POOSL : an object-oriented specification language for the analysis and design of hardware/software systems

Research Report
ISSN 0167-9708
Coden: TEUEDE

Eindhoven
University of Technology
Netherlands

Faculty of Electrical Engineering

POOSL:
An Object-Oriented Specification Language for the Analysis and Design of Hardware / Software Systems

by
J.P.M. Voeten

EUT Report 95-E-290
ISBN 90-6144.290-7
May 1995

Eindhoven University of Technology Research Reports

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
Eindhoven, The Netherlands

ISSN 0167-9708 Coden: TEUEDE

POOSL: An Object-Oriented Specification Language for the Analysis and Design of Hardware/Software Systems

by

J.P.M. Voeten

EUT Report 95-E-290
ISBN 90-6144-290-7

Eindhoven
May 1995

POOSL: An Object-Oriented Specification Language
for the Analysis and Design of Hardware/Software Systems

J.P.M. Voeten

Abstract

POOSL (Parallel Object-Oriented Specification Language) is meant for the specification, design, and description of systems which contain a mixture of software and hardware components. It is based upon the object-oriented paradigm to support flexible and reusable design, as well as on the basic concepts of CCS to enable formal verification, simulation, and transformation of specifications. POOSL differs from traditional (parallel) object-oriented (specification) languages in a number of ways. First of all, it allows explicit representation of system architecture and hierarchy. Further, objects in POOSL communicate through channels using a one-way synchronous message-passing mechanism, allowing true parallelism. Finally, the language explicitly distinguishes (statically interconnected) process objects from (dynamically moving) data objects. This, together with the fact that POOSL supports tail recursion, creates the possibility to specify the (communication) behaviour of (systems of) objects in an abstract and elegant way.

Keywords: object-oriented methods, specification languages, formal specification.

Voeten J.P.M.
POOSL: An object-oriented specification language
for the analysis and design of hardware/software systems.
Eindhoven : Faculty of Electrical Engineering, Eindhoven University of Technology, 1995.
EUT Report 95-E-290

Address of the author:
Section of Digital Information Systems
Faculty of Electrical Engineering
Eindhoven University of Technology
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands

Table of Contents

Acknowledgements

I sincerely thank R. Liskova, P. van der Putten, A. van Rangelrooij, M. Stevens, A. Verschueren, and M. van Weert for reading earlier drafts of this report and for many fruitful discussions.

List of Figures

To Inge

Chapter 1

Introduction

1.1 History and Background

The specification language presented in this report has its roots in [Ver92]. This thesis describes how object-oriented techniques can be applied to the design of systems which contain a mixture of hardware and software modules. [Ver92] also presents a design methodology, an object model based upon Smalltalk [GR89], and a simulation environment meant for the specification, verification, and simulation of complex (communicating) systems. The design methodology is split into the following three phases:

- *High-level behaviour analysis.* The system is analyzed and an abstract specification (the behaviour (object) model) of the functionality is built. Such a specification consists of a set of (process) objects which communicate through static channels. In the analysis phase a specification is architecture and implementation independent.

- *High-level architecture synthesis.* An architecture (object) model or specification is made. Such a specification describes, in addition to a more detailed behaviour or functionality, the architecture of the system to implement. For every entity and channel in the specification it is decided how it will be implemented. In the ideal situation the architecture model could be constructed directly from the behaviour model by defining the implementation of objects and channels. However, because of technological, architectural or timing constraints, this might be impossible. Also the behaviour model might not be detailed enough. In these cases the behaviour model has to be transformed into an architecture model which meets the extra constraints and which has the desired level of detail. These transformations are performed by predefined so-called functionality-preserving transformations. Non-predefined transformation steps, like refinement or decomposition steps, can also be made. The correctness of such transformation steps has to be checked (automatically) using verification tools.

- *Low-level synthesis and implementation.* During this phase the entities and channels specified in the architecture model are actually implemented. The implementation

can contain a mixture of software and hardware.

In [Ver92] a number of useful and interesting functionality-preserving transformations are described. Our research began as an investigation of how to formalize these transformations and how to prove them correct. We soon came to the conclusion that a formal semantics of the object model was indispensible and therefore we started to define the model formally. However, the object model turned out to be very complex, mainly due to a number of very high level (communication) constructs, and it seemed not possible to define a usable formal semantics. Therefore we decided to design a new language meant as a formalization of the object model and based upon the following basic concepts of this model:

- A model consists of a set of statically interconnected distributed objects (processes) which execute in parallel and which communicate through a static network of channels.
- The distributed objects communicate by exchanging messages which may contain parameters. The communication mechanism is one-way synchronous which means in particular that objects do not have to wait for replies.
- Distributed objects contain internal data in the form of traveling objects or data objects which are strictly private to the owning object.
- Message parameters refer to traveling objects or data objects which are private to the sending process. If a message is exchanged, the involved parameter objects are actually *copied* from the sending process to the receiving process.
- A model explicitly reflects a system architecture (or a system topology) and can be hierarchical.

A first official version of the designed language is defined in [Voe94]. This technical report describes an improved version which was achieved after a number of case studies and experiments with the language.

1.2 Language Concepts

In [Weg87] the following basic concepts of the object-oriented paradigm are distinguished:

- a unified notion of *objects*, which are composed from a set of "operations" and a "state". Objects are *encapsulated* in the sense that they interact with other objects according to a predefined interface.
- the concept of *classes*
- the use of *inheritance*

Another important concept is polymorfism. [Mey88] explains that polymorfism, "the ability to take several forms", is the key factor in producing reusable designs or specifications.

The design methodology and the object model mentioned in the previous section have a number of characteristics which are not directly covered by the object-oriented paradigm. Especially, transformation, (automatic) verification, parallelism, and hierarchy are topics which are better supported by (algebraic) process-oriented techniques. These algebraic techniques are especially applicable to systems in which communication between system elements is a major feature [Koo91]. Algebraic techniques enable formal (automatic) verification, simulation, and even transformation [Lan92] of specifications. Furthermore, architecture and hierarchy can be expressed in algebraic languages in an elegant way. One of the first algebraic theories dealing in a formal way with the communication behaviour of systems is CCS [Mil80, Mil89]. Other algebraic formalisms are CSP [Hoa85], LOTOS [EVD89], and ACP [Bae86]. CCS is based upon the following two concepts [Mil80]:

- *Observation.* Concurrent systems are described fully enough to determine what behaviour will be seen or experienced by an external observer. Two systems are indistinguishable if we cannot tell them apart without pulling them apart. *Observation equivalence* is based upon this notion.

- *Synchronized communication.* A concurrent system is built from independent agents (processes) which communicate synchronously. *Parallel composition* is used to compose two independent agents, allowing them to communicate.

To support reusable design in a flexible way, POOSL builds upon the concepts of the object-oriented paradigm. On the other hand, to support verification, simulation, and transformation, the language also builds on the concepts of CCS. These process formalisms provide a number of elegant language constructions to express hierarchy, architecture, topology, and parallelism.

1.3 Processes versus Data

Like most, if not all, (practically applicable) specification languages, formalisms, and techniques, POOSL distinguishes *processes* from *data*. A specification in POOSL consists of a fixed number of statically interconnected distributed processes which are able to execute in parallel. Processes, or process objects, are connected to a fixed network of channels, through which they can communicate by sending messages. These messages may carry parameters in the form of data objects. These data objects are also used to model private data as well as internal computations of process objects. Data objects are essentially *sequential* in nature, which means that within one process object at any time only one data object can be active. The data part of POOSL is based upon (restricted) versions of POOL [AR89] and Smalltalk [GR89]. Data objects and their declarations are described in detail in Chapter 2. Chapters 3 and 4 deal with the explanation of process objects and

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Voeten, J.P.M.

POOSL: an object-oriented specification language for the analysis and design of hardware/software systems / by J.P.M. Voeten. - Eindhoven : Eindhoven University of Technology, Faculty of Electrical Engineering. - Fig. - (EUT report, ISSN 0167-9708 ; 95-E-290)
With ref.
ISBN 90-6144-290-7
NUGI 832
Subject headings: object-oriented methods / specification languages / formal specification.

specifications of systems of process objects. This process part of POOSL is mainly based on CCS [Mil80].

The strict separation between statically interconnected process objects and dynamic (traveling) data objects makes it possible to specify systems in an abstract and natural way. One can define the abstract (communication) behaviour of process objects or systems of interconnected process objects, without being concerned about (internal) data of process objects or data which has to be transported over the communication channels. The separation also opens the way to formal (automatic) verification and transformation of static system structure.

1.4 Related Work

The idea of combining object-oriented concepts with the concepts of process-oriented formalisms is not new.

In [Bla90] and [CRS90], for example, it is shown how LOTOS can be used as an object-oriented specification language. [MC94, MC93] describes the ROOA (Rigorous Object-Oriented Analysis) method which combines object-oriented analysis methods and formal description techniques (LOTOS) to produce a formal object-oriented analysis model that acts as the requirement specification of a system. [Cus88] demonstrates that it is possible to use a process algebra (CSP) in a natural and integrated way within the context of object-oriented specification of distributed systems. Other, more fundamental and theoretical, approaches towards integrating object-oriented and process-oriented concepts are given in [HT91] and [Nie91].

The great advantage of using algebraic process-oriented techniques for object-oriented specification is that they enable the description of *what* a system should do, without giving the solution of, or even details about, its implementation. Further, process-oriented techniques allow for a strict formal reasoning about objects, relations between objects, and other object-oriented concepts.

However, using algebraic process languages for object-oriented specification also has a number of disadvantages. First of all, it seems difficult to interpret *all* object-oriented concepts in existing process-oriented languages. Especially the concept of inheritance (*strict* as well as *non-strict* inheritance) is hard to capture [Bla90, CRS90, Cus88].

