Recipe Request	Send Request

Fig.5.6: Messages between ROUTER and PPS.

Messages between ROUTER and Production Evaluation System

1. *Product Status*
   To update the PES the ROUTER can inform the it (solicited or unsolicited) about the position of specific products in the system: the amount (optional: sequence) of products in each track, the amount (sequence) of products that already haved arrived at the receiving workstation, an indication wether the transport has been completed, etc. This information can be sent to the Cell by the message Product Status.
   Parameters:

   - The Product Type of the product concerned.
   - A parameter Type of Information which indicates the sort of information (is used within the status request).
   - A parameter Product Information which contains the information.

2. *Product Status Request*
   The PES can send a Product Status Request to the ROUTER to ask for information about a product type (Product Status).
   Parameter:

- The Product Type of the products concerned.
- The parameter Type of Information which indicates the sort of information which is required.

3. *Facility Status*
   The ROUTER can inform (solicited or unsolicited) the PES about service qualities, capacities, capabilities, extra features, small disfunctions etc. of the transport facility. This information is put in the message Facility Status.
   Parameters:

   - The parameter Type of information which indicates the sort of information (is used by the Facility Status Request).
   - The parameter Facility Information which contains the information.

4. *Facility Status Request*
   The PES can send a Facility Status Request to receive some information about the transport facility (Facility Status).
   Parameter:

   - The parameter Type of Information which indicates the sort of information which is required.

MESSAGES	PARAMETERS
Product Status	Product Type Type of Information Product Information
Product Status Request	Product Type Type of Information
Facility Status	Type of Information Facility Information
Facility Status Request	Type of Information

Fig.5.7: Messages between ROUTER and PES.

5.2.2 Temporal ordering ROUTER

In analogy with the general model of a controller defined in [CAM87] the process ROUTER is specified as the parallel composition of three subprocesses: the ROUTE_DECOMPOSER (the *H-Module*), the STATUS_COLLECTOR (the *G-Module*) and the MONITOR (the *M-Module*).

- The type of the product which Route Command has to be removed.

7. *Cancel Acknowledge*
   When a specific Route Command is removed from the Route Command table, the Node returns a Cancel Acknowledge to the ROUTER.
   Parameter:

   - The type of the product which Route Command has been removed.

8. *Stop Command*
   This message commands a Node to stop moving products. If the Node is just moving a product, the product will be delivered to the required track first before the Node will stop. No product will be accepted by the Node until a Start Command is received. Table contents will be saved.
   No parameters.

9. *Stop Acknowledge*
   When a Node has finished all necessary activities to stop properly (finish the movement if moving), it returns a Stop Acknowledge to the ROUTER.
   No Parameters.

10. *Start Command*
    The Start Command (re-)starts the movement of a Node.
    No parameters.

11. *Start Acknowledge*
    After receiving a Start Command from the ROUTER and finishing the start up properly, the Node will return a Start Acknowledge to the ROUTER.
    No parameters.

12. *Route Command Request*
    If a Node receives a product which type is unavailable or which type is known but no Route command for this type is available in the command table, the Node will send a Route Command Request to the ROUTER to know what to do with the just arrived product.
    Parameter:

    - The Product Type of the arrived product. If no Product Type is available, the value of the parameter will be "ptype_not_available".

13. *Node Status*
    The Node can send a Node Status (solicited or unsolicited) to inform the ROUTER about its current state.
    Parameters:

    - The Node Identity of the Node concerned.
    - The parameter Type of Information which indicates the sort of the information. This can be used by the Node Status Request to indicate what information about which Node is required.

See Fig.5.8. This figure is a combination of fig.4.1 and fig.5.1b. The process RECIPES represents the PPS and the process EVALUATION_TABLES represents the PES.

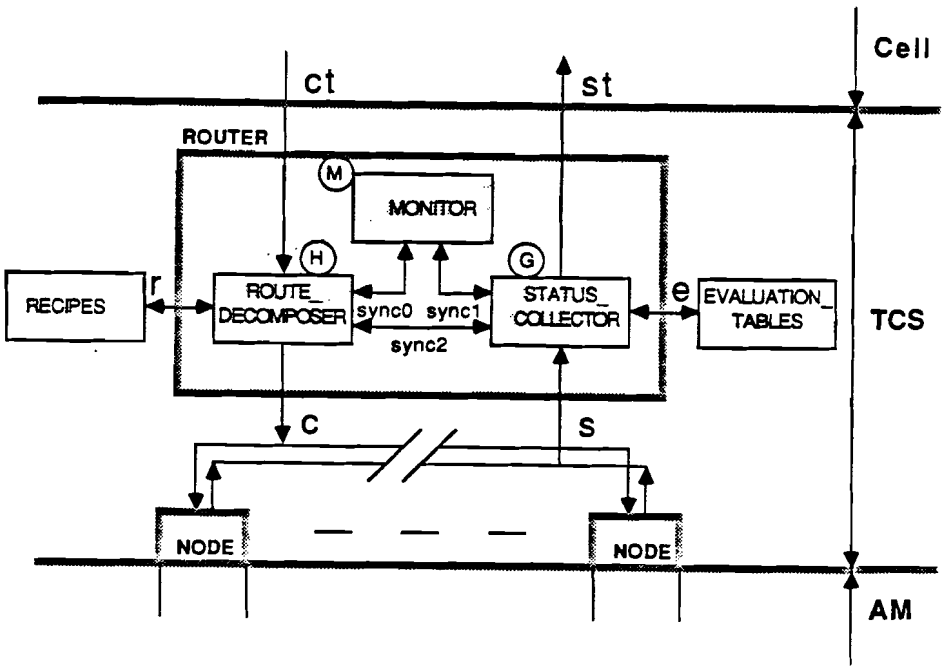


Fig.5.8: The ROUTER specified as the parallel composition of the G-, M- and H-Modules as presented in [CAM87]

The functions of the three processes in the ROUTER are:

1. The ROUTE_DECOMPOSER.
   The main function of the ROUTE_DECOMPOSER is to receive commands from the Cell controller, to indicate to the Cell wether or not commands can be executed, and to decompose the commands in route commands for the Nodes, based on information from the Production Preparation System (recipes) and from the MONITOR (world model).

2. The STATUS_COLLECTOR.
   The main function of the STATUS_COLLECTOR is to collect and to process status from the Nodes in order to update the MONITOR and the Production Evaluation System (evaluation tables). It can indicate to the Cell the current state of the transport facility, or it can inform the Cell about the transport or the delivery of the products.

3. The MONITOR.
   The main function of the MONITOR is to comprise application dependent facility primitives such as service qualities, capabilities and capacities, as well as information about the current state of the transport facility (extra or missing features, new routes, small disfunctions, etc). Thereby the MONITOR maintains information about where the products are in the transport system.
   The facility and product informations can be used by the ROUTE_DECOMPOSER to

choose for a specific route.
The facility and product informations are updated by the STATUS_COLLECTOR.

ROUTE_DECOMPOSER

The ROUTE_DECOMPOSER behaves as three processes that are concurrently operational. These processes are RECEIVE_COMMANDS, EXECUTE_COMMANDS and RECIPE_MANAGER (see Fig.5.9):

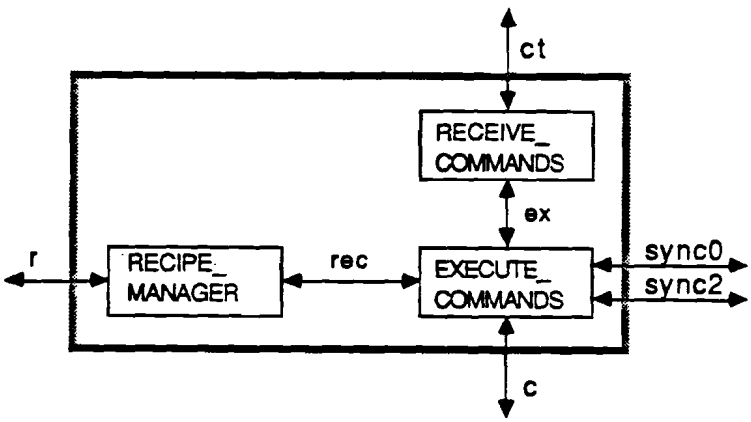


Fig.5.9: ROUTE_DECOMPOSER

1. RECEIVE_COMMANDS.
   This process receives either a Send Request, or a Reset Command, or a Cancel Command, or a Stop/Start Command from the Cell controller, or it receives a Reject Indication from EXECUTE_COMMANDS e.g. in case no route is available. The commands from the Cell are passed to EXECUTE_COMMANDS and the process will return in its original state (except in the case of a Stop Command: first a Start, Reset or Cancel Command should be received before RECEIVE_COMMANDS can receive again all possible commands). The Reject Indication from EXECUTE_COMMANDS is passed to the Cell controller, and again the process returns in its original state.

2. EXECUTE_COMMANDS.
   This process receives a command from RECEIVE_COMMANDS and executes it. The execution of the commands is specified as the parallel composition of five processes which are each responsible for the execution of a specific command:

   - SET_TABLES
     After a Send Request is received from RECEIVE_COMMANDS and a Facility Status is received from MONITOR the process SET_TABLES will ask for a Recipe from the RECIPE_MANAGER. When it is received the Recipe is used to send specific Route Commands to the Nodes on the chosen route. If no Recipe exists, e.g. because no route is available, a Reject Indication is sent to RECEIVE_COMMANDS.

- RECEIVE_REJECTS
  If a Node does not accept the Route Command (e.g. the command contains gates of the Node which are not connected to a track), the process SET_TABLES will receive a Reject Indication from the Node and will pass it to the RECIPE_MANAGER.
- RESET_TABLES
  After a Reset Command is received from RECEIVE_COMMANDS, the Reset Command is sent to all Nodes in the transport system and to the MONITOR.
- CANCEL_COMMANDS
  If a Cancel Command is received from either RECEIVE_COMMANDS or STATUS_COLLECTOR the command is send to the MONITOR. Then a Cancel Command is sent to all Nodes containing the type of the product which Route Command should be removed.
- STOP_START_COMMANDS
  If a Stop/Start Command is received from RECEIVE_COMMANDS the Stop/Start Command is sent to all Nodes in the system and to the MONITOR.
- PRESENT_DEFAULTS
  It could occur that at a Node a product arrives which type is unavailable (e.g. bar code is unreadable) or which type is available but no Route Command for this type exists in the Route Commands Table. In both cases the process PRESENT_DEFAULTS will receive a Route Command Request from a Node. After receiving this request PRESENT_DEFAULTS will ask the RECIPE_MANAGER for a Route Command and will send this Route Command to the Node.
  (Note: if the product type is unavailable it could be possible to estimate the type, based on recent information about products on the track from which the product has been received. This feature has not been implemented in the LOTOS specification.)

3. RECIPE_MANAGER.
   This process contains a set of Recipes. Using a product type this set can deliver a Recipe, and using a product type as well as a node identity this set can deliver a Route Command.
   The process:
   either offers the process EXECUTE_COMMANDS a Recipe that is required by this process,
   or receives a Route Command Request and offers a required Route Command to EXECUTE_COMMANDS,
   or receives a Reject Indication from EXECUTE_COMMANDS and passes it to the PPS.
   or asks the PPS for a Recipe by sending a Recipe Update Request,
   or receives a Recipe Update from the PPS.