A second problem is caused by algebraic data typing languages, which are usually part of (practically applicable) algebraic process languages. It seems hard to interpret these languages according to the object-oriented paradigm and this makes it difficult to interpret process-oriented languages in their full strength [CRS90, Cus88].

Two other, more practical, problems are also caused by algebraic data languages. In an attempt to prohibit all implementation biases from the specification, the algebraic (data) specifications become difficult to understand. In practice, most designers find it very hard to define abstract data types in algebraic data languages [Nar87]. Further, because of their highly abstract nature, abstract data types do in general not allow direct and efficient implementation [MM89].

On the opposite side of formal description techniques we find (parallel) object-oriented programming languages like POOL [Ame87, AR89], PROCOL [BL89], DROL [TT92], Act1 [Lie87], and ABCL/1 [YBS86]. Although these languages can in principle be used for the specification of software or hardware systems, they are not really suited for our purposes. This is mainly due to the fact that none of these languages support all characteristics of the design methodology and the object model mentioned in Section 1.1. Especially system architecture, topology, and hierarchy are topics which are not explicitly expressible in any current object-oriented language. This is caused by the traditional conception that all objects should, in some sense, be equivalent. In our view there exists a kind of dualism, which is why we chose to distinguish data objects from process objects.

Our approach aims at developing a new practical language which combines the advantages of object orientation as well as those of process orientation. Because in our view one of the most important aspects of specifications is readability and understandability, we have chosen to base the data part of POOSL upon constructs of object-oriented programming languages and not upon any algebraic data typing language. Concepts of classes, inheritance and polymorfism are also taken from traditional object-oriented languages. Of course, the descriptive nature of these languages may result in overspecified specifications. However, Meyer [Mey88] shows that object-oriented program constructs allow the definition of very abstract data entities which, in a sense, come very close to algebraic specifications.

The process part of POOSL is strongly based upon CCS. This languages offer elegant and intuitively clear constructs to express parallelism, communication, synchronization, and system structure.

Chapter 2

Data Objects

2.1 Informal Explanation

Data objects or traveling objects in POOSL are much alike objects in *sequential* object-oriented programming languages such as Smalltalk [GR89], C++ [Str92], Eiffel [Mey88], and SPOOL [AB90]. A data object consists of some private data and has the ability to act on this data. Data is stored in *instance variables*, which contain (references to) other objects or to the object which owns the variables. The variables of an object cannot be accessed directly by any other object. They can only be read and changed by the object itself.

Objects can interact by sending *messages* to each other. A message consists of a *message name*, also called a message selector, and zero or more parameters. A message can be seen as a request to carry out one of the objects' *services*. An object explicitly states to which object it wants to send a message. When an object sends a message, its activities are suspended until the result of the message arrives. An object that receives a message will execute a corresponding so-called *method*. A method implements one of the object's services. It can access all instance variables of its corresponding object. In addition, it may have local variables of its own. The result of a method execution is returned to the sender.

Objects are grouped into *classes*. A class describes a set of objects which all have the same functionality. The individual objects in a class are called *instances*. The instance variables and methods, which are the same for all instances, are specified within a *class definition*.

Future versions of POOSL should support some form of inheritance. The precise form, however, has not been decided yet (see also Section 5.5). For now we will assume that the language incorporates the liberal inheritance scheme of Smalltalk. In Smalltalk a class can have a number of *subclasses*. The instances of such a subclass inherit all instance variables and methods of the corresponding *superclass*. Next to these variables and methods they

can also have additional variables, additional methods or redefined (overridden) methods. Within the class definition of the subclass it is specified how the subclass' instances differ from the instances of its superclass. Every class can have at most one superclass, which implies that multiple inheritance is not supported.

POOSL has four predefined classes of commonly used data types, namely *Boolean*, *Integer*, *Real*, and *Char*. Instances of these predefined classes are called *standard* objects. The set of messages of these objects correspond to the usual operations of the object's data type. Besides these objects a special standard object, named *nil*, exists. This object has no methods and an error occurs when a message is sent to it.

2.2 Formal Syntax

In this section an (abstract) syntax of the language of data objects is given. The syntax resembles the syntax of Smalltalk defined in [GR89]. We assume that the following sets of syntactic elements are given:

IVar	instance variables	$x, \cdots$
LVar	local variables	$u, v, w, \cdots$
CName	class names	$C, \cdots$
MName	method names	$m, \cdots$

First, we define the set *SObj* of standard objects with typical elements $\gamma, \cdots$. This set contains boolean objects, integer objects, real objects, char objects, and *nil*.

$$SObj = \mathbb{B} \cup \mathbb{Z} \cup \mathbb{R} \cup Char \cup \{nil\}$$

We define the set *Exp* of expressions, with typical elements $E, \cdots$, as follows:

$$\begin{array}{rcl} E & ::= & x \\ & | & u \\ & | & \text{new}(C) \\ & | & \text{self} \\ & | & E\ m(E_1, \cdots, E_n) \\ & | & \underline{\gamma} \\ & | & \underline{S}.E \end{array}$$

The first two expressions are instance variables, local variables, or parameters. The value of such a variable expression is (a reference to) the object currently stored in that variable. The next type of expression is the *new* expression. This expression indicates that a new object (of class C) has to be created. The expression yields the newly created object. Expression *self* refers to the object which is currently evaluating this expression. The sixth type of expression is a message-send expression. Here E refers to the object to which message m has to be sent and $E_1, \cdots, E_n$ are the parameters of the message. When a send

expression is evaluated, first the destination expression is evaluated, then the parameters are evaluated from left to right, and then the message is sent to the destination object. This object initializes its method parameters to the objects in the message and initializes its local method variables to *nil*. Next, the receiving object starts evaluating its method expression. The result of this evaluation is the result of the send expression which is returned to the sending object. Next, we have constant expressions $\underline{\gamma}, \cdots$, which refer to the above defined standard objects, such as integers and booleans. $\underline{\gamma}$ stands for the direct naming (textual representation) of standard object γ. An expression can be composed from a statement and another expression. When such a composite expression is evaluated, first the statement is executed and then the succeeding expression is evaluated. The value of this expression will be the value of the composite expression.

Next, we define the set *Stat* of statements. We let $S, \cdots$ range over *Stat* which is defined as

$$
\begin{array}{rcl}
S & ::= & E \\
& | & \text{skip} \\
& | & x \leftarrow E \\
& | & u \leftarrow E \\
& | & S_1 \cdot S_2 \\
& | & \text{if } E \text{ then } S_1 \text{ else } S_2 \text{ fi} \\
& | & \text{do } E \text{ then } S \text{ od}
\end{array}
$$

The first type of statement is an expression. Executing such a statement means that the expression is evaluated and result is discarded. The effect of the execution is the side-effect of the expression evaluation. Statement skip has no computational effect. Next, we have two assignment statements; the first to an instance variable and the second to a local variable. Upon execution of an assignment statement, the expression is evaluated and the result, a reference to an object, is assigned to the variable. The sequential composition (indicated with a dot $(\cdot)$ as in Smalltalk [GR89]), the if-statement, and the do-statement have their usual meaning.

Further, we define the set *Systems* with typical elements $Sys, \cdots$.

$$Sys \ ::= \ \langle CD_1 \cdots CD_n \rangle$$

A system *Sys* is a set of user-defined classes, comparable to a set of *system classes* in Smalltalk. A system is built from a number of class definitions.

The set *Classdef* of class definitions, with typical elements $CD, \cdots$, is defined as

$$
\begin{array}{rlll}
CD & ::= \text{class name} & C \\
& \quad \text{instance variable names} & x_1 \cdots x_n \\
& \quad \text{instance methods} & MD_1 \cdots MD_k
\end{array}
$$

|	class name	C_1
	super class	C_2
	instance variable names	$x_1 \cdots x_n$
	instance methods	$MD_1 \cdots MD_k$

Within a class definition the functionality of the classes' instances is specified. First, the name of the class is given. Then, optionally, the name of a possible superclass is specified. Next, the instance variables $x_1 \cdots x_n$ of the class are indicated. The last part of a class definition consists of a number of method definitions $MD_1 \cdots MD_k$.

The set of all method definitions is called *Methdef* and has typical elements $MD, \cdots$,

$$
\begin{array}{rcl}
MD & ::= & \mathbf{m}(u_1, \cdots, u_n) \\
& & \mid\ v_1 \cdots v_m\ \mid \\
& & E \\
& \mid & \mathbf{m}(u_1, \cdots, u_n) \\
& & \text{primitive}
\end{array}
$$

Within a method definition the functionality of a certain message or method is described. A method definition starts with a method or message name **m** and zero or more parameters $u_1, \cdots, u_n$. Next, zero or more local variables $v_1 \cdots v_m$ are specified. A method definition ends with an expression E which is the body of the method. This expression is evaluated when the method is invoked. The result of this evaluation is returned to the message sender.

However, there exist methods for which the functionality cannot be expressed in terms of expressions. The functionality of these, often called primitive methods, is specified in the form of axioms in the semantics of the language. A primitive method definition only contains the parameters of the method and a keyword which indicates that the method is primitive. A typical example of a primitive method is a *deepCopy* method which is used to create a complete copy of some object.

2.3 Example: An Unbounded FIFO Buffer

In this section we will give an example of a description of an unbounded FIFO buffer in POOSL. To ease the readability we will use the following syntactical conventions:

- statements of the form if E then S else skip fi are abbreviated to if E then S fi
- method calls or method headers of the form E $m()$ are abbreviated to m
- empty local variable declarations in method definitions are left out
- pairs of brackets around the parameters of method calls of standard objects are often left away. So we will write $3 + 2$ to mean $3 + (2)$ and $5 > 10$ instead of $5 > (10)$

First we define a class *FIFOLink*. A FIFO link is composed from an actual FIFO element and two references to other FIFO links. The class definition of class *FIFOLink* we will give is specified as:

class name *FIFOLink*
instance variable names *element*
previousLink
nextLink
instance methods

setElement(*anElement*)
element ← *anElement*·
self

setNextLink(*aLink*)
nextLink ← *aLink*·
self

setPreviousLink(*aLink*)
previousLink ← *aLink*·
self

element
element

nextLink
nextLink

previousLink
previousLink

By means of class *FIFOLink* we are able to declare class *FIFOBuffer*. A FIFO buffer has a depth, a reference to a FIFO link which represents the bottom of the buffer and a reference to another FIFO link which represents the head of the buffer. We assume that there exists a class *Object* which defines (primitive) method *error*. Class *FIFOBuffer* will be a subclass of *Object* which means that it also recognizes *error* messages. The class definition of *FIFOBuffer* is as follows:

class name *FIFOBuffer*
superclass *Object*
instance variable names *fifoDepth*
firstLink
lastLink
instance methods

clear
fifoDepth ← 0·
firstLink ← *nil*·
lastLink ← *nil*·
self

```
read
  | aLink |
  if fifoDepth = 0 then self error fi ·
  aLink ← lastLink·
  if fifoDepth > 1 then lastLink ← lastLink previousLink·
                                   lastLink setNextLink(nil)
                   else firstLink ← nil·
                        lastLink ← nil
  fi ·
  fifoDepth ← fifoDepth − 1·
  alink element


write(anElement)
  | aLink |
  if fifoDepth = 0 then firstLink ← new(FIFOLink)
                                    setElement(anElement)·
                        lastLink ← firstLink
                   else aLink ← new(FIFOLink)
                                setElement(anElement)
                                setNextLink(firstLink)·
                        firstLink setPreviousLink(aLink)·
                        firstLink ← aLink
  fi ·
  self


isEmpty
  fifoDepth = 0
```

If we call the class definitions of classes *FIFOLink*, *FIFOBuffer* and *Object* respectively $CD_{FIFOLink}$, $CD_{FIFOBuffer}$ and CD_{Object}, then we can define a system *Sys* of classes

$$Sys = \langle CD_{FIFOLink}\, CD_{FIFOBuffer}\, CD_{Object} \rangle$$

Chapter 3

Process Objects

This chapter describes the process-oriented part of POOSL. This part is based upon the language of data objects described in the previous chapter.

3.1 Informal Explanation

A specification in POOSL consists of a fixed set of *statically interconnected distributed processes* (a definition is given in [Weg87]). Processes or *process objects* are connected by statically defined *channels* through which they can communicate by exchanging messages. The communication mechanism we use is based upon the synchronous (rendez-vous) pairwise message-passing mechanism of CCS. It resembles the one-way synchronous message-passing mechanism of PROCOL [BL89].

When a process wants to send a message it explicitly states to which channel this message has to be sent. It also explicitly states when and from which channel it wants to receive a message. Immediately after a message has been received, the sending process resumes its activities (it does not have to wait for a result). If a process receives a message, it does *not* execute a method like in traditional object-oriented languages. Also, a possible expected result is *not* automatically returned to the sender. If a result of the message is expected, it has to be transmitted by means of another rendez-vous.

Process objects can call one of their methods. Such a call is either a call of a *terminating* method or a call of a *non-terminating* one. Terminating methods can be compared to procedures of imperative programming languages such as C or Pascal. Non-terminating methods are used to support tail recursion. Tail recursion has proven to be a very useful construct for the specification of communicating systems and is incorporated in all process oriented languages. Non-terminating methods of a process-object can be considered as *abstract* states of the object, and can be compared with agent identifiers in CCS.

Processes contain internal data in the form of data objects which are stored in instance

variables. Data objects are private to the owning process, i.e., process objects have no shared variables or shared data. A process can interact with its data objects by sending messages to them. When a process sends a message to one of its data objects, its activitities are suspended until the result of the message arrives. Data objects themselves cannot send messages (except for replies) to a process object.

When two processes communicate, a message and a (possibly empty) set of parameters is passed from one process to another. The parameters refer to objects which are private to the sending process. Because processes do not have any data in common, it does not suffice just to pass a set of references to the data objects, as in traditional object-oriented languages. Instead, the objects themselves have to be passed. This means that a new set of objects has to be created within the environment of the receiving process. These objects are (deep) copies of the data objects involved in the rendez-vous.

Process objects are grouped in classes, just as data objects. We distinguish two kinds of process classes: *basic* classes and *composite* classes. Basic process classes are defined in a similar manner as classes of data objects. Composite classes, on the other hand, are composed from other (basic or composite) classes, by means of *parallel composition*, *channel renaming*, and/or *channel hiding*. These combinators are based upon similar combinators originally used in CCS [Mil80]. Subclass relations between such composite process classes can not (yet) be defined. Instances of basic classes will be called basic processes (or basic process objects) and instances of composite classes will be called composite processes (or composite process objects). Future versions of POOSL should support some form of inheritance among process classes (see also Section 5.6). The precise form, however, is not yet decided on.

3.2 Formal Syntax

This section describes the formal abstract syntax of POOSL. It is based on the language of data objects of the previous chapter. We assume that the following sets of syntactic elements are given:

$Chan$	communication channels	$ch, \cdots$
Var	$Ivar \cup LVar$	$p, \cdots$

We define the set $Stat^p$ of process or parallel statements. These statements are used to specify the behaviour of process objects.

$$
\begin{aligned}
S^p \ ::= \ & S \\
\mid \ & ch!m(E_1, \cdots, E_n) \\
\mid \ & ch?m(p_1, \cdots, p_m) \\
\mid \ & m(E_1, \cdots, E_n \ ; \ p_1, \cdots, p_m) \\
\mid \ & m(E_1, \cdots, E_n) \\
\mid \ & S_1^p \cdot S_2^p \\
\mid \ & \text{sel } G_1 \text{ or } \cdots \text{ or } G_n \text{ les}
\end{aligned}
$$

The first type of statement is a statement defined in the language of data objects of the previous chapter. These statements are used to model internal data computations of a process.

The next two statements are the message-send and message-receive statements. A message-send statement $ch!m(E_1, \cdots, E_n)$ indicates that a process is willing to send message m together with parameters $E_1, \cdots, E_n$, on channel ch. Before the message is offered to the channel, first the parameters $E_1, \cdots, E_n$, which are data expressions, are evaluated from left to right. The actual message transfer can only happen if some other process is executing a message-receive statement $ch?m(p_1, \cdots, p_m)$ (or a guarded command which contains such a statement). Conversely, a message-receive statement can only be executed if another process executes a message-send statement. After a process has sent a message, it can immediately continue with its activities; it does not have to wait for an answer to arrive. Upon reception of a message, deep copies of the message parameters are bound to the input parameters $p_1, \cdots, p_m$ of the message-receive statement of the receiving process.

The fourth statement is a method call. By means of such a method call statement a process object can call its methods. A method call $m(E_1, \cdots, E_n \ ; \ p_1, \cdots, p_m)$ is executed in the following way: First, expressions $E_1, \cdots, E_n$ are evaluated from left to right. Next, the values of these expressions are bound to the input parameters of the method m and the local variables are initialized to *nil*. Then the method body is executed. After this execution terminates, the exit parameters of the method are bound to variables $p_1, \cdots, p_n$.

Next, we have another method call statement. $m(E_1, \cdots, E_n)$ is executed in a similar way as $m(E_1, \cdots, E_n \ ; \ p_1, \cdots, p_m)$. expressions $E_1, \cdots, E_n$ are evaluated from left to right. The body execution, however, does not (necessarily) terminate and the execution control is never returned to the point before the call. Non-terminating method calls in POOSL are used to support tail recursion. Note that these method calls are completely different from procedure calls in imperative programming languages.

The sixth sort of statement is sequential composition, which is indicated with a dot ($\cdot$) as in Smalltalk. Sequential composition has its usual meaning.

The last kind of statement is a select statement. A select statement indicates that a process can choose between a number of alternative statements, called guarded commands. The

set *GCom* of guarded commands with typical elements $G, \cdots$ is defined as

$$\begin{array}{rcl} G & ::= & E;\ \ ch?m(p_1, \cdots, p_m)\,\text{then}\,S^p \\ & | & E;\ \ ch!m(E_1, \cdots, E_n)\,\text{then}\,S^p \\ & | & E\,\text{then}\,S^p \end{array}$$

A guarded command is composed of an expression E, which is called a guard, and a rest statement. Upon execution of a select statement sel G_1 or $\cdots$ or G_n les, first all sub expressions (guards and possibly parameter expressions) of the statement are evaluated from left to right. All guards have to result in an object of class *Boolean*. The guarded commands from which the guard result in *false* are discarded (they do not play a role in the further execution of the statement). Now, zero or more of the following cases apply:

- One of the remaining guarded commands contains a message-receive statement. In that case the message can be received only if another process is trying to send the same message. If the message is received it is handled in the same way as in the case of a normal message receive statement. Next, the statement behind the keyword *then* is executed and the select statement is terminated.

- One of the remaining guarded commands contains a message-send statement. In this case the message of this statement can only be sent if some other process is willing to receive the message. After the message has been sent, the statement behind *then* is executed and the select statement is terminated.

- One of the remaining guarded commands is of the form $E\,\text{then}\,S^p$. In this case statement S^p can be executed, after which the select statement terminates.

- There are no remaining guarded commands. This means that all guards were evaluated to *false* and that no guarded commands can be executed. This implies that the process which is executing the select statement will deadlock.

If more than one of the guarded commands can be executed, then a non-deterministic choice between the different alternatives is made. If no guarded command can be executed, the process which is executing the select statement is suspended until one or more guarded commands become executable.