STATUS_COLLECTOR

The process STATUS_COLLECTOR receives sequentially status indications or status requests. The indications and requests are executed parallel. The process is specified as the parallel composition of three concurrent processes RECEIVE_STATUS, EXECUTE_STATUS and EVALUATION_TABLE_MANAGER (see Fig.5.10):

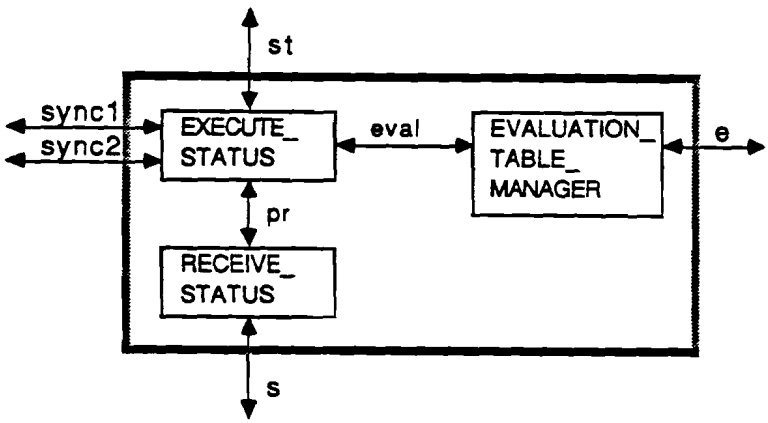


Fig.5.10: STATUS_COLLECTOR

1. RECEIVE_STATUS.
   This process:
   either receives a Movement Completed from a Node and passes this message to EXECUTE_STATUS,
   or receives an Alarm from a Node and passes it to EXECUTE_STATUS,
   or receives a Stop, Start, Reset or Cancel Acknowledge from a Node and passes it to EXECUTE_STATUS,
   or receives a Node Status Request from EXECUTE_STATUS and passes it to the Node,
   or receives a Node Status from a Node and passes it to EXECUTE_STATUS.

2. EXECUTE_STATUS.
   This process executes a status indication or status request. The process is specified as the parallel composition of four processes:

   - UPDATE_PRODUCT_STATUS
     This process receives a Movement Completed from RECEIVE_STATUS and passes it to MONITOR to update the Account (which is a part of Product Information) of the product type which just was moved by the Node. Then UPDATE_ACCOUNTS will receive the updated Account from MONITOR. If all the products under one Send Request have reached the destination, a Transport Completed is received from MONITOR and is passed to the Cell and to the EVALUATION_TABLE_MANAGER, and a Cancel Command is send to EXECUTE_COMMANDS. If not all the products are yet delivered only the Account is passed to the EVALUATION_TABLE_MANAGER.

- UPDATE_FACILITY_STATUS
  This process receives a Facility Update Request from the EVALUATION_ TABLE_MANAGER and passes it to MONITOR.
- EXECUTE_ACKNOWLEDGES
  This process receives a Stop, Start, Reset or Cancel Acknowledge from RECEIVE_STATUS. The process waits until it has received this Acknowledge from all Nodes. Then it passes a Stop, Start, Reset or Cancel Acknowledge to the Cell controller and the EVALUATION_TABLE_MANAGER.
- EXECUTE_ALARM
  This process receives an Alarm from RECEIVE_STATUS and passes it to EXECUTE_COMMANDS, MONITOR, the Cell and the EVALUATION_TABLE_MANAGER
- GIVE_STATUS
  This process gives respons to a status request, i.e. a Product Status Request, a Facility Status Request or a Node Status Request.
  Either the request comes from the Cell. Then GIVE_STATUS asks MONITOR for a Product Status or a Facility Status, or asks the Node for a Node Status, depending on what was required. The Product Status, Facility Status or Node Status is sent to the Cell.
  Or the request comes from the EVALUATION_ TABLE_MANAGER, so the Product Status, Facility Status or Node Status will be sent to the EVALUATION_TABLE_MANAGER.

3. EVALUATION_TABLE_MANAGER.
   This process provides the interface with the Production Evaluation System (PES). The PES is represented as a set of evaluation tables each dedicated to one of the various evaluation systems within the PES. (In this specification of the interface only a general approach is made. As an extension the concept of "Formulas" can be used as introduced by Biemans/Sjoerdsma in [BIE86]. A Formula can prescribe which status information should be sent to which system.)
   The EVALUATION_TABLE_MANAGER
   either receives an Product Status, a Facility Status or a Node Status from EXECUTE_STATUS and updates the evaluation tables,
   or receives from the PES a status request and asks EXECUTE_STATUS for the required status.

MONITOR

In the MONITOR three processes operate concurrently. These processes are UPDATE_-SEND_COMMANDS, UPDATE_STATUS and WORLD (See Fig.5.11).

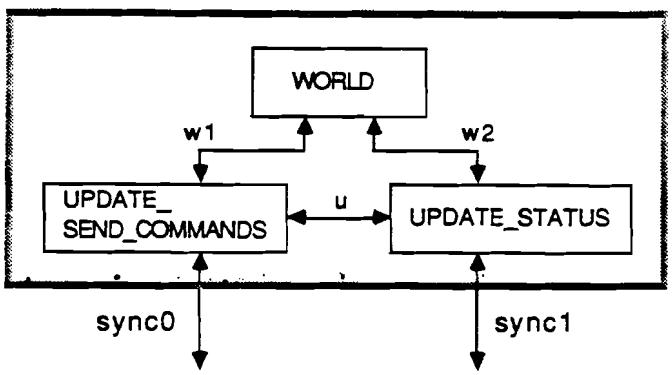


Fig.5.11: MONITOR

1. UPDATE_SEND_COMMANDS
   This process maintains a set of Send Requests which are active in the transport system. The process
   either receives a Send Request from the ROUTE_DECOMPOSER, then asks the WORLD for the Facility Status, passes this Facility Status to the ROUTE_DECOMPOSER, and finally adds the Send Request to the set of Send Requests,
   or receives a Cancel Command from the ROUTE_ DECOMPOSER, passes this command to UPDATE_STATUS, and removes the concerned Send Request from the set of Send Requests,
   or receives a Reset Command from the ROUTE_ DECOMPOSER, passes this command to UPDATE_STATUS, and clears the complete set of Send Requests,
   or receives a Stop Command and waits for a Start Command to continue in the same state as when it was stopped,
   or receives a Transport Completed from UPDATE_STATUS and removes the Send Request in Transport Completed from the set of Send Requests.

2. UPDATE_STATUS
   This process maintains the set of Accounts which contains information about the place of products in the transport system.
   The process
   either receives a Send Request from UPDATE_SEND_ COMMANDS to initiate an Account dedicated to the product type in this Send Request,
   or receives a Movement Completed from the STATUS_COLLECTOR, updates the set of Accounts, returns to the STATUS_COLLECTOR the Account of the product type which just was moved by the Node, and. If all the products under one Send Request have reached the destination, the Account is removed from the Account set and a Transport Completed is sent to the UPDATE_SEND_ COMMANDS and the STATUS_COLLECTOR,

or receives a Product Status Request or a Facility Status Request from the STATUS_COLLECTOR, gets the required Account from the set or a Facility Status from the WORLD, and send it to the STATUS_COLLECTOR,
or receives a Facility Update Request from the STATUS_COLLECTOR and passes it to WORLD,
or receives an Alarm from the STATUS_COLLECTOR and passes it to WORLD,
or receives a Cancel Command from UPDATE_ SEND_COMMANDS, and removes the concerned Account from the Account set,
or receives a Reset Command from UPDATE_ SEND_COMMANDS, and clears the complete Account set.

3. WORLD
This process comprises the transport facility application dependent primitives. The process:
either provides the UPDATE_COMMANDS with the required Facility Status,
or updates the Facility Status based on a received Alarm or a Facility Update Request.

5.2.3 Temporal ordering Node Controller

Figure 5.12 shows the Node Controller represented as a process with six communication gates: four gates for the exchange of products with four tracks connected to the Node (wa,wb,wc and wd), and two gates for the exchange of commands (c) and status (s) with the ROUTER.

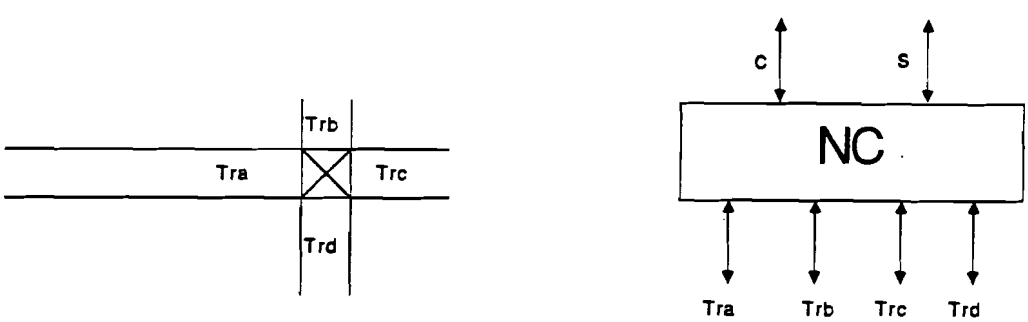


Fig.5.12: The Node Controller represented as a process with six communication gates.

For an implementation of the Node Controller the concept of five parallel processes as presented in the simplified ITS specifications (see section 3.1) appeared to be suitable and very useful (see fig.5.13).

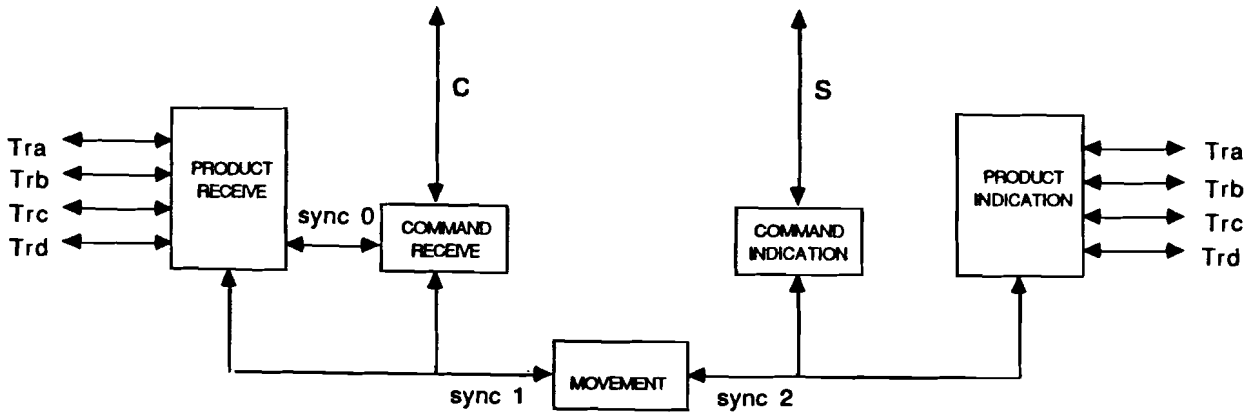


Fig5.13: Implementation of the Node Controller as the parallel composition of five concurrent processes.

These processes respectively manage the reception of a product (the process PRODUCT_-RECEIVE), the reception of commands from the ROUTER (COMMAND_RECEIVE), the movement of the received product to the required track (the process MOVEMENT), the indication of status to the ROUTER (STATUS_INDICATION) and the delivery of the product to the required track or Workstation (PRODUCT_INDICATION).
The temporal ordering of these processes are:

PRODUCT_RECEIVE

This process handles the reception of a product from a track. The process receives a product from a track, sends the type of the product to the process COMMAND_RECEIVE, and waits until the process MOVEMENT has taken over the control of the product, before COMMAND_RECEIVE can receive another product.

COMMAND_RECEIVE

The process COMMAND_RECEIVE manages the interface with the ROUTER to receive Route Commands. It contains a set of received Route Commands. It contains the identity of the Node concerned to investigate whether a command is intended for it.
The process:
Either receives a Route Command from the ROUTER. A check is made whether the command is executable. If the command is executable, it is inserted in the set of Route Commands, if the command is not executable, a Reject Indication is returned the the ROUTER containing the reason of rejection.
Or receives a Reset Command from the ROUTER, passes it to MOVEMENT and clears the Route Commands set.
Or receives a Cancel Command from the ROUTER. When a Route Command for the product

type presented by the Cancel Command is present in the Route Commands set, the the Cancel Command is passed to MOVEMENT and the Route Command concerned is removed from the Route Commands set. If no Route Command for the product type presented by the Cancel Command, only the Cancel Command is passed to MOVEMENT.
Or a Stop or Start Command is received. These are passed to MOVEMENT.
Or a product type is received from PRODUCT_RECEIVE. Then the Route Command belonging to this product type is taken from the commands set. If no Route Command exists for this product type, a Route Command Request is sent to the ROUTER. If it exists, the contents of the Route Command are passed to MOVEMENT.

MOVEMENT

MOVEMENT is the process within the Node Controller which actually deals with the transport of a product from track to track. It controls the sensors and actuators in the AM-layer. This control exeeds the scope of the present architecture and is therefore represented as an unobservable (internal) action.
The process:
Either receives a product and the number of the output gate to which the product has to be moved. The product will be moved and the output gate and the type of the product are passed to STATUS_INDICATION and PRODUCT_INDICATION.
Or receives a Reset Command, a Cancel Command or a Stop/Start Command from COMMAND_INDICATION and passes it to STATUS_INDICATION and PRODUCT_INDICATION.
Because MOVEMENT can not receive Reset, Cancel, Stop or Start Commands parallel with the actual movement of a product, it is assurred that always the movement will be finished before one of these commands can be executed. In case a Stop Command is received, no movement can take place before a Start Command will be received.

STATUS_INDICATION

This process manages the indication of status to the ROUTER. It contains the identity of the Node concerned. This identity is used to indicate the ROUTER from which Node it receives a status.
The process:
Either receives a message from MOVEMENT that a product with a specific type has been transported to a specific track. The process will wait until the delivery of the product to the receiving track is confirmed by PRODUCT_INDICATION, before it sends a Movement Completed message to the ROUTER.
Or recieves a Reset, Cancel, Stop or Start Command from MOVEMENT and will send an acknowledge to the ROUTER.
Or receives a Node Status Request from the ROUTER. The Node Status will be investigated and send to the ROUTER.

PRODUCT_INDICATION

The process PRODUCT_INDICATION deals with the delivery of the product to the receiving track or Workstation.
The process a message from MOVEMENT, containing the output gate and a product type, indicates the receiving track that the product has been arrived, and sends a confirmation of the delivery of the product to STATUS_INDICATION.

5.3 Formal specification ROUTER and Node Controller

The LOTOS specifications of the ROUTER and the Node Controller can be found in Appendix B of this document.
Until the moment this document was finished, the specifications have not been verified. Only the syntax and the static semantic has been checked and erorless installed.
Before the specifications can be verified, some application dependent processes have to be specified. Their specifications depent on e.g. how many Nodes, which connections, which routes are present in the application. The specification of the Node Controller is based on a Node which can receive products from two gates (wa and wb) and can deliver them to one of the other gates (wc and wd). The specification is quite easy to extend to a general application which can receive products from all gates, and move them to one of these gates (i.e. the Node can send the product to the same track as from which it received the product).

Chapter 6

Conclusions and Recommendations

6.1 Conclusions

- As a result of the comparison between four FDTs, LOTOS was established to be the most useful for CAM-Architectures. It is important to realise that a lot of developments are going on to bring other FDTs in the same position as LOTOS at the moment. So a comparison after a few years from now could result in an other conclusion.

- Service protocols, messages between processes, the composition of components, and full integration of system modules are typical concepts within a CAM environment. After an intensive research on the usefulness of FDTs for the specification of CAM-Architectures, it became obvious that especially these concepts make CAM modules very suitable to be described by a FDT.
  Uniformity in description techniques realise an uniformity and normality in products, which will improve the exchangability.

- The usage of an FDT to describe system parts improves the possibility to verify by mathematical rules whether system parts can be connected correctly. This can be stated undoubtedly after specifying the new TS-Architecture in LOTOS. Based on a general architecture (the transport of a product from one workstation to another workstation), the system was divided in several subsystems (Router, Node Controllers, Tracks) which were specified and verified with the aid of the LOTOS Toolset. After composing the subsystems the overall system could be verified without too much effort. The next step can be to describe more CAM systems in LOTOS to verify their composed behaviour (e.g. the composition of the TCS, some Workstations and a Cell/Line controller).

- The use of the LOTOS Toolset showed that already in a very early stage of analysis, fundamental constraints for the design can be checked on their consistency. One can start with a very general specification of a system, and then use the Toolset to verify whether or not some constraints are contradictory. Thereafter one can specify more details of the system, and again use the Toolset for verification. This can be repeated until the desired depth in details is reached in the system specification.

- It appeared that for verifying the communication behaviour of several subsystems most of the time was used for verifying the synchronisation. Verifying the value passing was only a small step after the synchronisation was proved to be correct.

- The Blonk/Biemans Route Service specifications on level 0, based on a fully distributed control architecture, appeared to be correct after verification. The several systemparts communicated free of deadlock. However, Beukebooms reconsiderations asked for a general architecture which conformed more with already existing transport controllers made within Philips by I&E, CE and CFT: a partially distributed control architecture.

- The concept of partially distributed control of the transport system was easy to implement in LOTOS. Actually, because LOTOS was made to describe communication of concurrent processes, especially a transport system is a suitable object to describe in LOTOS and to show the enhanced features of this FDT, due to the fact that the system consists of many Nodes which run in parallel.

- The analogy of the TCS with the Workstation controller supported to implement in LOTOS. Both because of the analogy with the decomposition of tasks within a workstation controller, and because a LOTOS version of a workstation controller already existed.

6.2 Recommendations

- One important feature of the formalism in LOTOS has not been used: the possibility to prove the equivalence of systems. With the LOTOS Toolset only behaviour trees can be generated, not behaviour expressions. These expressions are necessary to use equivalence rules. Of course, in case of small trees it is possible to generate an expression from a tree by a simple look at the tree. But normally the trees are very large, so some software tools should be developed to realise a behaviour expression from a behaviour tree.

- The LOTOS Toolset can not generate a behaviour tree by itself. The tree is built up by an interaction between the user en the Toolset. The user will choose a possible event and the Toolset enters the path in the tree which belongs to that event. In this way it is possible to explore all the paths in the tree. However, in large trees a lot of paths exist so it becomes a rather time consuming business to generate them. Therefore a software tool should be developed which can generate all possible sequences of events to test the system behaviour. Then, with these test sequences, all paths in the behaviour tree can be found automatically.

- Because the main effort during the verification of the sytem specifications was used for the verification of the synchronisation, it could be an interesting exercise to represent the process description part of the TCS in CCS. Then the verification will be limited to the synchronisation behaviour and with the aid of the rather compact representation in CCS all the fundaments of the formalisms which form the basis of LOTOS can be used easily to analyse and check the composition of subsystems.

- Next to the FDTs a lot of "Design Techniques" exist (Yourdon, Mascot, SADT, etc.). Many of these design techniques make use of very clear and surveyable charts and graphics, but are not based on an algebraic formalism.
  Although LOTOS is in the first place a description technique, its algebraic formalism can be used to verify steps in a design process.
  So a combination of LOTOS and one of these design techniques could result in a design tool which uses charts and graphics to realise a LOTOS specification, based on a formal verification in each design step.

- As mentioned, during the description of the new TS-Architecture the behaviour of the Router showed a strong analogy with the behaviour of a Workstation Controller. Actually, the LOTOS specifications of the Router were based on the LOTOS specifications of a Workstation Controller. So if such a strong connection exists between the behaviour of the Router and the behaviour of a Workstation Controller, it could be very interesting to investigate whether an application independent LOTOS specification can be made of controllers at workstation level by using e.g. parameters or general commands. Then a specific application (a Router or a Workstation Controller) can be realised by attaching values to the parameters or replacing the general commands by application dependent commands.
  This is also found in practice: it is investigated to what extend the already existing physical implementation of a Workstation Controller can be used for the physical implementation of the Router.

References

[BES88] Bessems, P.
"CCS, LOTOS, Estelle en SDL als specificatie technieken voor CAM-Architecturen"
Faculty of Electrical Engineering, Group EB
Eindhoven University of Technology, 1988

[BEU88] Beukeboom, R. and W. Offringa, R. Vandebergh
"On the Transport Controller System, general user requirements"
CFT-Report 23/88, 1988

[BIE86] Biemans, F. and Sj. Sjoerdsma
"Description and motivation of a workstation controller architecture"
CFT-Report 55/86, 1986

[BLO87] Blonk, P. and F. Biemans
"An architecture of internal transport systems"
CFT-Report 09/87 and 10/87, 1987

[BRI85] Brinksma, E.
"A tutorial on LOTOS"
Proc. of the IFIP WG 6.1 5th. Int. Workshop on
Protocol Specification, Testing and Verification.
North-Holland, 1985

[CAM87] Philips and DEC
"CAM Reference Model of production systems, vs. 1.0
CFT-Report, 1987

[EHR83] Ehrig, H. and W. Fey, H. Hansen
"ACT-ONE: An algebraic specification language with two levels of semantics"
Technical University of Berlin, 1983

[EST87] ISO/TC 97
"Information processing systems - Open Systems Interconnection - Estelle (Formal Description Technique Based on an Extended State Transition Model)"
Draft International Standard ISO/DIS 9074
ISO, 1987

[HEH87] Hehl, C. and H. Lammers
"The realisation of Flexible Cell/Line Controllers in perspective"
CFT/CAM-Centre, 1987

[KOO88] Koomen, C.J.
"System Technology I"
Lecture Notes
Eindhoven University of Technology, 1988

[LIN85] Linn Jr., R.J.
"The features and facilities of Estelle"
Proc. of the IFIP'85 Workshop on Protocol Specification, Testing and Verification
North-Holland, 1985

[LOT87] ISO/TC 97
"Information processing systems - Open systems interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour"
Draft International Standard ISO/DIS 8807
ISO, 1987

[MIL80] Milner, R.
"A Calculus of Communicating Systems"
Lecture Notes in Computer Science
Springer-Verlag, Berlin, 1980

[PAD88] Padt, v.d. A.
"Inventory of User Requirements for Controllers at the Workstation Level"
CFT-Report 22/88, 1988

[ROC82] Rockstrom, A. and R. Saracco
"SDL - CCITT Specification and Description Language"
IEEE Trans. on Communications,
Vol. COM-30, No.6, June 1982

[SCO86] Scollo, G. and C.A. Vissers, A. di Stefano
"LOTOS in practice"
IFIP'86
North Holland, 1986

[SDL84] CCITT
"Functional Specification and Description Language"
Recommendations Z.101-Z.104, Red Book, VI.10
8th. Plenary Assembly,
Malaga-Torremolinos, Spain, 1984