Next, we define a set $Systems^p$ of process systems, with typical elements $Sys^p, \cdots$.

$$Sys^p ::= \langle CD_1^p \cdots CD_k^p \rangle$$

A process system is a set of user-defined process classes. Such a system is built from a number of process class definitions. The set of all process class definitions $Classdef^p$ has typical elements $CD^p, \cdots$ and is defined as:

CD^p	::=	process class name	C
		instance variable names	$x_1 \cdots x_n$
		communication channels	$ch_1 \cdots ch_k$
		message interface	$cm_1 \cdots cm_l$
		initial method call	$m(E_1, \cdots, E_q)$
		instance methods	$MD_1^p \cdots MD_k^p$
	|	process class name	C
		communication channels	$ch_1 \cdots ch_k$
		message interface	$cm_1 \cdots cm_l$
		behaviour specification	$BSpec$

Within a process class definition the functionality of the instances of the class is specified. We distinguish two kinds of process classes, each with its own specification format. The first kind of class is called *basic* and the second kind is called *composite*.

A specification of a basic process class starts with the name of that class. Then a number of instance variables are specified. These variables model the private or internal data of each instance of the class. Next, all communication channels, through which the class' instance processes communicate with other processes, are specified. This channel specification is followed by a description of a message interface. A message interface is a list of channel-message combinations. A channel-message combination states that a process can send a certain message to a certain channel, or that a process can receive a specified message from a certain channel. The set CM of all channel-message combinations has typical elements $cm, \cdots$, and is defined as follows:

$$\begin{array}{rcl} cm & ::= & ch!m(p_1, \cdots, p_m) \\ & | & ch?m(p_1, \cdots, p_m) \end{array}$$

The first clause states that a process, at some point in time, can send message m, together with (formal) parameters $p_1, \cdots, p_m$, to channel ch. The second channel-message combination states that message m, with (formal) parameters $p_1, \cdots, p_m$ can be received from channel ch. A message interface is not a fundamental part of the behaviour specification of a process class. It mainly serves as an abstract description of the functionality of the instances of a process class.

The description of a message interface is followed by the specification of an initial method call of the form $m(E_1, \cdots, E_m)$. An instance of a process class always starts its activitities by first evaluating the parameter expressions $E_1, \cdots, E_m$ from left to right followed by calling its initial (non-terminating) method.

The last part of a basic process class definition consists of a number of method definitions. A method definition specifies the behaviour of its corresponding method. The set $Methdef^p$ of all method definitions, with typical elements $MD^p, \cdots$, is defined as

$$
\begin{array}{rcl}
MD^p & ::= & \mathbf{m}(u_1, \cdots, u_m)\ \mathbf{exitWith}(v_1, \cdots, v_n) \\
& & \quad \mid\ w_1 \cdots w_k\ \mid S^p \\
& \mid & \mathbf{m}(u_1, \cdots, u_m)\ \mathbf{noExit} \\
& & \quad \mid\ w_1 \cdots w_k\ \mid S^p
\end{array}
$$

Every method is either terminating or non-terminating. Terminating methods are invoked though a $m(E_1, \cdots, E_n\ ;\ p_1, \cdots, p_m)$ statement. A method definition of a terminating method contains a header with name m, the input parameters $u_1, \cdots, u_m$, and the exit variables $v_1, \cdots, v_n$ of the method. The keyword **exitWith** is used to express that the method terminates. The method header is followed by a declaration $\mid\ w_1 \cdots w_k\ \mid$ of local variables. Then the message body, which is a statement S^p, is defined.

Non-terminating methods are used to describe tail recursion. Such a method is invoked if a process which owns the method calls it by means of a statement of the form $m(E_1, \cdots, E_n)$. A non-terminating method definition has almost the same format as a definition of a terminating method. The only difference is that the former does not terminate and therefore have no exit variables, as is indicated by the keyword **noExit**.

The second kind of process classes are called *composite* process classes. A composite process class is built from other classes, which can be either basic or composite themselves. A class definition of a composite class consists of a process class name, communication channels, and a message interface. The behaviour of a process class is specified by means of a *behaviour specification*. The set *BSpecifications* of all process specifications has typical elements *BSpec* and is defined as follows:

$$
\begin{array}{rcl}
BSpec & ::= & C \\
& \mid & BSpec_1 \parallel BSpec_2 \\
& \mid & BSpec \setminus L \\
& \mid & BSpec[f]
\end{array}
$$

Here $L \subseteq Chan$ denotes a set of channels. f is a so-called channel relabelling function: a function from $Chan$ to $Chan$, which respects channel types. The first sort of behaviour specifications are class names of process classes. A class name C expresses the behaviour of a single instance of class C.

The second kind of specifications $BSpec_1 \parallel BSpec_2$ expresses the parallel composition of specifications $BSpec_1$ and $BSpec_2$. Assume, for example, that the class definition of a class, say C, contains behaviour specification $C_1 \parallel C_2$. This specification expresses the behaviour of two process objects, one of class C_1 and the other of class C_2, which execute in parallel and which (perhaps) communicate through their common channels. The channel set of class C is the union of the channel sets of classes C_1 and C_2. An instance of class C can sent and receive any message which can be sent and received by instances of either C_1 or C_2. The parallel composition combinator is comparable to the composition combinator of CCS.

The third kind of behaviour specification is called channel hiding. A channel hiding $BSpec \setminus L$ expresses a specification *BSpec* from which the channels in L are made externally invisible. This means that other (external) processes cannot communicate through channels in L with processes contained in specification *BSpec*. Assume, for example, that the method definition of a class C contains a behaviour specification $(C_1 \parallel C_2) \setminus \{ch\}$. This means that, even though classes C_1 and C_2 may contain channel ch, class C may not. Channel ch may only be used for the communication between the processes of classes C_1 and C_2. The channel hiding constructor is similar to the restriction combinator of CCS.

The last sort of specification expresses a channel renaming. The channel relabelling $BSpec[f]$ denotes a specification *BSpec* from which the channels are relabelled as dictated by f. We shall often write $ch'_1/ch_1, \cdots, ch'_n/ch_n$ for the relabelling function f for which $f(ch_i) = ch'_i$ for $i = 1, \cdots, n$ and $f(ch) = ch$ otherwise. Channel renaming can be very useful if process objects of the same class are used within different process environments and have to communicate through different channels.

Chapter 4

System Specifications

We are now ready to define what we consider to be a specification of a system. A specification of a system of parallel process objects consists of three parts. The first part is a behaviour specification *BSpec* which expresses how the actual system is composed from process classes defined in Sys^p. The second part is a process system Sys^p which contains a set of user-defined process classes. The last part is a system *Sys* of user-defined classes of data objects. Formally we define the set of all system specifications *SSpecifications*, with typical elements $SSpec, \cdots$, as

$$SSpec ::= \langle BSpec, Sys^p, Sys \rangle$$

4.1 Example: A Simple Integer Unit

In this section we will specify the behaviour of a simple unit which can perform operations on integer numbers. The unit, shown in figure 4.1, has an input port *in* and an output port *out*.

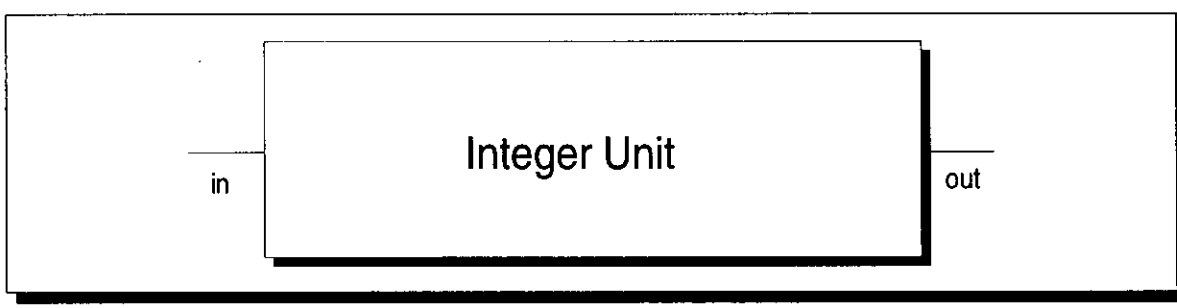

Figure 4.1: A simple integer unit

It can receive commands *add*, *substract*, *multiply*, and *power* from channel *in*, compute the result of the command, and send this result as a message to the output port. To simplify the syntactical formal notations, we will use the following conventions:

- guarded commands of the form *true*; $ch?m(p_1, \cdots, p_m)$ then S^p, or those of the form *true*; $ch!m(E_1, \cdots, E_n)$ then S^p may be abbreviated to $ch?m(u_1, \cdots, u_m)$ then S^p respectively $ch!m(E_1, \cdots, E_n)$ then S^p.
- "then skip" parts of a guarded command may be left out
- a message receive statement $ch?m()$ may be replaced by $ch?m$
- statements $ch!m()$ and $m()$ may be abbreviated to respectively $ch!m$ and m
- method definition headers of the form m() **exitWith**$(v_1, \cdots, v_n)$ or of the form m() **noExit** may be replaced by respectively m **exitWith**$(v_1, \cdots, v_n)$ and m **noExit**
- empty local variable declarations within method definitions are left out

The class definition of *IntegerUnit* can then be defined as

process class name	*IntegerUnit*
instance variable names	
communication channels	*in out*
message interface	$in?add(int_1, int_2)$ $in?substract(int_1, int_2)$
	$in?multiply(int_1, int_2)$ $in?power(int_1, int_2)$
	$out!result(anInt)$ $out!error$
initial method call	*start*()
instance methods	

start() noExit
sel
$in?add(int_1, int_2)$ then $out!result(int_1 + int_2)$
or
$in?substract(int_1, int_2)$ then $out!result(int_1 - int_2)$
or
$in?multiply(int_1, int_2)$ then $out!result(int_1 * int_2)$
or
$in?power(int_1, int_2)$ then **powerresult**$(int_1, int_2\ ;\ )$
les · **start**

Within non-terminating method **start** terminating method **powerresult** is called. This method is used to compute int_1 to the power of int_2 ($int_1{}^{int_2}$). If $int_2 < 0$, an error message is generated. Otherwise, the computed result is send to channel *out*. The method definition of **powerresult** is

powerresult(int_1, int_2) **exitWith**()
sel
($int_2 < 0$) then $out!error$
or
($int_2 \geq 0$) then
$out!result(int_1 \hat{} int_2)$
les

If we call the class definition of class *IntegerUnit* $CD_{IntegerUnit}$, we can define a system specification *SSpec*, describing a single integer unit, as follows:

$$SSpec = \langle IntegerUnit, \langle CD_{IntegerUnit} \rangle, \langle \rangle \rangle$$

Note that the system of user-defined classes of data objects in this specification is empty. Further, note that we have assumed that the standard objects of class *Integer* recognize +,−,∗, and ^ messages.

4.2 Example: An Unbounded Transmission Channel

By means of the class *FIFOBuffer* defined in Section 2.3, we are able to construct an unbounded transmission channel (see figure 4.2). The channel has an input port *in* and an output port *out*.

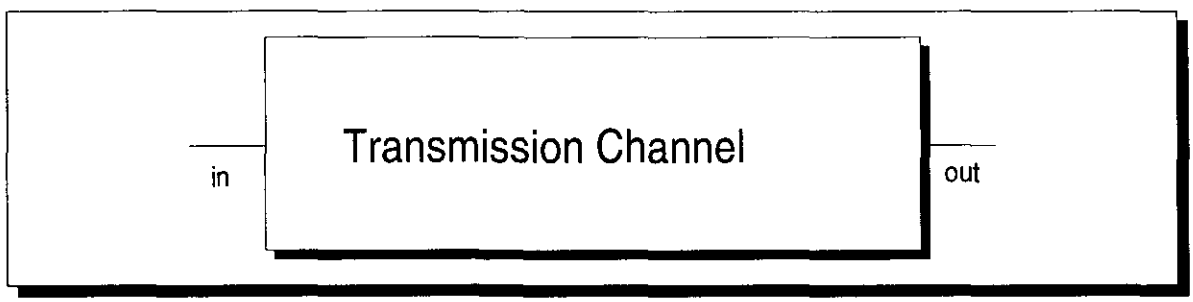

Figure 4.2: An unbounded transmission channel

Incoming *accept*-messages are received at port *in*, after which they are internally buffered in a FIFO fashion. The buffered messages are delivered at the output port in the form of *deliver* messages. A specification of the class *TransmissionChannel* is

process class name	*TransmissionChannel*
instance variable names	*buffer*
communication channels	*in out*
message interface	*in?accept(data) out!deliver(data)*
initial method call	*start*
instance methods	

```
start noExit
  buffer ← new(FIFOBuffer) clear·
  loop

loop noExit
  | data |
  sel
      in?accept(data) then buffer write(data) · loop
   or
      (buffer isEmpty not); out!deliver(buffer read) · loop
  les
```

4.3 Example: The Alternating-bit Protocol

A communication protocol is a discipline for transmission of messages from a source to a destination. An often-studied protocol is the alternating-bit protocol. A simple specification of this protocol is given in [Mil89]. In this section a slightly different version is specified. As shown in figure 4.3, the specification consists of six communicating processes: a Sender, a Receiver, two Timers, an Acknowledgement Channel and a Transmissions Channel.

If the sender receives a message (together with some data) from its input port, it sets a Timer, adds a bit to the received data, and sends the tagged message via the Transmission Channel to the Receiver. The value of the bit is the complement of the value assigned to the previous message. After the tagged message has been sent, the Sender waits for a message together with an acknowledge bit to arrive. The value of this bit should be equal to the value of the bit assigned to the previously sent message. If this is the case, a new message can be offered by the environment. If the bits are not equal, the acknowledge message is just absorbed. If in the mean time the Timer expires and gives a timeout, the tagged message is retransmitted.

The Receiver operates in a similar manner. If the Receiver is offered a tagged message it checks whether the tag bit equals to the expected bit. If this is so, the message, together with the data is offered to the environment, an acknowledge message is offered to the Acknowledgement Channel and the Timer is set. If the bit differs from the expected bit, the message is just ignored. When the Receiver gets a timeout, the previously sent acknowledge message is retransmitted. Both channels are unreliable: they can either duplicate or lose messages.

Before we start specifying the protocol, we first define a class *Bit*, which is used to model the tag bits.

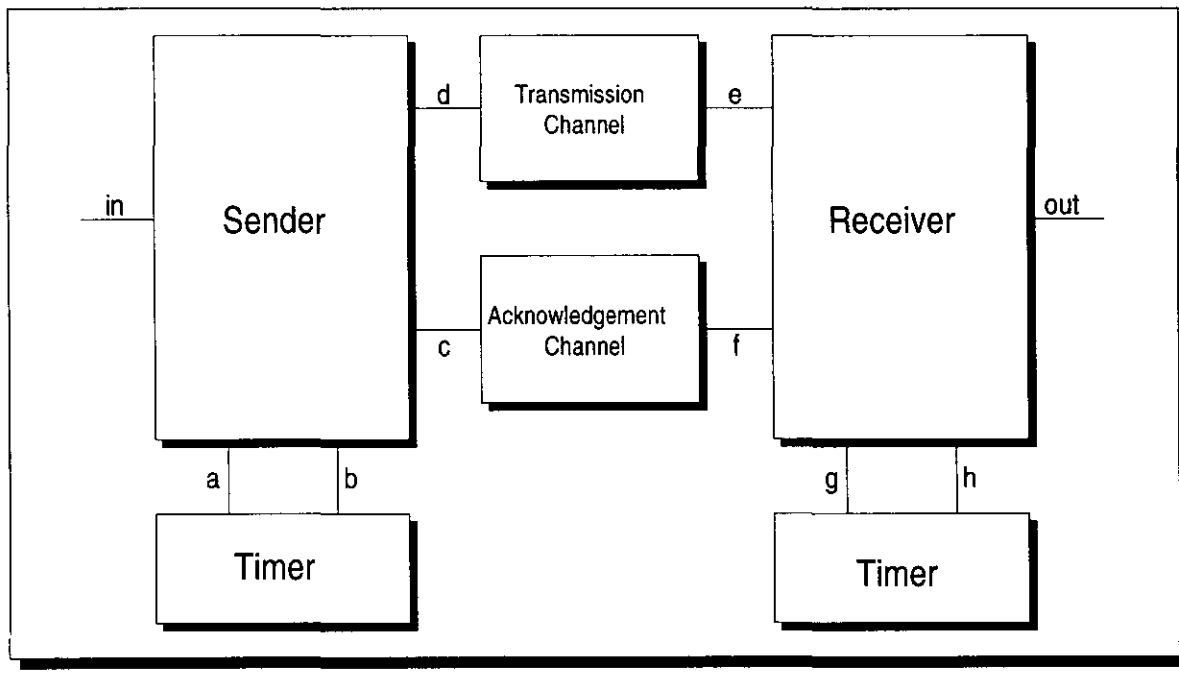

Figure 4.3: The Alternating-bit Protocol

class name *Bit*
instance variable names *bit*
instance methods

setToZero
bit $\leftarrow 0\cdot$
self

setToOne
bit $\leftarrow 1\cdot$
self

isZero
$(bit = 0)$

isOne
$(bit = 1)$

invert
bit $\leftarrow 1 - bit\cdot$
self

equals$(aBit)$
(self *isZero* $\wedge$ *aBit isZero*) $\vee$
(self *isOne* $\wedge$ *aBit isOne*)

inverts$(aBit)$
self *equals*$(aBit)$ *not*

The classes *Receiver* and *Sender* are specified as follows:

process class name *Sender*
instance variable names
communication channels *in a b c d*
message interface *in?receive(data) a!trigger b?timeout*
d!packet(data,aBit) c?ack(ackBit)
initial method call *start*
instance methods

start noExit
accept(new(*Bit*)*setToOne*)

accept(*aBit*) **noExit**
| *data* |
in?receive(*data*)·
send(*data*, *aBit*)

send(*data*, *aBit*) **noExit**
d!packet(*data*, *aBit*)·
a!trigger·
sending(*data*, *aBit*)

sending(*data*, *aBit*) **noExit**
| *ackBit* |
sel
b?timeout then *send*(*data*, *aBit*)
or
c?ack(*ackBit*) then
sel
aBit equals(*ackBit*) then *accept*(*aBit invert*)
or
(*aBit equals*(*ackBit*)) *not* then *sending*(*data*, *aBit*)
les
les

process class name *Receiver*
instance variable names
communication channels *out e f g h*
message interface *out!send(data) e?packet(data,aBit)*
f!ack(ackBit) g!trigger h?timeout
initial method call *start*
instance methods

start noExit
reply(new(*Bit*)*setToZero*)

deliver(*ackBit*, *data*) **noExit**
out!send(*data*)·
reply(*ackBit*)

reply(*ackBit*) **noExit**
| *data* |
f!ack(*ackBit*)·
g!trigger·
replying(*ackBit*)

replying(*ackBit*) **noExit**
| *data aBit* |
sel
h?*timeout* then *reply*(*aBit*)
or
e?*packet*(*data*, *aBit*) then
sel
aBit inverts(*ackBit*) then *deliver*(*ackBit invert*, *data*)
or
aBit inverts(*ackBit*) *not* then *replying*(*ackBit*)
les
les

Before we define the class *AcknowledgeChannel*, we first define a class *AckBuf*. An instance of *AckBuf* models an unreliable one-place buffer. We will model the Acknowledge Channel as a parallel composition of four unreliable one-place buffers.

process class name	*AckBuf*
instance variable names	*aBit*
communication channels	*in out*
message interface	*in?ack(aBit) out!ack(aBit)*
initial method call	*emptyBuf*
instance methods	

emptyBuf noExit
sel
in?*ack*(*aBit*) then *emptyBuf*
or
in?*ack*(*aBit*) then *fullBuf*
les

fullBuf noExit
sel
in?*ack*(*aBit*) then *emptyBuf*
or
in?*ack*(*aBit*) then *fullBuf*
or
out!*ack*(*aBit*) then *emptyBuf*
or
out!*ack*(*aBit*) then *fullBuf*
les

We are now able to specify class *AcknowledgeChannel*.

process class name	*AcknowledgeChannel*
instance variable names	
communication channels	*c f*
message interface	*c!ack(aBit) f?ack(aBit)*
behaviour specification	$(AckBuf[f/in, c_1/out] \parallel AckBuf[c_1/in, c_2/out] \parallel AckBuf[c_2/in, c_3/out] \parallel AckBuf[c_3/in, c/out]) \setminus \{c_1, c_2, c_3\}$

In a similar manner we can specify class *TransmissionChannel*. Note that both the transmission channel and the acknowledgement channel are bounded. Unbounded reliable or unreliable communication channels can be specified by means of the unbounded FIFO buffer of Section 2.3 just like the unbounded transmission channel of the previous section.