Appendix A

LOTOS specification TS-Architecture version 1: ITS (simplified)

```
(*
ROUTE SERVICE SPECIFICATION
Level 0
Data Type Definitions
*)

specification ROUTE_SERVICE[ct,wa,wb,st]:noexit

library FBoolean, Boolean, SetElement, Set,
        NaturalNumber endlib


  (* QUEUE OF ELEMENTS *)
type QUEUE is SetElement, FBoolean
sorts Queue
opns <>     : -> Queue
     Add    : Element, Queue -> Queue
     First  : Queue -> Element
     Rest   : Queue -> Queue
     Link   : Queue, Queue -> Queue
     _eq _, _ne _: Queue, Queue -> FBool
     IsEmpty: Queue -> FBool

eqns forall e,e1,e2:Element, q,q1,q2:Queue
     ofsort Queue
     Rest(<>) = <>;
     (q ne <>) = true => Rest(Add(e,q)) = Add(e,Rest(q));
     Rest(Add(e,<>)) = <>;
     Link(<>,q) = q;
```

```
      Link(q,<>) = q;
      Link(q1,q2) = Link(Add(First(q2),q1),Rest(q2));

      ofsort Element
      First(Add(e,<>)) = e;
      (q ne <>) = true => First(Add(e,q)) = First(q);

      ofsort FBool
      <> eq <> = true;
      Add(e1,q1) eq Add(e2,q2) = (e1 eq e2) and (q1 eq q2);
      q1 eq q2 = q2 eq q1;
      Add(e,q) eq <> = false;
      q1 ne q2 = not (q1 eq q2);
      IsEmpty(<>) = true;
      IsEmpty(Add(e,<>)) = false;
endtype

  (* END OF QUEUE OF ELEMENTS *)

  (* END OF ELEMENT ACTUALIZED DATA TYPES *)

type PRODUCT is Boolean
sorts ProductSort
opns type1,type2,type3 : -> ProductSort
     _eq _, _ne _, _lt _: ProductSort, ProductSort -> Bool
eqns forall s,t: ProductSort
     ofsort Bool
     type1 eq type1 = true;
     type2 eq type2 = true;
     type3 eq type3 = true;
     type1 eq type2 = false;
     type1 eq type3 = false;
     type2 eq type3 = false;
     s eq t = t eq s;
     s ne t = not(s eq t);
endtype (* PRODUCT *)

type R_ADDRESS is Boolean
sorts RaddrSort
opns a,b : -> RaddrSort
     _eq _, _ne _       : RaddrSort, RaddrSort -> Bool
eqns forall x,y: RaddrSort
     ofsort Bool
     a eq a = true;
     b eq b = true;
```

```
     a eq b = false;
     x eq y = y eq x;
     x ne y = not(x eq y);
endtype (* R_ADDRESS *)

type WORKSTATION_DATA is Boolean
sorts Workstation_dataSort
opns no_wsdata: -> Workstation_dataSort
     _eq _, _ne _ : Workstation_dataSort,
                    Workstation_dataSort -> Bool
endtype (* WORKSTATION_DATA *)

type COMMANDER_DATA is Boolean
sorts Commander_dataSort
opns no_comdata: -> Commander_dataSort
     _eq _, _ne _ : Commander_dataSort,
                    Commander_dataSort -> Bool
endtype (* COMMANDER_DATA *)


(*
The R_PRODUCTreq and R_PRODUCTind primitives are used
in the transfer of only one product to and from a workstation.
The R_PRODUCT_DATAreq and R_PRODUCT_DATAind primitives
are used in the transfer of a product, together
with workstation_data to and from a workstation.
*)

type R_PRODUCT is PRODUCT, WORKSTATION_DATA, NaturalNumber
sorts R_PRODUCTreqSort, R_PRODUCTindSort,
      R_PRODUCT_DATAreqSort, R_PRODUCT_DATAindSort
opns R_PRODUCTreq        : ProductSort -> R_PRODUCTreqSort
     R_PRODUCTind        : ProductSort -> R_PRODUCTindSort
     R_PRODUCT_DATAreq   : ProductSort,Workstation_dataSort
                           -> R_PRODUCT_DATAreqSort
     R_PRODUCT_DATAind   : ProductSort,Workstation_dataSort
                           -> R_PRODUCT_DATAindSort
     R_PRODUCT           : R_PRODUCTreqSort -> ProductSort
     R_PRODUCT           : R_PRODUCTindSort -> ProductSort
     R_PRODUCT           : R_PRODUCT_DATAreqSort -> ProductSort
     R_PRODUCT           : R_PRODUCT_DATAindSort -> ProductSort
     R_PRODUCT_DATA      : R_PRODUCT_DATAreqSort
                           -> Workstation_dataSort
     R_PRODUCT_DATA      : R_PRODUCT_DATAindSort
                           -> Workstation_dataSort
```

```
      num_1,num_2,num_3,num_4,num_5,num_6,num_7,num_8,
      num_9,num_10        : -> R_PRODUCTreqSort
      val                 : ProductSort -> Nat
eqns forall pu:ProductSort, wd:Workstation_dataSort,
             numa,numb:ProductSort
      ofsort ProductSort
      R_PRODUCT(R_PRODUCTreq(pu))          = pu;
      R_PRODUCT(R_PRODUCTind(pu))          = pu;
      R_PRODUCT(R_PRODUCT_DATAreq(pu,wd)) = pu;
      R_PRODUCT(R_PRODUCT_DATAind(pu,wd)) = pu;
      ofsort Workstation_dataSort
      R_PRODUCT_DATA (R_PRODUCT_DATAreq(pu,wd)) = wd;
      R_PRODUCT_DATA (R_PRODUCT_DATAind(pu,wd)) = wd;
      ofsort Bool
      numa eq numb = val(numa) eq val(numb);
      numa ne numb = not(numa eq numb);
      numa lt numb = val(numa) lt val(numb);
      ofsort Nat
      val(R_PRODUCT(num_1)) = 0;
      val(R_PRODUCT(num_2)) = succ(val(R_PRODUCT(num_1)));
      val(R_PRODUCT(num_3)) = succ(val(R_PRODUCT(num_2)));
      val(R_PRODUCT(num_4)) = succ(val(R_PRODUCT(num_3)));
      val(R_PRODUCT(num_5)) = succ(val(R_PRODUCT(num_4)));
      val(R_PRODUCT(num_6)) = succ(val(R_PRODUCT(num_5)));
      val(R_PRODUCT(num_7)) = succ(val(R_PRODUCT(num_6)));
      val(R_PRODUCT(num_8)) = succ(val(R_PRODUCT(num_7)));
      val(R_PRODUCT(num_9)) = succ(val(R_PRODUCT(num_8)));
      val(R_PRODUCT(num_10))= succ(val(R_PRODUCT(num_9)));
endtype (* R_PRODUCT *)

type PR_SET is Set renamedby
      sortnames PRSet for Set
      opnnames empty_prset for {}
endtype (* PR_SET *)

type PRSET is PR_SET
      actualizedby R_PRODUCT,NaturalNumber using
      sortnames ProductSort for Element
                Bool        for FBool
                Nat         for FNat
endtype (* PRSET *)

type GENERIC_ADDRESS is Boolean, R_ADDRESS
opns generic_equal : RaddrSort, RaddrSort -> Bool
eqns forall addr1,addr2:RaddrSort
```

```
     ofsort Bool
     (addr1 eq addr2) implies
                      generic_equal(addr1,addr2) = true;
endtype (* GENERIC_ADDRESS *)


(*
The Route service quality consists of four elements:
 1 - transport time
 2 - transport capacity
 3 - priority
 4 - cost
*)


type TIME is
sorts Time
endtype (* TIME *)

type TRANSPORT_CAPACITY is
sorts Transport_capacity
endtype (* TRANSPORT_CAPACITY *)

type PRIORITY is
sorts Priority
endtype (* PRIORITY *)

type COST is
sorts Cost
endtype (* COST *)

type R_QOS is TIME, TRANSPORT_CAPACITY, PRIORITY, COST
sorts RqosSort
opns r_qos          : Time, Transport_capacity, Priority,
                      Cost -> RqosSort
     transport_time : RqosSort -> Time
     transport_cap  : RqosSort -> Transport_capacity
     priority       : RqosSort -> Priority
     costs          : RqosSort -> Cost
     no_qos         : -> RqosSort
eqns forall t_time:Time, t_cap:Transport_capacity,
            prio:Priority, cost:Cost
     ofsort Time
     transport_time (r_qos(t_time,t_cap,prio,cost)) = t_time;
     ofsort Transport_capacity
```

```
     transport_cap  (r_qos(t_time,t_cap,prio,cost)) = t_cap;
     ofsort Priority
     priority       (r_qos(t_time,t_cap,prio,cost)) = prio;
     ofsort Cost
     costs      (r_qos(t_time,t_cap,prio,cost)) = cost;
endtype (* R_QOS *)

type R_REPORT is
sorts R_Report
opns spontaneous_product, product_lost, sequence_error,
     station_will_not_accept : -> R_Report
endtype (* R_REPORT *)

type DELIVERY is
sorts Delivery
opns product_delivered, not_delivered : -> Delivery
endtype (* R_REPORT *)


(*
There are two types of options:
 1 - provider options which cannot be selected by the user;
     the implementer has the choice of incorporating them.
     These options provide extra functions.
 2 - user options can be selected in an interaction and can
     be used to select a particular type of service.
*)

type R_OPTIONS is Boolean
sorts R_OptionSort
opns Sequence,Generic_address,Acknowledge    : -> R_OptionSort
     Write,Read,Report,Series,Reset,Facility : -> R_OptionSort
     _eq_, _ne_, _lt_ : R_OptionSort,R_OptionSort  -> Bool
endtype (* R_OPTIONS *)

type R_REQUIREMENTS is Set renamedby
     sortnames Ruser_optsSort for Set
     opnnames  empty_reqs     for {}
endtype (* R_REQUIREMENTS *)

(*
When a product is sent, a set of user options can be
requested. The type Requirements models this set.
*)
```

```
type REQUIREMENTS is R_REQUIREMENTS
     actualizedby R_OPTIONS, NaturalNumber using
     sortnames R_OptionSort for Element
               Bool         for FBool
               Nat          for FNat
endtype (* REQUIREMENTS *)

type RSCommands is R_ADDRESS, GENERIC_ADDRESS, R_QOS,
      REQUIREMENTS, NaturalNumber, COMMANDER_DATA,
      R_REPORT
sorts R_PRODUCT_SENDreqSort, R_PRODUCT_SENDackSort,
      R_PRODUCT_ARRIVALindSort, R_FACreqSort, R_FACindSort,
      R_REPORTindSort, R_RESETreqSort, R_RESETindSort
opns RSCsource    : R_PRODUCT_SENDreqSort -> RaddrSort
     RSCsource    : R_PRODUCT_SENDackSort -> RaddrSort
     RSCsource    : R_PRODUCT_ARRIVALindSort -> RaddrSort
     RSCsource    : R_FACreqSort -> RaddrSort
     RSCsource    : R_FACindSort -> RaddrSort
     RSCsource    : R_REPORTindSort -> RaddrSort
     RSCsource    : R_RESETreqSort -> RaddrSort
     RSCsource    : R_RESETindSort -> RaddrSort
     RSCdest      : R_PRODUCT_SENDreqSort -> RaddrSort
     RSCdest      : R_PRODUCT_ARRIVALindSort -> RaddrSort
     RSCdest      : R_REPORTindSort -> RaddrSort
     RSCdest      : R_FACreqSort -> RaddrSort
     RSCdest      : R_FACindSort -> RaddrSort
     RSCqos       : R_PRODUCT_SENDreqSort -> RqosSort
     RSCqos       : R_PRODUCT_SENDackSort -> RqosSort
     RSCqos       : R_FACindSort -> RqosSort
     RSCcom_data  : R_PRODUCT_SENDreqSort
                    -> Commander_dataSort
     RSCcom_data  : R_PRODUCT_ARRIVALindSort
                    -> Commander_dataSort
     RSCcom_data  : R_REPORTindSort -> Commander_dataSort
     RSCuser_opts: R_PRODUCT_SENDreqSort -> Ruser_optsSort
     RSCuser_opts: R_FACindSort -> Ruser_optsSort
     RSCnum       : R_PRODUCT_SENDreqSort -> Nat
     RSCnum       : R_PRODUCT_ARRIVALindSort -> Nat
     RSCreport    : R_REPORTindSort -> R_Report
     R_PRODUCT_SENDreq : RaddrSort, RaddrSort, RqosSort,
                         Ruser_optsSort, Nat, Commander_dataSort
                         -> R_PRODUCT_SENDreqSort
     R_PRODUCT_SENDack     : RaddrSort, RqosSort
                             -> R_PRODUCT_SENDackSort
     R_PRODUCT_ARRIVALind : RaddrSort, RaddrSort, Nat,
```

```
                              Commander_dataSort
                              -> R_PRODUCT_ARRIVALindSort
      R_FACILITYreq : RaddrSort, RaddrSort -> R_FACreqSort
      R_FACILITYind : RaddrSort, RaddrSort, RqosSort,
                      Ruser_optsSort -> R_FACindSort
      R_REPORTind   : RaddrSort, RaddrSort, R_Report,
                      Commander_dataSort -> R_REPORTindSort
      R_RESETreq    : RaddrSort -> R_RESETreqSort
      R_RESETind    : RaddrSort -> R_RESETindSort