Finally, we specify the class *Timer*.

process class name *Timer*
instance variable names
communication channels *a b*
message interface *a?trigger b!timeout*
initial method call *timerNotSet*
instance methods

timerSet noExit
sel
a?trigger then *timerSet*
or
b!timeout then *timerNotSet*
les

timerNotSet noExit
a?trigger · *timerSet*

A behaviour specification *BSpec* of the Alternating-bit protocol can now be defined as

$$(Timer \parallel Sender \parallel AcknowledgeChannel \parallel TransmissionChannel \parallel Receiver \parallel Timer[g/a, h/b]) \setminus \{a, b, c, d, e, f, g, h\}$$

and a total system specification would be

$$\langle BSpec, \langle CD^p_{Timer}\, CD^p_{Sender}\, CD^p_{AcknowledgeChannel}\, CD^p_{TransmissionChannel}\, CD^p_{Receiver}\, CD^p_{Timer}\rangle, \langle CD_{Bit}\rangle\rangle$$

The specification of the Alternating-bit protocol shows how tail recursion can be used to model typical process-oriented systems in POOSL. These systems are very hard to model naturally in traditional object-oriented languages. Of course, POOSL is also suited to describe typical object-oriented systems.

Chapter 5

Final Remarks and Future Work

5.1 Abstract Specifications

In Sections 1.3 and 1.4 we mentioned that POOSL supports the possibility to specify systems in an abstract and natural way. By means of *message receive*, *message send*, and *select* statements one can define the abstract (communication) behaviour of process objects or systems of interconnected process objects, without being concerned about (internal) data of process objects or data which has to be transported over the communication channels. The main reason why abstract specifications are expressible is that POOSL strictly distinguishes *statically interconnected* process objects, whose communication behaviour can be statically determined, from data objects, whose communication behaviour is very dynamic and time dependent. Another reason is that POOSL supports tail recursion. Complex abstract communication behaviour is hard to express *naturally* in languages which do not support tail recursion. Especially state-machine-alike behaviour is difficult to model without the expressive power of tail recursion.

5.2 Semantics and Transformations

One of the main goals of our research is to formalize the functionality-preserving transformations mentioned in Section 1.1 and to prove them correct. The formalization of these transformations can easily be established by expressing them in POOSL. Correctness proofs of the transformations, however, can only be given if a formal semantics is available. A formal operational semantics of POOSL has been defined in [Voe]. Currently we are looking at the formalization and correctness proving of the transformations.

5.3 Automatic Verification

Verification is an activity which decides whether a specification meets certain properties. One of the most popular verification methods is *equivalence checking*, by means of which

it can be decided whether two specifications have the same observable behaviour. Other interesting methods are *preorder checking* and *model checking*. These verification methods can be used very effectively during the high-level architecture synthesis phase of the design methodology described in Section 1.1. By means of equivalence checking or preorder checking, for example, it can be decided whether a (potential) implementation satisfies its specification. Model checking can be used to verify whether a specification satisfies certain temporal properties such as deadlock freedom.

Currently, a number of tools which automate verification activities are available. Examples of such tools include CCStool2 [VV94], The Concurrency Workbench [CPS93], and Auto/Autograph [BRSV89, RS90]. Recently we have applied CCStool2 for the verification of the Alternating-bit protocol of Section 4.3. For such a verification, POOSL specifications have to be transformed to equivalent CCS descriptions. Abstract POOSL specifications, specifications without data, can be translated into CCS in a straightforward way. It is not yet clear how full POOSL descriptions can in general be transformed into CCS. This, and the general linking of POOSL with automatic verification tools, will be the subject of future research.

5.4 Messages and Methods

In most (parallel) object-oriented languages, messages are strongly coupled to their corresponding methods. If an object receives a message, a method is executed which takes care of that message. The data part of POOSL follows this convention. In the process part of the language, however, messages and methods are not coupled at all! After a number of case studies, during which we tried to model reactive objects, we found out that message reception often resulted in nothing more than copying of the data parameters of the message. In our first official version of POOSL [Voe94], message reception resulted in the call of a terminating method. In most cases, however, the only thing these methods had to achieve was to copy the input parameters to the output parameters. Further, the syntactic notation for message reception was conceptually unclear. Therefore, we decided to completely decouple message reception from method calling in the new version of POOSL.

5.5 Inheritance and Typing

The version of POOSL described in this report is a typeless language. For reasons of clarity and improved reliability, we believe that any (object-oriented) specification language should be

(i) *statically typed*: types of variables, parameters, and method results are explicitly defined in the specification

(ii) *strongly typed*: all type checking can be performed statically

(iii) *type-safe*: it can be statically checked that execution never causes "message not understood errors"

This would mean that we have to develop a type system for (the data part of) POOSL. However, developing appropriate type systems for object-oriented languages has shown to be difficult. The major difficulty involves the interaction and competition between *classes* and *inheritance* on one side and *types* and *subtyping* on the other. The construction of a suitable type system for POOSL requires more investigation and will be subject of future research. Since there exists a strong interdependence between type systems and inheritance schemes, we have also not yet decided on the latter. An appropriate inheritance scheme for (the data part of) POOSL will have to be developed simultaneously with the type system. For more information on type systems, subtyping, classes, and inheritance, we refer to [DT92, Hür94, Coo89, AL90, CHC90].

5.6 The Inheritance Anomaly

The inheritance anomaly refers to the serious difficulty in combining inheritance and concurrency in a simple and satisfactory way within a concurrent object-oriented language. The problem is closely connected with the need to impose *synchronization constraints* on the acceptance of a message by an object [Mes93]. In most object-oriented concurrent programming languages *synchronization code* is used to control the acceptance of messages by objects. It has been pointed out that synchronization code cannot be effectively inherited without non-trivial class redefinitions or violation of class encapsulation. An excellent analysis and survey of the inheritance anomaly together with a number of proposed solutions is presented in [MY93]. The most promising solutions of the anomaly tend to the application of very flexible and dynamic *synchronization schemes* (schemes for achieving object-wise synchronization using language primitives) (possibly) build upon *reflective* [YW88, WY88] language capabilities.

Although POOSL employs a synchronization scheme which is more flexible than that of some other object-oriented programming languages, it is not nearly flexible enough to allow complete elimination of the inheritance anomaly, independent of the future inheritance scheme. In our opinion one of the major problems with very flexible and dynamic synchronization schemes is that they clash with the verifiability and understandability requirements of specification languages, which is why we have chosen to adopt a more static scheme.

The severeness of the problem has caused a number of well-known concurrent object-oriented languages, such as POOL/T [Ame87], Act1 [Lie87] and ABCL/1 [YBS86], to give up supporting inheritance as a basic language feature. In POOSL only the *process part* can possibly suffer from the inheritance anomaly. This means that even if the anomaly causes too many problems, which is not known yet, the language would still support inheritance,

the prime language feature in sequential object-oriented languages, among classes of data objects.

5.7 Assertions

The Eiffel language is one of the few object-oriented programming languages that enables designers to define specification elements within Eiffel programs [Mey88]. Such a specification element, called an *assertion*, states *what* a certain element must do, independently of *how* it does it. Assertions in Eiffel create the possibility to define program entities which come very close to abstract data types, and they make Eiffel both a design and a programming language. Ideally, the assertion language should at least have the power of first-order predicate logic [AL90]. However, if the assertion language is so powerful, it is inherently impossible to automatically check (a priori by a compiler or at run-time) whether program entities meet their specifications. Eiffel, therefore, uses a somewhat restricted, yet practical, set of assertions. These assertions can be monitored at run-time to validate program correctness.

We find the approach taken by Eiffel very elegant, especially because it is practically applicable, easily understandable, and readable. We are therefore considering to follow the Eiffel approach by incorporating assertions into POOSL too.

5.8 Selective Message Reception

Assume we are dealing with a number of identical resource objects, sharing a common set of communication channels, and suppose that some process would like to claim one of the available resources, exchange a number of messages, and give the resource free. For this the process would have to be able to selectively send messages to one of the resources. However, since the resources can only select messages on the basis of channel names and message names, and because all resources are acquainted with the same channels and messages, it is impossible for the process to indicate that a certain message is meant for *one* specific resource.

However, constructions such as the above occur frequently, especially in object-oriented systems. The analysis and design method described in [Ver92], for example, recognizes this and uses a special kind of entity, called *(dynamic) multiple*, to model these constructions. The ROOA method, described in [MC94, MC93], uses *class templates* and *object generators* for similar purposes. Both methods use constructs for selective message reception, i.e., method reception on the basis of values of parameters (*addresses* respectively *object identifiers*) of messages.

To deal with the described problem, we have to increase the expressive power of POOSL.

This can be achieved by replacing the message receive statement $ch?m(p_1, \cdots, p_m)$ by a similar statement $ch?m(p_1, \cdots, p_m \mid E)$, with E denoting a boolean expression over parameters $p_1, \cdots, p_m$. The intended semantics is that a process is willing to receive message m with parameters $p_1, \cdots, p_m$ on channel ch only if expression E evaluates to true. We expect that the statement, which solves the described problem very elegantly, can be incorporated in (the formal semantics of) POOSL in a straightforward way.