eqns forall src,dst:RaddrSort, q:RqosSort,
             user_opt:Ruser_optsSort, n:Nat,
             cd:Commander_dataSort, rp:R_Report
      ofsort RaddrSort
      RSCsource(R_PRODUCT_SENDreq(src,dst,q,user_opt,n,cd))=src;
      RSCsource(R_PRODUCT_SENDack(src,q)) = src;
      RSCsource(R_PRODUCT_ARRIVALind(dst,src,n,cd)) = src;
      RSCsource(R_FACILITYreq(src,dst)) = src;
      RSCsource(R_FACILITYind(src,dst,q,user_opt)) = src;
      RSCsource(R_REPORTind(dst,src,rp,cd)) = src;
      RSCsource(R_RESETreq(src)) = src;
      RSCsource(R_RESETind(src)) = src;
      RSCdest(R_PRODUCT_SENDreq(src,dst,q,user_opt,n,cd))=dst;
      RSCdest(R_PRODUCT_ARRIVALind(dst,src,n,cd)) = dst;
      RSCdest(R_REPORTind(dst,src,rp,cd)) = dst;
      RSCdest(R_FACILITYreq(src,dst)) = dst;
      RSCdest(R_FACILITYind(src,dst,q,user_opt)) = dst;
      ofsort RqosSort
      RSCqos(R_PRODUCT_SENDreq(src,dst,q,user_opt,n,cd))=q;
      RSCqos(R_PRODUCT_SENDack(src,q)) = q;
      RSCqos(R_FACILITYind(dst,src,q,user_opt))= q;
      ofsort Ruser_optsSort
      RSCuser_opts(R_PRODUCT_SENDreq(src,dst,q,user_opt,n,cd))
                                                  = user_opt;
      RSCuser_opts(R_FACILITYind(src,dst,q,user_opt))= user_opt;
      ofsort Nat
      RSCnum(R_PRODUCT_SENDreq(src,dst,q,user_opt,n,cd))=n;
      RSCnum(R_PRODUCT_ARRIVALind(src,dst,n,cd)) = n;
      ofsort Commander_dataSort
      RSCcom_data(R_PRODUCT_SENDreq(src,dst,q,user_opt,n,cd))
                                                        =cd;
      RSCcom_data(R_PRODUCT_ARRIVALind(dst,src,n,cd)) = cd;
      RSCcom_data(R_REPORTind(dst,src,rp,cd)) = cd;
      ofsort R_Report
      RSCreport(R_REPORTind(dst,src,rp,cd)) = rp;
```

```
endtype (* RSCommands *)


(*
Products that are transported are modelled by
RouteServiceObjects. These consist of a product,
workstation data, commander data and information
from the Route service provider. The parameters
workstation data, commander data and information
from the Route service provider, can have the
values no_data. The value no_data indicates that
the information is not present.
*)

type RSOBJECTS is R_PRODUCT, R_ADDRESS, COMMANDER_DATA,
     NaturalNumber, WORKSTATION_DATA, REQUIREMENTS, Boolean
sorts RSOSort
opns RSObject      : ProductSort, RaddrSort, RaddrSort, Nat,
           Nat, Ruser_optsSort, Commander_dataSort,
           Workstation_dataSort -> RSOSort
     RSOprod       : RSOSort -> ProductSort
     RSOsource     : RSOSort -> RaddrSort
     RSOdest       : RSOSort -> RaddrSort
     RSOnumber     : RSOSort -> Nat
     RSOserialnum  : RSOSort -> Nat
     RSOuser_opts  : RSOSort -> Ruser_optsSort
     RSOcom_data   : RSOSort -> Commander_dataSort
     RSOws_data    : RSOSort -> Workstation_dataSort
     _eq _, _ne _, _lt_ : RSOSort, RSOSort -> Bool

eqns forall p:ProductSort, src,dst:RaddrSort, n,sn:Nat,
     uo:Ruser_optsSort, cd:Commander_dataSort,
     wd:Workstation_dataSort, rso1,rso2:RSOSort
     ofsort ProductSort
     RSOprod       (RSObject(p,dst,src,n,sn,uo,cd,wd)) = p;
     ofsort RaddrSort
     RSOsource     (RSObject(p,dst,src,n,sn,uo,cd,wd)) = src;
     RSOdest       (RSObject(p,dst,src,n,sn,uo,cd,wd)) = dst;
     ofsort Nat
     RSOnumber     (RSObject(p,dst,src,n,sn,uo,cd,wd)) = n;
     RSOserialnum  (RSObject(p,dst,src,n,sn,uo,cd,wd)) = sn;
     ofsort Ruser_optsSort
     RSOuser_opts  (RSObject(p,dst,src,n,sn,uo,cd,wd)) = uo;
     ofsort Commander_dataSort
     RSOcom_data   (RSObject(p,dst,src,n,sn,uo,cd,wd)) = cd;
```

```
     ofsort Workstation_dataSort
     RSOws_data    (RSObject(p,dst,src,n,sn,uo,cd,wd)) = wd;
     ofsort Bool
     rso1 eq rso2 = (RSOprod(rso1) eq RSOprod(rso2)) and
    (RSOsource(rso1) eq RSOsource(rso2)) and
    (RSOdest(rso1) eq RSOdest(rso2)) and
    (RSOnumber(rso1) eq RSOnumber(rso2)) and
    (RSOserialnum(rso1) eq RSOserialnum(rso2)) and
    (RSOuser_opts(rso1) eq RSOuser_opts(rso2)) and
    (RSOcom_data(rso1) eq RSOcom_data(rso2)) and
    (RSOws_data(rso1) eq RSOws_data(rso2));
        rso1 ne rso2 = not(rso1 eq rso2)
endtype (* RSOBJECTS *)


(*
The RSQueue models the transport of products of
which the order must be maintained. The sender Adds
RSOBJECTS behind the queue and the receiver extracts
RSOBJECTS from the beginning of the queue Queue
(FIFO buffer)
*)

type RS_QUEUE is QUEUE renamedby
     sortnames RSQueue     for Queue
     opnnames  empty       for <>
endtype (* RS_QUEUE *)

type RSQUEUE is RS_QUEUE
     actualizedby RSOBJECTS, NaturalNumber using
     sortnames RSOSort for Element
               Bool    for Fbool
endtype (* RSQUEUE *)

type RS_SEQUENCE is RSQUEUE
opns same_elements : RSQueue, RSQueue -> Bool
     _IsIn _ : RSOSort, RSQueue -> Bool
     Remove : RSOSort, RSQueue -> RSQueue
eqns forall q1,q2:RSQueue, e1,e2:RSOSort
     ofsort Bool
     e1 IsIn empty = false;
endtype (* RS_SEQUENCE *)


(*
```

```
The RSSet models the transport of products in which
the order ned not be maintained. The sender Adds
RSOBJECTS the set and the receiver extracts RSOBJECTS
from the set. The transport system determines the order
in which the receiver is offered the RSOBJECTS.
*)

type RS_SET is Set renamedby
     sortnames RSSet       for Set
     opnnames  empty_rsset for {}
endtype (* RS_SET *)

type RSSET is RS_SET
     actualizedby RSOBJECTS,NaturalNumber using
     sortnames RSOSort for Element
               Bool    for FBool
               Nat     for FNat
endtype (* RSSET *)


(*
A generic Address represents a set of Route Addresses.
The function generic_equal has a Route or a Generic
Address as the First parameter and as second a Route
Address. The function delivers the value TRUE if two
Route Addresses are the same or if the second Address
element is from the set of Route Addresses, indicated
by the First Generic Address.
*)


(*
The INDICATION_MARKER_SET is the data model of the set
of serial numbers for which an indication has already
been given. This set is maintained in the receiving
part of the Route service.
*)

type INDICATION_SET is Set renamedby
     sortnames Marker_Set  for Set
     opnnames  empty_markers for {}
endtype (* INDICATION_SET *)

type INDICATION_MARKER_SET is INDICATION_SET
     actualizedby NaturalNumber using
```

```
     sortnames Nat     for Element
               Bool    for FBool
               Nat     for FNat
endtype (* INDICATION_MARKER_SET *)


(*
The parameters which characterise a Route consist of :
 - the local Route Address
 - the Address of the receiver
 - the available Route options
*)

type R_PARAMETERS is R_ADDRESS, Boolean, REQUIREMENTS
sorts Parameters
opns def_parameters : RaddrSort, RaddrSort, Ruser_optsSort
                      -> Parameters
     send_address_in : Parameters -> RaddrSort
     receive_address_in : Parameters -> RaddrSort
     service_requirements_in : Parameters -> Ruser_optsSort
     implemented : R_OptionSort, Parameters -> Bool

eqns forall ad1,ad2:RaddrSort, opts:Ruser_optsSort,
             opt:R_OptionSort
     ofsort RaddrSort
     send_address_in(def_parameters(ad1,ad2,opts)) = ad1;
     receive_address_in(def_parameters(ad1,ad2,opts)) = ad2;
     ofsort Ruser_optsSort
     service_requirements_in(def_parameters(ad1,ad2,opts))
                                                    = opts
     ofsort Bool
     implemented(opt,def_parameters(ad1,ad2,opts))
                                          = opt IsIn opts;

endtype (* R_PARAMETERS *)

(*
ROUTE SERVICE SPECIFICATION
Level 0
Process Description Part

The Route service is specified between a sender A
and a receiver B. The Route Service Specification
has the gates ct and st to exchange commands and
status, and the gates wa and wb to exchange products
```

```
with A and B respectively.
*)

behaviour

hide sync1,sync2,sync3 in
(
  PRODUCT_RECEIVE[wa,sync1]
 |[sync1]|
  COMMAND_RECEIVE[ct,sync1]
 |[sync1]|
  TRANSPORT[sync1,sync2]
 |[sync2]|
  PRODUCT_INDICATION[sync2,sync3,wb]
 |[sync2,sync3]|
  COMMAND_INDICATION[sync2,sync3,st]
)

where


process PRODUCT_RECEIVE[wa,sync1]:noexit
:=
  PRODUCT_ONLY[wa,sync1]

where

process PRODUCT_ONLY[wa,sync1]:noexit
:=
    wa ?p:R_PRODUCTreqSort
   ;sync1 ?any_command:R_PRODUCT_SENDreqSort
          !R_PRODUCT(p)
          !no_wsdata
   ;PRODUCT_ONLY[wa,sync1]

endproc (* PRODUCT_ONLY *)
endproc (* PRODUCT_RECEIVE *)

process COMMAND_RECEIVE[ct,sync1]:noexit
:=
   SEND_REQUEST_RECEIVE[ct,sync1]

where

process SEND_REQUEST_RECEIVE[ct,sync1]:noexit
```

```
:=
        ct ?sreq:R_PRODUCT_SENDreqSort
       ;sync1 !sreq
              ?any_product:ProductSort
              ?any_data:Workstation_dataSort
       ;SEND_REQUEST_RECEIVE[ct,sync1]

endproc (* SEND_REQUEST_RECEIVE *)
endproc (* COMMAND_RECEIVE *)

process TRANSPORT[sync1,sync2]:noexit
:=
   NORMAL_TRANSPORT[sync1,sync2](empty_prset)

where

process NORMAL_TRANSPORT[sync1,sync2](product_set:PRSet):noexit
:=

      sync1?a_command:R_PRODUCT_SENDreqSort
            ?prod:ProductSort
            ?work_data:Workstation_dataSort
    ;(let rso:RSOSort = RSObject(prod,RSCdest(a_command),
                                 RSCsource(a_command),
                                 RSCnum(a_command),0,
                                 RSCuser_opts(a_command),
                                 RSCcom_data(a_command),
                                 work_data)
      in
          NORMAL_TRANSPORT[sync1,sync2]
                          (Insert(prod,product_set))
     )
   []
     ([(product_set ne empty_prset) = true] ->
       (choice rso:RSOSort []
            [(RSOprod(rso) IsIn product_set) = true] ->
            i;sync2!rso
             ;NORMAL_TRANSPORT[sync1,sync2]
                     (Remove(RSOprod(rso),product_set))
       )
      )

endproc (* NORMAL_TRANSPORT *)
endproc (* TRANSPORT *)
```

```
process PRODUCT_INDICATION[sync2,sync3,wb]:noexit
:=
  PRODUCT_ONLY[sync2,sync3,wb]

where

process PRODUCT_ONLY[sync2,sync3,wb]:noexit
:=
        sync2 ?rso:RSOSort
       ;wb !RSOdest(rso) !R_PRODUCTind(RSOprod(rso))
       ;sync3 !product_delivered
       ;PRODUCT_ONLY[sync2,sync3,wb]

endproc (* PRODUCT_ONLY *)
endproc (* PRODUCT_INDICATION *)


process COMMAND_INDICATION[sync2,sync3,st]:noexit
:=
   ARRIVAL_INDICATION[sync2,sync3,st]

where

process ARRIVAL_INDICATION[sync2,sync3,st]:noexit
:=
   sync2 ?rso:RSOSort
  ;sync3 !product_delivered
  ;st !R_PRODUCT_ARRIVALind(RSOdest(rso),RSOsource(rso)
                           ,RSOnumber(rso),RSOcom_data(rso))
  ;ARRIVAL_INDICATION[sync2,sync3,st]

endproc (* ARRIVAL_INDICATION *)
endproc (* COMMAND_INDICATION *)


endspec (* ROUTE_SERVICE *)
```

Appendix B

LOTOS specification TS-Architecture version 2: TCS

```
specification ROUTER_TEST[ct,st,c,s,r,e,w1,w2,w3,w4]:noexit

library Boolean, NaturalNumber, SetElement, Set endlib

type REQUIREMENTS is
     sorts User_optsSort
endtype (* REQUIREMENTS *)

type WORLD_INFO is
     sorts World_InfoSort
endtype (* WORLD_INFO *)

type PRIORITY is
     sorts Priority
endtype (* PRIORITY *)

type DELIVERY is
     sorts Delivery
     opns  product_delivered : -> Delivery
endtype (* DELIVERY *)

type REASON_REJECT is
     sorts ReasonSort
     opns no_route_available: -> ReasonSort
endtype (* REASON_REJECT *)

type PRODUCT is NaturalNumber
     sorts ProductSort
     opns type1,type2,type3 : -> ProductSort
```

```
          ptype_not_available : -> ProductSort
          _eq _, _ne _, _lt _: ProductSort, ProductSort -> Bool
     eqns forall s,t: ProductSort
          ofsort Bool
          type1 eq type1 = true;
          type2 eq type2 = true;
          type3 eq type3 = true;
          type1 eq type2 = false;
          type1 eq type3 = false;
          type2 eq type3 = false;
          s eq t = t eq s;
          s ne t = not(s eq t);
endtype (* PRODUCT *)

type PRODUCT_INFO is NaturalNumber
     sorts ProductInfoSort
     opns _eq_ : ProductInfoSort, ProductInfoSort -> Bool
          transport_finished :-> ProductInfoSort
endtype (* PRODUCT_INFO *)

type FACILITY_INFO is
     sorts FacilityInfoSort
endtype (* FACILITY_INFO *)

type NODE_INFO is
     sorts NodeInfoSort
endtype (* NODE_INFO *)

type TYPE_OF_INFO is
     sorts TypeOfInfoSort
endtype (* TYPE_OF_INFO *)

type R_ADDRESS is NaturalNumber
     sorts RaddrSort
     opns a,b,c : -> RaddrSort
          _eq_ :RaddrSort,RaddrSort -> Bool
          val:RaddrSort -> Nat
     eqns forall x,y:RaddrSort
          ofsort Bool
          x eq y = val(x) eq val(y)
          ofsort Nat
          val(a) = 0;
          val(b) = succ(val(a));
          val(c) = succ(val(b))
endtype (* R_ADDRESS *)
```

```
type NODE_ID is NaturalNumber
     sorts NodeIdSort
     opns one,two,three: -> NodeIdSort
          all_nodes: -> NodeIdSort
          nid      : -> NodeIdSort
          _eq_,_ne_,_lt_ :NodeIdSort,NodeIdSort -> Bool
          val: NodeIdSort -> Nat
     eqns forall n1,n2: NodeIdSort
          ofsort Bool
          n1 eq n2 = val(n1) eq val(n2)
          ofsort Nat
          val(one)   = 0;
          val(two)   = succ(val(one));
          val(three) = succ(val(two))
endtype (* NODE_ID *)

type GATE_OUT is NaturalNumber
     sorts GoutSort
     opns g1,g2,g3,g4 : -> GoutSort
          _eq_ : GoutSort,GoutSort -> Bool
          val: GoutSort -> Nat
     eqns forall g,f:GoutSort
          ofsort Bool
          g eq f = val(g) eq val(f)
          ofsort Nat
          val(g1) = 0;
          val(g2) = succ(val(g1));
          val(g3) = succ(val(g2));
          val(g4) = succ(val(g3))
endtype (* GATE_OUT *)

type RECIPE is Set renamedby
     sortnames RECIPESort for Set
     opnnames  empty for {}
endtype (* RECIPE *)

type RRECIPE is RECIPE
     actualizedby MESSAGES, NaturalNumber using
     sortnames R_COMMANDSort for Element
               Bool          for FBool
               Nat           for FNat
endtype (* RRECIPE *)

type NODE_SET is Set renamedby
```

```
     sortnames NodeSetSort for Set
     opnnames  empty for {}
endtype (* NODE_SET *)

type NNODE_SET is NODE_SET
     actualizedby NODE_ID, NaturalNumber using
     sortnames NodeIdSort for Element
               Bool       for FBool
               Nat        for FNat
endtype (* NNODE_SET *)

type ACCOUNT_SET is Set renamedby
     sortnames AccountSet for Set
     opnnames  empty_account_set for {}
endtype (* ACCOUNT_SET *)

type AACOUNT_SET is ACCOUNT_SET
     actualizedby MESSAGES, NaturalNumber using
     sortnames ACCOUNTSort for Element
               Bool        for FBool
               Nat         for FNat
endtype (* AACCOUNT_SET *)

type SREQ_SET is Set renamedby
     sortnames SreqSet for Set
     opnnames  empty_sreq_set for {}
endtype (* SREQ_SET *)

type SSREQ_SET is SREQ_SET
     actualizedby MESSAGES, NaturalNumber using
     sortnames SENDreqSort for Element
               Bool        for FBool
               Nat         for FNat
endtype (* SSREQ_SET *)

type RCOM_SET is Set renamedby
     sortnames RcomSet for Set
     opnnames  empty_rcom_set for {}
endtype (* RCOM_SET *)

type RRCOM_SET is RCOM_SET
     actualizedby MESSAGES, NaturalNumber using
     sortnames R_COMMANDSort for Element
               Bool          for FBool
               Nat           for FNat
```

```
endtype (* RRCOM_SET *)

type MESSAGES is R_ADDRESS, REQUIREMENTS, NaturalNumber,
                 PRODUCT, PRIORITY, REASON_REJECT,
                 NODE_ID, GATE_OUT, FACILITY_INFO,
                 PRODUCT_INFO, NODE_INFO, TYPE_OF_INFO
sorts SENDreqSort, Transport_CompletedSort,
      Movement_CompletedSort,RESETcomSort, CANCELcomSort,
      STOP_STARTcomSort,AckSort,REJECTindSort,
      R_COMMANDSort,   R_COMMANDreqSort,
      RECIPEreqSort, ALARMSort, Product_Status_reqSort,
      Product_StatusSort, Facility_Status_ReqSort,
      Facility_StatusSort, Node_Status_ReqSort,
      Node_StatusSort,ACCOUNTSort
opns SENDreq             : RaddrSort,RaddrSort,ProductSort,
                           Priority,Nat -> SENDreqSort
     RESET               : -> RESETcomSort
     CANCELcom           : SENDreqSort -> CANCELcomSort
     REJECTind           : ReasonSort -> REJECTindSort
     R_COMMANDreq        : ProductSort,NodeIdSort
                           -> R_COMMANDreqSort
     R_COMMAND           : NodeIdSort,ProductSort,Priority,
                           Nat,GoutSort -> R_COMMANDSort
     no_route_command    : -> R_COMMANDSort
     RECIPEreq           : SENDreqSort,Facility_StatusSort
                           -> RECIPEreqSort
     stp,strt            : ->STOP_STARTcomSort
     stp_ack, strt_ack,
     rst_ack, cncl_ack,
     acknowledge         : -> AckSort
     Transport_Completed: ProductSort
                           -> Transport_CompletedSort
     Movement_Completed : ProductSort,NodeIdSort
                           -> Movement_CompletedSort
     Product_Status_Req : ProductSort,TypeOfInfoSort ->
                           Product_Status_ReqSort
     Product_Status      : ProductSort,TypeOfInfoSort,
                           ProductInfoSort -> Product_StatusSort
     Facility_Status_Req: TypeOfInfoSort
                           -> Facility_Status_ReqSort
     Facility_Status     : TypeOfInfoSort,FacilityInfoSort ->
                           Facility_StatusSort
     Node_Status_Req     : NodeIdSort,TypeOfInfoSort
                           -> Node_Status_ReqSort
     Node_Status         : NodeIdSort,TypeOfInfoSort,
```

```
                          NodeInfoSort -> Node_StatusSort
     Alarm              : NodeIdSort -> ALARMSort
     Account            : ProductSort -> ACCOUNTSort
     Transport          : Product_StatusSort -> ProductInfoSort
     Source             : SENDreqSort -> RaddrSort
     Dest               : SENDreqSort -> RaddrSort
     Ptype              : SENDreqSort -> ProductSort
     Ptype              : R_COMMANDreqSort -> ProductSort
     Ptype              : Transport_CompletedSort
                          -> ProductSort
     Ptype              : Product_StatusSort -> ProductSort
     Ptype              : R_COMMANDSort -> ProductSort
     Prior              : SENDreqSort -> Priority
     Quantity           : SENDreqSort -> Nat
     Sendreq            : RECIPEreqSort -> SENDreqSort
     Sendreq            : ACCOUNTSort -> SENDreqSort
     Sendreq            : CANCELcomSort -> SENDreqSort
     Sendreq            : Product_StatusSort -> SENDreqSort
     Reason             : REJECTindSort -> ReasonSort
     Reason             : Bool -> ReasonSort
     Nodeid             : ALARMSort -> NodeIdSort
     Gout               : R_COMMANDSort -> GoutSort
     _eq_,_ne_,_lt_     : R_COMMANDSort,R_COMMANDSort -> Bool
     _eq_,_ne_,_lt_     : ACCOUNTSort,ACCOUNTSort -> Bool
     _eq_,_ne_,_lt_     : SENDreqSort,SENDreqSort -> Bool
eqns forall src,dst:RaddrSort, ptype:ProductSort,
            prior:Priority,n:Nat, sreq:SENDreqSort,
            reason:ReasonSort,
            facility_status:Facility_StatusSort,
            node_id:NodeIdSort
     ofsort RaddrSort
     Source   (SENDreq(src,dst,ptype,prior,n)) = src;
     Dest     (SENDreq(src,dst,ptype,prior,n)) = dst
     ofsort ProductSort
     Ptype    (SENDreq(src,dst,ptype,prior,n)) = ptype;
     Ptype    (R_COMMANDreq(ptype,node_id)) = ptype;
     Ptype    (Transport_Completed(ptype)) = ptype;
     ofsort Priority
     Prior    (SENDreq(src,dst,ptype,prior,n)) = prior
     ofsort Nat
     Quantity (SENDreq(src,dst,ptype,prior,n)) = n
     ofsort SENDreqSort
     Sendreq  (RECIPEreq(sreq,facility_status)) = sreq;
     Sendreq  (CANCELcom(sreq)) = sreq
     ofsort ReasonSort
```

```
     Reason    (REJECTind(reason)) = reason
     ofsort NodeIdSort
     Nodeid    (Alarm(node_id)) = node_id
endtype (* RC_MESSAGES *)

behaviour

hide c,s in

ROUTER[ct,st,c,s,r,e]
|[c,s]|
NODE[w1,w2,w3,w4,c,s](nid)

where

process ROUTER[ct,st,c,s,r,e]:noexit
:=
     hide sync0,sync1,sync2 in

     ( ROUTE_DECOMPOSER[ct,c,r,sync0,sync2]
      |[sync2]|
       STATUS_COLLECTOR[st,s,e,sync1,sync2]
     )
      |[sync0,sync1]|
       MONITOR[sync0,sync1]

where