Chapter 6

Conclusions

In this report we have presented an object-oriented language for the specification of complex hardware and software systems. The language builds upon a design methodology formulated in [Ver92]. This methodology demands a number of very specific requirements of the specification language. The most important requirements are:

- the language should be object-oriented
- it should support functionality-preserving transformations and (automatic) verification
- the language should support the explicit representation of system architecture, system topology, and hierarchy.

To meet these requirements, POOSL builds upon the concepts of the object-oriented paradigm as well as on the basic concepts of the algebraic process formalism CCS. POOSL explicitly distinguishes (statically interconnected distributed) process objects from data objects.

Process objects communicate through channels using a synchronous message-passing mechanism based on CCS. Further, by means of *parallel composition*, *restriction*, and *channel renaming* constructions, process objects can be composed to form *composite* processes. These constructions, originating from CCS, are also used to make *hierarchical* system specifications which reflect system *architecture* and system *topology*.

Data or traveling objects are used to model private data of process objects as well as data which is exchanged between (different) process objects. Data objects are much like objects in traditional object-oriented programming languages, such as POOL and Smalltalk. Also concepts of classes, inheritance, and polymorfism are taken from these languages.

The strict distinction between process and data objects opens the way to formal (automatic) verification and transformation. This, together with possibility to describe tail recursion, also creates a way to specify systems in an abstract and natural way.

The version of POOSL described in this report is only a *basic* software/hardware specification language. To complete the language, it has to be extended with a number of language constructions, such as inheritance, static typing, assertions, and selective message reception. At this moment the precise form of these extensions is unknown and requires more research.

References

[AB90] America, P.H.M. and F.S. de Boer.
A proof theory for a sequential version of POOL.
Faculty of Mathematics and Computer Science, Eindhoven University of Technology, October, 1990.
Report Series: Computing Science Notes, nr. 90/12.

[AL90] America, P.H.M. and F. van der Linden.
A parallel object-oriented language with inheritance and subtyping.
In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP'90), Ottawa, Canada, October 21–25, 1990. Ed. by N. Megrowitz. New York : ACM, 1990. P. 161–168.

[Ame87] America, P.H.M.
Synchronizing actions.
In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP'87), Paris, France, June 15–17, 1987. Ed. by J. Bezivin et al. Berlin : Springer, 1987 (Lecture Notes in Computer Science, Vol. 276). P. 234–242.

[AR89] America, P.H.M. and J.J.M.M. Rutten.
A parallel object-oriented language: Design and semantic foundations.
Amsterdam : Vrije Universiteit, 1989.
Ph.D. thesis.

[Bae86] Baeten, J.C.M.
Procesalgebra: Een formalisme voor parallel, communicerende processen.
Deventer : Kluwer, 1986.

[BL89] Bos, J. van den and C. Laffra.
PROCOL: A parallel object language with protocols.
In: Proceedings of the 1989 conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'89), New Orleans, Louisiana, October 1–6, 1989. Ed. by N. Meyrowitz. New York : ACM, 1989. P. 95–102.

[Bla90] Black, B.
Objects and LOTOS.
In: Proceedings of the second international conference on formal description techniques for distributed systems and communication protocols (FORTE'89), Vancouver, Canada, December 5–8, 1989. Ed. by Son T. Vuong. Amsterdam : North-Holland, 1990. P. 285–297.

[BRSV89] Boudol, G. and V. Roy, R. de Simone, D. Vergamini.
Process calculi, from theory to practice: Verification tools.
In: Proceedings of the International workshop on automatic verification methods for finite state systems, Grenoble, France, June 12–14, 1989. Ed. by J. Sifakis. Berlin : Springer, 1990. P. 1–10.

[CHC90] Cook, W. and W.L. Hill, P.S. Canning.
Inheritance is not subtyping.
In: Conference record of the 17th annual symposium on principles of programming languages, San Francisco, USA, January 17–19, 1990. New York : ACM, 1990. P. 125–135.

[Coo89] Cook, W.
A proposal for making Eiffel type-safe.
In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP'89), Nottingham, England, July 10–14, 1989. Ed. by S. Cook. Cambride : Cambridge University Press, 1989. P. 57–70.

[CPS93] Cleaveland, R. and J. Parrow, B. Steffen.
The Concurreny Workbench: A semantics-based tool for the verification of concurrent systems.
ACM Transactions on programming languages and systems, Vol. 15(1993), no. 1, p. 36–72.

[CRS90] Cusack, E. and S. Rudkin, C.Smith.
An object-oriented interpretation of LOTOS.
In: Proceedings of the second International conference on formal description techniques for distributed systems and communication protocols (FORTE'89), Vancouver, Canada, December 5–8, 1989. Ed. by Son T. Vuong. Amsterdam : North-Holland, 1990. P. 265–284.

[Cus88] Cusack, E.
Formal object-oriented specification of distributed systems.
In: Proceedings of the BCS-FACS workshop on specification and verification of concurrent systems, Stirling, Scotland, July 6–8, 1988. Ed. by C. Rattray. Berlin : Springer, 1990. P. 71–83.

[DT92] Dodani, M. and C. Tsai.
ACTS: A type system for object-oriented programming based on abstract and concrete classes.
In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP'92), Utrecht, The Netherlands, June 29–July 3, 1992. Ed. by O. Lehrmann Madsen. Berlin : Springer, 1992 (Lecture Notes in Computer Science, Vol. 615). P. 309–324.

[EVD89] Eijk, P.H.J. van and C. Vissers, M. Diaz.
The formal description technique LOTOS.
Amsterdam : North-Holland, 1989.

[GR89] Goldberg, A. and D. Robson.
Smalltalk-80: The language.
Reading, Massachusetts : Addison-Wesley, 1989.

[Hoa85] Hoare, C.A.R.
Communicating sequential processes.
Englewood Cliffs, New Jersey : Prentice-Hall, 1985.

[HT91] Honda, K. and M. Tokoro.
An object calculus for asynchronous communication.
In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP'91), Geneva, Switzerland, July 15–19, 1991. Ed. by P. America. Berlin : Springer, 1991 (Lecture Notes in Computer Science, Vol. 512). P. 133–147.

[Hür94] Hürsch, W.L.
Should superclasses be abstract?
In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP'94), Bologna, Italy, July 4–8, 1994. Ed. by M. Tokoro and R. Pareschi. Berlin : Springer, 1994 (Lecture Notes in Computer Science, Vol. 821). P. 12–31.

[Koo91] Koomen, C.J.
The design of communicating systems: A system engineering approach.
Dordrecht : Kluwer, 1991.

[Lan92] Langerak, R.
Transformations and semantics for LOTOS.
Ph.D. thesis, University of Twente, 1992.

[Lie87] Liebermann, H.
Concurrent object-oriented programming in Act 1.
In: Object-oriented concurrent programming. Ed. by A. Yonezawa and M. Tokoro. Cambridge : Mit Press, 1987. P. 9–36.

[MC93] Moreira, A.M.D. and R.G. Clark.
Rigorous object-oriented analysis.
University of Strirling, Computing Science Department, 1993.
Technical Report, nr. TR 109.

[MC94] Moreira, A.M.D. and R.G. Clark.
Combining object-oriented analysis and formal description techniques.
In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP'94), Bologna, Italy, July 4–8, 1994. Ed. by M. Tokoro and R. Pareschi. Berlin : Springer, 1994 (Lecture Notes in Computer Science, Vol. 821). P. 345–364.

[Mes93] Meseguer, J.
Solving the inheritance anomaly in concurrent object-oriented programming.
In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP'93), Kaiserslautern, Germany, July 26–30, 1993. Ed. by O. Nierstrasz. Berlin : Springer, 1993 (Lecture Notes in Computer Science, Vol. 707). P. 220-246.

[Mey88] Meyer, B.
Object-oriented software construction.
New Jersey, Englewood Cliffs : Prentice-Hall, 1988.

[Mil80] Milner, R.
A Calculus of communicating systems.
Berlin : Springer, 1980.
(Lecture Notes in Computer Science, Vol. 92).

[Mil89] Milner, R.
Communication and concurrency.
London : Prentice Hall, 1989.

[MM89] Mañas J. and T. De Miguel.
From LOTOS to C.
In: Proceedings of the first international conference on formal description techniques for distributed systems and communication protocols (FORTE'88), Stirling, Scotland, September 6–9, 1988. Ed. by K. Turner. Amsterdam : North-Holland, 1989. P. 79–84.

[MY93] Matsuoka, S. and A. Yonezawa.
Analysis of inheritance anomaly in object-oriented concurrent programming languages.
In: Research directions in concurrent object-oriented programming. Ed. by G. Agha and P. Wegner, A. Yonezawa. London : MIT Press, 1993. P. 107–150.

[Nar87] Narfelt, K.H.
SYSDAX: An object-oriented design methodology based on SDL.
In: Proceedings of SDL'87: State of the art and future trends, The Hague, The Netherlands, April 3–10, 1987. Ed. by R. Saracco and P. Tilanus. Amsterdam : North-Holland, 1987. P. 247–254.

[Nie91] Nierstrasz, O.
Towards an object calculus.
In: Proceedings of the ECOOP'91 workshop on object-based concurrent programming, Geneva, Switzerland, July 15–16, 1992. Ed. by M. Tokoro and O. Nierstrasz, P. Wegner. Berlin : Springer, 1992 (Lecture Notes in Computer Science, Vol. 612). P. 1–20.

[RS90] Roy, V. and R. de Simone.
Auto/Autograph.
In: Proceedings of the 2nd International conference on Computer-Aided Verification (CAV'90), New Brunswick, USA, June 18–21, 1990. Ed. by E. Clark and R. Kurshan. Berlin : Springer, 1990 (Lecture Notes in Computer Science, Vol. 531). P. 65–75.