process ROUTE_DECOMPOSER[ct,c,r,sync0,sync2]:noexit
:=
     hide ex,rec in

      RECEIVE_COMMANDS[ct,ex]
     |[ex]|
      EXECUTE_COMMANDS[c,ex,rec,sync0,sync2]
     |[rec]|
      RECIPE_MANAGER[r,rec]

where


process RECEIVE_COMMANDS[ct,ex]:noexit
:=
   ( ct ?sreq:SENDreqSort
    ;ex !sreq
```

```
     ;RECEIVE_COMMANDS[ct,ex]
    )
   []
    ( ct ?reset_command:RESETcomSort
     ;ex !reset_command
     ;RECEIVE_COMMANDS[ct,ex]
    )
   []
    ( ct ?cancel_command:CANCELcomSort
     ;ex !cancel_command
     ;RECEIVE_COMMANDS[ct,ex]
    )
   []
    ( ct ?stop_command:STOP_STARTcomSort
     ;ex !stop_command
     ;(( ct ?reset_command:RESETcomSort
        ;ex !reset_command
        ;ct ?start_command:STOP_STARTcomSort
        ;ex !start_command
        ;RECEIVE_COMMANDS[ct,ex]
       )
      []
       ( ct ?cancel_command:CANCELcomSort
        ;ex !cancel_command
        ;ct ?start_command:STOP_STARTcomSort
        ;ex !start_command
        ;RECEIVE_COMMANDS[ct,ex]
       )
      []
       ( ct ?start_command:STOP_STARTcomSort
        ;ex !start_command
        ;RECEIVE_COMMANDS[ct,ex]
       )
      )
    )
   []
    ( ex ?reject_indication:REJECTindSort
     ;ct !reject_indication
     ;RECEIVE_COMMANDS[ct,ex]
    )

endproc (* RECEIVE_COMMANDS *)

process EXECUTE_COMMANDS[c,ex,rec,sync0,sync2]:noexit
:=
```

```
      SET_TABLES[c,ex,rec,sync0]
     |||
      RECEIVE_REJECTS[c,rec]
     |||
      RESET_TABLES[c,ex,sync0]
     |||
      CANCEL_COMMANDS[c,ex,sync0,sync2]
     |||
      STOP_START_COMMANDS[c,ex,sync0]
     |||
      PRESENT_DEFAULTS[c,rec]

where

process SET_TABLES[c,ex,rec,sync0]:noexit
:=
      ex ?sreq:SENDreqSort
     ;sync0 !sreq
     ;sync0 ?facility_status:Facility_StatusSort
     ;rec !RECIPEreq(sreq,facility_status)
     ;rec ?recipe:RECIPESort
     ;(  [(recipe eq empty) = true] ->
           ex !REJECTind(no_route_available)
          ;SET_TABLES[c,ex,rec,sync0]
       [][(recipe eq empty) = false] ->
           SET_NODE_TABLES[c,ex](recipe)
           >>
           SET_TABLES[c,ex,rec,sync0]
      )

     where

     process SET_NODE_TABLES[c,ex](recipe:RECIPESort):exit
     :=
          [(recipe eq empty) = true] ->
            exit
        [][(recipe eq empty) = false] ->
           ( choice route_command:R_COMMANDSort []
                    [(route_command IsIn Recipe) = true] ->
             c !route_command
            ;SET_NODE_TABLES[c,ex](Remove(route_command,recipe))
           )
     endproc (* SET_NODE_TABLES *)
endproc (* SET_TABLES *)
```

```
process RECEIVE_REJECTS[c,rec]:noexit
:=
     c ?reject_indication:REJECTindSort ?nodeid:NodeIdSort
    ;rec !reject_indication !nodeid
    ;RECEIVE_REJECTS[c,rec]

endproc (* RECEIVE_REJECTS *)

process RESET_TABLES[c,ex,sync0]:noexit
:=
     ex ?reset:RESETcomSort
    ;c !reset !all_nodes
    ;sync0 !reset
    ;RESET_TABLES[c,ex,sync0]

endproc (* RESET_TABLES *)

process CANCEL_COMMANDS[c,ex,sync0,sync2]:noexit
:=
    ( ex ?cancel:CANCELcomSort
     ;c !cancel !all_nodes
     ;sync0 !cancel
     ;CANCEL_COMMANDS[c,ex,sync0,sync2]
    )
   []
    ( sync2 ?cancel:CANCELcomSort
     ;c !cancel !all_nodes
     ;sync0 !cancel
     ;CANCEL_COMMANDS[c,ex,sync0,sync2]
    )
endproc (* CANCEL_COMMANDS *)

process STOP_START_COMMANDS[c,ex,sync0]:noexit
:=
     ex ?stop_or_start:STOP_STARTcomSort
    ;c !stop_or_start !all_nodes
    ;sync0 !stop_or_start
    ;STOP_START_COMMANDS[c,ex,sync0]

endproc (* STOP_START_COMMANDS *)

process PRESENT_DEFAULTS[c,rec]:noexit
:=
     c ?route_command_request:R_COMMANDreqSort
    ;rec !route_command_request
```

```
    ;rec ?route_command:R_COMMANDSort
    ;c !route_command
    ;PRESENT_DEFAULTS[c,rec]

endproc (* PRESENT_DEFAULTS *)
endproc (* EXECUTE_COMMANDS *)


process RECIPE_MANAGER[r,rec]:noexit
:=
   ( rec ?recipe_request:RECIPEreqSort
    ;GET_RECIPE(recipe_request)
    >>  accept recipe:RECIPESort,recipe_request:RECIPEreqSort
        in
        [(recipe eq empty) = true] ->
            r !recipe_request
           ;r ?recipe:RECIPESort
           ;rec !recipe
           ;RECIPE_MANAGER[r,rec]
      [][(recipe eq empty) = false] ->
            rec !recipe
           ;RECIPE_MANAGER[r,rec]
   )
  []
   (  rec ?route_command_request:R_COMMANDreqSort
    ;(  [(Ptype(route_command_request) eq
          ptype_not_available) = false] ->
          GET_ROUTE_COMMAND(route_command_request)
          >> accept route_command:R_COMMANDSort
             in
             exit(route_command)
      [][(Ptype(route_command_request) eq
          ptype_not_available) = true] ->
          GET_DEFAULT_COMMAND(route_command_request)
          >> accept route_command:R_COMMANDSort
             in
             exit(route_command)
     )
     >> accept route_command:R_COMMANDSort
        in
        rec !route_command
       ;RECIPE_MANAGER[r,rec]
   )
  []
   ( rec ?reject_indication:REJECTindSort ?nodeid:NodeIdSort
```

```
     ;r !reject_indication !nodeid
     ;RECIPE_MANAGER[r,rec]
    )

endproc (* RECIPE_MANAGER *)
endproc (* ROUTE_DECOMPOSER *)

process STATUS_COLLECTOR[st,s,e,sync1,sync2]:noexit
:=
     hide pr,eval in

      RECEIVE_STATUS[s,pr]
     |[pr]|
      EXECUTE_STATUS[pr,sync1,sync2,st,eval]
     |[eval]|
      EVALUATION_TABLE_MANAGER[eval,e]

where

process RECEIVE_STATUS[s,pr]:noexit
:=
    (  s ?movement_completed:Movement_CompletedSort
      ;pr !movement_completed
      ;RECEIVE_STATUS[s,pr]
    )
   []
    (  s ?node_status:Node_StatusSort
      ;pr !node_status
      ;RECEIVE_STATUS[s,pr]
    )
   []
    (  s ?alarm:ALARMSort
      ;pr !alarm
      ;RECEIVE_STATUS[s,pr]
    )
   []
    (  s ?ack:AckSort ?nodeid:NodeIdSort
      ;pr !ack
      ;RECEIVE_STATUS[s,pr]
    )
   []
    (  pr ?node_status_req:Node_Status_reqSort
      ;s !node_status_req
      ;s ?node_status:Node_StatusSort
      ;pr !node_status
```

```
      ;RECEIVE_STATUS[s,pr]
    )

endproc (* RECEIVE_STATUS *)

process EXECUTE_STATUS[pr,sync1,sync2,st,eval]:noexit
:=
     UPDATE_PRODUCT_STATUS[pr,sync1,sync2,st,eval]
    |||
     UPDATE_FACILITY_STATUS[pr,sync1,sync2,st,eval]
    |||
     EXECUTE_ACKNOWLEDGES[pr,sync1,sync2,st,eval]
    |||
     GIVE_STATUS[pr,sync1,sync2,st,eval]

    where

    process UPDATE_PRODUCT_STATUS[pr,sync1,sync2,
                                          st,eval]:noexit
    :=
         pr ?movement_completed:Movement_CompletedSort
        ;sync1 !movement_completed
        ;sync1 ?product_status:Product_StatusSort
        ;(  [(Transport(product_status) eq
              transport_finished) = false] ->
              eval !product_status
             ;UPDATE_PRODUCT_STATUS[pr,sync1,sync2,st,eval]
          [][(Transport(product_status) eq
              transport_finished) = true] ->
              st !Transport_Completed(Ptype(product_status))
             ;eval !product_status
             ;sync2 !CANCELcom(Sendreq(product_status))
             ;UPDATE_PRODUCT_STATUS[pr,sync1,sync2,st,eval]
         )
    endproc (* UPDATE_PRODUCT_STATUS *)

    process UPDATE_FACILITY_STATUS[pr,sync1,sync2,
                                          st,eval]:noexit
    :=
         ( pr ?node_status:Node_statusSort
          ;sync1 !node_status
          ;UPDATE_FACILITY_STATUS[pr,sync1,sync2,st,eval]
         )
        []
         ( pr ?alarm:AlarmSort
```

```
        ;sync1 !alarm
        ;st !alarm
        ;eval !alarm
        ;UPDATE_FACILITY_STATUS[pr,sync1,sync2,st,eval]
      )
endproc (* UPDATE_FACILITY_STATUS *)


process EXECUTE_ACKNOWLEDGES[pr,sync1,sync2,st,eval]:noexit
:=
     pr ?ack:AckSort ?nodeid:NodeIdSort
    ;ACK_RECEIVE[pr,st](Remove(nodeid,All_Nodes))
     >>
       EXECUTE_ACKNOWLEDGES[pr,sync1,sync2,st,eval]

   where

   process ACK_RECEIVE[pr,st](node_set:NodeSetSort):exit
   :=
        [(node_set eq empty) = false] ->
          pr ?ack:AckSort ?nodeid:NodeIdSort
         ;ACK_RECEIVE[pr,st](Remove(nodeid,node_set))
      [][(node_set eq empty) = true] ->
          (let ack:AckSort = acknowledge
           in
           st !ack
          ;exit
          )
   endproc (* ACK_RECEIVE *)
endproc (* EXECUTE_ACKNOWLEDGES *)

process GIVE_STATUS[pr,sync1,sync2,st,eval]:noexit
:=
   ( (  st ?product_status_req:Product_Status_ReqSort
       ;sync1 !product_status_req
       ;sync1 ?product_status:Product_StatusSort
       ;st !product_status
       ;GIVE_STATUS[pr,sync1,sync2,st,eval]
     )
    []
     (  st ?facility_status_req:Facility_Status_ReqSort
       ;sync1 !facility_status_req
       ;sync1 ?facility_status:Facility_StatusSort
       ;st !facility_status
       ;GIVE_STATUS[pr,sync1,sync2,st,eval]
```

```
          )
         []
          (  st ?node_status_req:Node_Status_ReqSort
            ;pr !node_status_req
            ;pr ?node_status:Node_StatusSort
            ;st !node_status
            ;GIVE_STATUS[pr,sync1,sync2,st,eval]
          )
        )
       []
        ( (  eval ?product_status_req:Product_Status_ReqSort
            ;sync1 !product_status_req
            ;sync1 ?product_status:Product_StatusSort
            ;eval !product_status
            ;GIVE_STATUS[pr,sync1,sync2,st,eval]
          )
         []
          (  eval ?facility_status_req:Facility_Status_ReqSort
            ;sync1 !facility_status_req
            ;sync1 ?facility_status:Facility_StatusSort
            ;eval !facility_status
            ;GIVE_STATUS[pr,sync1,sync2,st,eval]
          )
         []
          (  eval ?node_status_req:Node_Status_ReqSort
            ;pr !node_status_req
            ;pr ?node_status:Node_StatusSort
            ;eval !node_status
            ;GIVE_STATUS[pr,sync1,sync2,st,eval]
          )
        )
    endproc (* GIVE_STATUS *)

endproc (* EXECUTE_STATUS *)

process EVALUATION_TABLE_MANAGER[eval,e]:noexit
:=
   (  eval ?product_status:Product_StatusSort
     ;UPDATE_PES_TABLES[e]
      >>
      EVALUATION_TABLE_MANAGER[eval,e]
   )
  []
   (  eval ?node_status:Node_StatusSort
     ;UPDATE_PES_TABLES[e]
```

```
      >>
      EVALUATION_TABLE_MANAGER[eval,e]
   )
  []
   (  eval ?facility_status:Facility_StatusSort
     ;UPDATE_PES_TABLES[e]
      >>
      EVALUATION_TABLE_MANAGER[eval,e]
   )
  []
   (  e ?product_status_req:Product_Status_ReqSort
     ;eval !product_status_req
     ;EVALUATION_TABLE_MANAGER[eval,e]
   )
  []
   (  e ?node_status_req:Node_Status_ReqSort
     ;eval !node_status_req
     ;EVALUATION_TABLE_MANAGER[eval,e]
   )
  []
   (  e ?facility_status_req:Facility_Status_ReqSort
     ;eval !facility_status_req
     ;EVALUATION_TABLE_MANAGER[eval,e]
   )

endproc (* EVALUATION_TABLE_MANAGER *)
endproc (* STATUS_COLLECTOR *)

process MONITOR[sync0,sync1]:noexit
:=
     hide u,w1,w2 in

   ( UPDATE_SEND_COMMANDS[sync0,w1,u](empty_sreq_set)
    |[u]|
     UPDATE_STATUS[sync1,w2,u](empty_account_set)
   )
    |[w1,w2]|
     WORLD[w1,w2]

where

process UPDATE_SEND_COMMANDS[sync0,w1,u]
                             (sreq_set:SreqSet):noexit
:=
   ( sync0 ?sreq:SENDreqSort
```

```
     ;u !sreq
     ;w1 !sreq
     ;w1 ?facility_status:Facility_StatusSort
     ;sync0 !facility_status
     ;UPDATE_SEND_COMMANDS[sync0,w1,u]
                          (Insert(sreq,sreq_set))
    )
   []
    ( sync0 ?cancel:CANCELcomSort
     ;u !cancel
     ;UPDATE_SEND_COMMANDS[sync0,w1,u]
                          (Remove(Sendreq(cancel),sreq_set))
    )
   []
    ( sync0 ?reset:RESETcomSort
     ;u !reset
     ;UPDATE_SEND_COMMANDS[sync0,w1,u](empty_sreq_set)
    )
   []
    ( sync0 ?stp:STOP_STARTcomSort
     ;sync0 ?strt:STOP_STARTcomSort
     ;UPDATE_SEND_COMMANDS[sync0,w1,u](sreq_set)
    )

endproc (* UPDATE_SEND_COMMANDS *)

process UPDATE_STATUS[sync1,w2,u]
                      (account_set:AccountSet):noexit
:=
    ( u ?sreq:SENDreqSort
     ;MAKE_ACCOUNT(sreq)
      >> accept product_status:Product_StatusSort
         in
         UPDATE_STATUS[sync1,w2,u]
                      (Insert(Account(Ptype(product_status)),
                              account_set))
    )
   []
    ( sync1 ?movement_completed:Movement_CompletedSort
     ;UPDATE_ACCOUNT(movement_completed)
      >> accept product_status:Product_StatusSort
         in
         sync1 !product_status
     ;(  [(Transport(product_status) eq
           transport_finished) = false] ->
```

```
          UPDATE_STATUS[sync1,w2,u](account_set)
      [][(Transport(product_status) eq
          transport_finished) = true] ->
          UPDATE_STATUS[sync1,w2,u]
                        (Remove(Account(Ptype(product_status)),
                                account_set))
    )
  )
  []
   ( sync1 ?product_status_req:Product_Status_ReqSort
    ;GET_PRODUCT_STATUS(product_status_req)
     >> accept product_status:Product_StatusSort
        in
        sync1 !product_status
       ;UPDATE_STATUS[sync1,w2,u](account_set)
  )
  []
   ( sync1 ?facility_status_req:Facility_Status_ReqSort
    ;w2 !facility_status_req
    ;w2 ?facility_status:Facility_StatusSort
    ;sync1 !facility_status
    ;UPDATE_STATUS[sync1,w2,u](account_set)
  )
  []
   ( sync1 ?node_status:Node_StatusSort
    ;w2 !node_status
    ;UPDATE_STATUS[sync1,w2,u](account_set)
  )
  []
   ( sync1 ?alarm:AlarmSort
    ;w2 !alarm
    ;UPDATE_STATUS[sync1,w2,u](account_set)
  )
  []
   ( u ?cancel:CANCELcomSort
    ;UPDATE_STATUS[sync1,w2,u]
                  (Remove(Account(Ptype(Sendreq(cancel))),
                          account_set))
  )
  []
   ( u ?reset:RESETcomSort
    ;UPDATE_STATUS[sync1,w2,u](empty_account_set)
  )

endproc (* UPDATE_STATUS *)
```

```
process WORLD[w1,w2]:noexit
:=
      ( w1 ?sreq:SENDreqSort
       ;GET_FACILITY_STATUS(sreq)
        >> accept facility_status:Facility_StatusSort
           in
           w1 !facility_status
          ;WORLD[w1,w2]
      )
     []
      ( w2 ?facility_status_req:Facility_Status_ReqSort
       ;GET_FACILITY_STATUS(facility_status_req)
        >> accept facility_status:Facility_StatusSort
           in
           w2 !facility_status
          ;WORLD[w1,w2]
      )
     []
      ( w2 ?node_status:Node_StatusSort
       ;UPDATE_FACILITY_STATUS
        >>
        WORLD[w1,w2]
      )
     []
      ( w2 ?alarm:AlarmSort
       ;UPDATE_FACILITY_STATUS
        >>
        WORLD[w1,w2]
      )

endproc (* WORLD *)
endproc (* MONITOR *)

endproc (* ROUTER *)

process NODE[w1,w2,w3,w4,c,s](nid:NodeIdSort):noexit
:=
      hide sync0,sync1,sync2 in

      PRODUCT_RECEIVE[w1,w2,sync0,sync1]
     |[sync0,sync1]|
      COMMAND_RECEIVE[c,sync0,sync1](nid,empty_rcom_set)
     |[sync1]|
      MOVEMENT[sync1,sync2]
```

```
    |[sync2]|
     STATUS_INDICATION[sync2,s](nid)
    |[sync2]|
     PRODUCT_INDICATION[w3,w4,sync2]

where

process PRODUCT_RECEIVE[w1,w2,sync0,sync1]:noexit
:=
   ( w1 ?ptype:ProductSort
    ;sync0 !ptype
    ;sync1 ?gout:GoutSort ?ptype:ProductSort
    ;PRODUCT_RECEIVE[w1,w2,sync0,sync1]
   )
  []
   ( w2 ?ptype:ProductSort
    ;sync0 !ptype
    ;sync1 ?gout:GoutSort ?ptype:ProductSort
    ;PRODUCT_RECEIVE[w1,w2,sync0,sync1]
   )
endproc (* PRODUCT_RECEIVE *)

process COMMAND_RECEIVE[c,sync0,sync1](nid:NodeIdSort,
                          r_command_set:RcomSet):noexit
:=
     ( c ?route_command:R_COMMANDSort !nid
      ;CHECK_COMMAND(route_command)
       >> accept reject:Bool,route_command:R_COMMANDSort
          in
             [reject = true] ->
              c !REJECTind(Reason(reject)) !nid
             ;COMMAND_RECEIVE[c,sync0,sync1](nid,r_command_set)
           [][reject = false] ->
              COMMAND_RECEIVE[c,sync0,sync1](nid,
                           Insert(route_command,r_command_set))
     )
   []
     ( c ?reset:RESETcomSort !nid
      ;sync1 !reset
      ;COMMAND_RECEIVE[c,sync0,sync1](nid,empty_rcom_set)
     )
   []
     ( c ?cancel:CANCELcomSort !nid
      ;GET_COMMAND(Ptype(Sendreq(cancel)))
       >> accept route_command:R_COMMANDSort,
```

```
                cancel:CANCELcomSort
          in
              [(route_command eq no_route_command) = true] ->
                sync1 !cancel
               ;COMMAND_RECEIVE[c,sync0,sync1]
                                (nid,r_command_set)
            [][(route_command eq no_route_command) = false] ->
                sync1 !cancel
               ;COMMAND_RECEIVE[c,sync0,sync1](nid,
                           Remove(route_command,r_command_set))
     )
    []
     ( c ?stop_or_start:STOP_STARTcomSort !nid
      ;sync1 !stop_or_start
      ;COMMAND_RECEIVE[c,sync0,sync1](nid,r_command_set)
     )
    []
     ( sync0 ?ptype:ProductSort
      ;GET_COMMAND(ptype)
       >> accept route_command:R_COMMANDSort,ptype:ProductSort
          in
              [(route_command eq no_route_command) = true] ->
                c !R_COMMANDreq(ptype,nid)
               ;COMMAND_RECEIVE[c,sync0,sync1]
                                (nid,r_command_set)
            [][(route_command eq no_route_command) = false] ->
                sync1 !Gout(route_command)
                      !Ptype(route_command)
               ;COMMAND_RECEIVE[c,sync0,sync1]
                                (nid,r_command_set)
     )

endproc (* COMMAND_RECEIVE *)

process MOVEMENT[sync1,sync2]:noexit
:=
     ( sync1 ?gout:GoutSort ?ptype:ProductSort
      ;i
      ;sync2 !gout !ptype
      ;MOVEMENT[sync1,sync2]
     )
    []
     ( sync1 ?reset:RESETcomSort
      ;sync2 !reset
      ;MOVEMENT[sync1,sync2]
```

```
      )
     []
      ( sync1 ?cancel:CANCELcomSort
       ;sync2 !cancel
       ;MOVEMENT[sync1,sync2]
      )
     []
      ( sync1 ?stop_or_start:STOP_STARTcomSort
       ;sync2 !stop_or_start
       ;MOVEMENT[sync1,sync2]
      )

endproc (* MOVEMENT *)

process STATUS_INDICATION[sync2,s](nid:NodeIdSort):noexit
:=
      ( sync2 ?gout:GoutSort ?ptype:ProductSort
       ;sync2 ?product_delivered:Delivery
       ;s !Movement_Completed(ptype,nid)
       ;STATUS_INDICATION[sync2,s](nid)
      )
     []
      ( s ?node_status_req:Node_Status_ReqSort
       ;GET_NODE_STATUS(node_status_req)
        >> accept node_status:Node_StatusSort
           in
           s !node_status
          ;STATUS_INDICATION[sync2,s](nid)
      )
     []
      ((  sync2 ?reset:RESETcomSort ;exit
        []sync2 ?cancel:CANCELcomSort ;exit
        []sync2 ?stop_or_start:STOP_STARTcomSort ;exit
       ) >>
            s !acknowledge
           ;STATUS_INDICATION[sync2,s](nid)
      )
endproc (* STATUS_INDICATION *)

process PRODUCT_INDICATION[w3,w4,sync2]:noexit
:=
      sync2 ?gout:GoutSort ?ptype:ProductSort
     ;(    [(gout eq g3) = true] -> w3 !ptype
                                 ;sync2 !product_delivered
                                 ;PRODUCT_INDICATION[w3,w4,sync2]
```

```
        [][(gout eq g4) = true] -> w4 !ptype
                                ;sync2 !product_delivered
                                ;PRODUCT_INDICATION[w3,w4,sync2]
     )
endproc (* PRODUCT_INDICATION *)
endproc (* NODE *)

endspec
```