[Str92] Stroustrup, B.
The C++ programming language.
Reading, Massachusetts : Addison-Wesley, 1992.

[TT92] Takashio, K. and M. Tokoro.
DROL: An object-oriented programming language for distributed real-time systems.
In: Proceedings of the 1992 conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'92), Vancouver, British Columbia, Canada, October 18–22, 1992. Ed. by A. Paepcke. New York : ACM Press, 1992 (ACM Sigplan Notices, Vol. 27/10). P. 276–294.

[Ver92] Verschueren, A.C.
An object-oriented modelling technique for analysis and design of complex (real-time) systems.
Ph.D. thesis, Eindhoven University of Technology, 1992.

[Voe] Voeten, J.P.M.
Semantics of POOSL: An object-oriented specification language for the analysis and design of hardware/software systems.
To be published as a technical report, Eindhoven University of Technology (1995).

[Voe94] Voeten, J.P.M.
POOSL, A parallel object-oriented specification language.
In: Proceedings of the eight workshop computer systems, Amsterdam, The Netherlands, March 25, 1994. Ed. by P. Hartel. Amsterdam : University of Amsterdam, 1994. Technical Report University of Amsterdam, Department of Computer Science, nr. CS-94-04. P. 25–45.

[VV94] Van Rangelrooij, A. and J.P.M. Voeten.
CCStool2: An expansion, minimization, and verification tool for finite state CCS descriptions.
Eindhoven : Eindhoven University of Technology, Faculty of Electrical Engineering, 1994.
EUT Report 94-E-284.

[Weg87] Wegner, P.
Dimensions of object-based language design.
In: Proceedings of the 1987 Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'87), Orlando, Florida, October 4–8, 1987. Ed. by N. Meyrowitz. New York : ACM Press, 1987 (ACM Sigplan Notices, Vol. 22/12). P. 168–182.

[WY88] Watanabe, T. and A. Yonezawa.
Reflection in an object-oriented concurrent language.
In: Proceedings of the 1988 conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'88), San Diego, California, September 25–30, 1988. Ed. by N. Meyrowitz. New York : ACM Press, 1988 (ACM Sigplan Notices, Vol. 23/11). P. 306–314.

[YBS86] Yonezawa, A. and J. Briot, E. Shibayama.
Object-oriented concurrent programming in ABCL/1.
In: Proceedings of the 1986 conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'86), Portland, Oregon, September 29–October 2, 1986. Ed. by N. Meyrowitz. New York : ACM Press, 1986 (ACM Sigplan Notices, Vol. 21/11). P. 258–268.

[YW88] Yonezawa, A. and T. Watanabe.
An introduction to object-based reflective concurrent computation.
In: Proceedings of the 1988 ACM SIGPLAN workshop on object-based concurrent programming, San Diego, September 26–27, 1988. Ed. by G. Agha and P. Wegner, A. Yonezawa. New York : ACM Press, 1988 (ACM Sigplan Notices, Vol. 24/04). P. 50–54.

Eindhoven University of Technology Research Reports ISSN 0167-9708
Coden: TEUEDE

Faculty of Electrical Engineering

(264) Freriks, L.W. and P.J.M. Cluitmans, M.J. van Gils
THE ADAPTIVE RESONANCE THEORY NETWORK: (Clustering-) behaviour in relation with brainstem auditory evoked potential patterns.
EUT Report 92-E-264. 1992. ISBN 90-6144-264-8

(265) Wellen, J.S. and F. Karouta, M.F.C. Schemmann, E. Smalbrugge, L.M.F. Kaufmann
MANUFACTURING AND CHARACTERIZATION OF GAAS/ALGAAS MULTIPLE QUANTUMWELL RIDGE WAVEGUIDE LASERS.
EUT Report 92-E-265. 1992. ISBN 90-6144-265-6

(266) Cluitmans, L.J.M.
USING GENETIC ALGORITHMS FOR SCHEDULING DATA FLOW GRAPHS.
EUT Report 92-E-266. 1992. ISBN 90-6144-266-4

(267) Józwiak, L. and A.P.H. van Dijk
A METHOD FOR GENERAL SIMULTANEOUS FULL DECOMPOSITION OF SEQUENTIAL MACHINES: Algorithms and implementation.
EUT Report 92-E-267. 1992. ISBN 90-6144-267-2

(268) Boom, H. van den and W. van Etten, W.H.C. de Krom, P. van Bennekom, F. Huijskens, L. Niessen, F. de Leijer
AN OPTICAL ASK AND FSK PHASE DIVERSITY TRANSMISSION SYSTEM.
EUT Report 92-E-268. 1992. ISBN 90-6144-268-0

(269) Putten, P.H.A. van der
MULTIDISCIPLINAIR SPECIFICEREN EN ONTWERPEN VAN MICROELEKTRONICA IN PRODUKTEN (in Dutch).
EUT Report 93-E-269. 1993. ISBN 90-6144-269-9

(270) Bloks, R.H.J.
PROGRIL: A language for the definition of protocol grammars.
EUT Report 93-E-270. 1993. ISBN 90-6144-270-2

(271) Bloks, R.H.J.
CODE GENERATION FOR THE ATTRIBUTE EVALUATOR OF THE PROTOCOL ENGINE GRAMMAR PROCESSOR UNIT.
EUT Report 93-E-271. 1993. ISBN 90-6144-271-0

(272) Yan, Keping and E.M. van Veldhuizen
FLUE GAS CLEANING BY PULSE CORONA STREAMER.
EUT Report 93-E-272. 1993. ISBN 90-6144-272-9

(273) Smolders, A.B.
FINITE STACKED MICROSTRIP ARRAYS WITH THICK SUBSTRATES.
EUT Report 93-E-273. 1993. ISBN 90-6144-273-7

(274) Bollen, M.H.J. and M.A. van Houten
ON INSULAR POWER SYSTEMS: Drawing up an inventory of phenomena and research possibilities.
EUT Report 93-E-274. 1993. ISBN 90-6144-274-5

(275) Deursen, A.P.J. van
ELECTROMAGNETIC COMPATIBILITY: Part 5. installation and mitigation guidelines, section 3, cabling and wiring.
EUT Report 93-E-275. 1993. ISBN 90-6144-275-3

(276) Bollen, M.H.J.
LITERATURE SEARCH FOR RELIABILITY DATA OF COMPONENTS IN ELECTRIC DISTRIBUTION NETWORKS.
EUT Report 93-E-276. 1993. ISBN 90-6144-276-1

Eindhoven University of Technology Research Reports ISSN 0167-9708
Coden: TEUEDE

Faculty of Electrical Engineering

(277) Weiland, Siep
A BEHAVIORAL APPROACH TO BALANCED REPRESENTATIONS OF DYNAMICAL SYSTEMS.
EUT Report 93-E-277. 1993. ISBN 90-6144-277-X

(278) Gorshkov, Yu.A. and V.I. Vladimirov
LINE REVERSAL GAS FLOW TEMPERATURE MEASUREMENTS: Evaluations of the optical arrangements for the instrument.
EUT Report 93-E-278. 1993. ISBN 90-6144-278-8

(279) Creyghton, Y.L.M. and W.R. Rutgers, E.M. van Veldhuizen
IN-SITU INVESTIGATION OF PULSED CORONA DISCHARGE.
EUT Report 93-E-279. 1993. ISBN 90-6144-279-6

(280) Li, H.Q. and R.P.P. Smeets
GAP-LENGTH DEPENDENT PHENOMENA OF HIGH-FREQUENCY VACUUM ARCS.
EUT Report 93-E-280. 1993. ISBN 90-6144-280-X

(281) Di, Chennian and Jochen A.G. Jess
ON THE DEVELOPMENT OF A FAST AND ACCURATE BRIDGING FAULT SIMULATOR.
EUT Report 94-E-281. 1994. ISBN 90-6144-281-8

(282) Falkus, H.M. and A.A.H. Damen
MULTIVARIABLE H-INFINITY CONTROL DESIGN TOOLBOX: User manual.
EUT Report 94-E-282. 1994. ISBN 90-6144-282-6

(283) Meng, X.Z. and J.G.J. Sloot
THERMAL BUCKLING BEHAVIOUR OF FUSE WIRES.
EUT Report 94-E-283. 1994. ISBN 90-6144-283-4

(284) Rangelrooij, A. van and J.P.M. Voeten
CCSTOOL2: An expansion, minimization, and verification tool for finite state CCS descriptions.
EUT Report 94-E-284. 1994. ISBN 90-6144-284-2

(285) Roer, Th.G. van de
MODELING OF DOUBLE BARRIER RESONANT TUNNELING DIODES: D.C. and noise model.
EUT Report 95-E-285. 1995. ISBN 90-6144-285-0

(286) Dolmans, G.
ELECTROMAGNETIC FIELDS INSIDE A LARGE ROOM WITH PERFECTLY CONDUCTING WALLS.
EUT Report 95-E-286. 1995. ISBN 90-6144-286-9

(287) Liao, Boshu and P. Massee
RELIABILITY ANALYSIS OF AUXILIARY ELECTRICAL SYSTEMS AND GENERATING UNITS.
EUT Report 95-E-287. 1995. ISBN 90-6144-287-7

(288) Weiland, Siep and Anton A. Stoorvogel
OPTIMAL HANKEL NORM IDENTIFICATION OF DYNAMICAL SYSTEMS.
EUT Report 95-E-288. 1995. ISBN 90-6144-288-5

(289) Konieczny, Pawel A. and Lech Józwiak
MINIMAL INPUT SUPPORT PROBLEM AND ALGORITHMS TO SOLVE IT.
EUT Report 95-E-289. 1995. ISBN 90-6144-289-3

(290) Voeten, J.P.M.
POOSL: An object-oriented specification language for the analysis and design of hardware/software systems.
EUT Report 95-E-290. 1995. ISBN 90-6144-290-7