# Semantics of 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

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

by: J.P.M. Voeten

EUT Report 95-E-293
ISBN 90-6144-293-1
October 1995

Eindhoven University of Technology Research Reports

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
Eindhoven, The Netherlands

ISSN 0167-9708 Coden: TEUEDE

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

by

J.P.M. Voeten

EUT Report 95-E-293
ISBN 90-6144-293-1

Eindhoven
October 1995

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Voeten, J.P.M.

Semantics of 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-293)
With ref.
ISBN 90-6144-293-1
NUGI 832
Subject headings: object-oriented methods / specification
languages ; semantics / formal specification.

Semantics of POOSL:
An Object-Oriented Specification Language
for the Analysis and Design of Hardware/Software Systems
J.P.M. Voeten

Abstract

POOSL, an acronym for Parallel Object-Oriented Specification Language, is a specification and design language which is developed as a part of an object-oriented methodology for the specification and design of data processing systems that contain a mixture of software and hardware components. The language is based on 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.

In this report we formalize the language and we argue why such a formalization is necessary. The formal description is a Plotkin-style structural operational semantics. Since POOSL distinguishes data from processes, the semantics is developed in two parts. The data part is a computational semantics which is specified in terms of a transition system. We clarify the formal description through an example in which we compute the semantics of a data expression. The process part is a computational interleaving semantics defined in terms of a labeled transition system. On top of this semantics we define observation equivalence, and we show in an example how to reason about the equivalence of specifications.

Keywords: specification language semantics, object-oriented methods, formal specification.

Voeten, J.P.M.
Semantics of 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-293

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

# List of Figures

# Acknowledgements

First of all I would like to thank my supervisor Prof. M. Stevens for giving me the opportunity to carry out my research and for his continues support. I further wish to thank all my colleagues and ex-colleagues for their help, comments and fruitful discussions. Special thanks are due to A. Verschueren, A. van Rangelrooij, M. Kolsteren and P. van der Putten, who read and commented on earlier drafts of this report. It is the Ph.D. research of A. Verschueren that gave rise to my research project in the first place. Through my two-years cooperation with A. van Rangelrooij I became familiar with the 'real-world' of electrical engineering. M. Kolsteren gave a number of useful suggestions that led to a considerable simplification of the semantics. Currently I work closely together with P. van der Putten on the development of a complete design methodology for hardware/software systems. I would like to thank him for his continuous support and cooperation, his constructive criticism, his accuracy and enthusiasm, and for the many inspiring and teachable conversations.

Further I would like to thank A. Moreira, a former Ph.D. student, whom I met at the ECOOP'94 conference in Bologna. Her research on rigorous object-oriented analysis gave rise to many new insights in the complex world of objects.

Finally I would like to acknowledge the members and ex-members of the Formal Methods Group, J. Baeten, F. de Boer, D. Dams, R. Gerth, J. Hooman, C. Huizing, R. Kuiper, S. Mauw, for answering lots of questions and for giving many useful suggestions. In particular, I would like to thank F. de Boer and C. Huizing who read earlier drafts of this report and who helped in developing a formal semantics.

To Inge

# Chapter 1

# Introduction

## 1.1 Background

Designing data processing systems becomes more and more difficult because of their increasing complexity and because of often competitive time and cost constraints. The Digital Information Systems Group develops methods and tools for the specification, design, and implementation of complex (real-time) data processing systems that contain a mixture of software and hardware components. There is a special interest in the application of object-oriented techniques. Currently there exist a number of accepted object-oriented analysis and design techniques, but unfortunately they all mainly focus on software development. Therefore our group is developing a methodology which is also suited for the design of hardware systems.

An important part of the methodology is a formal specification language called POOSL [Voe94, Voe95], an acronym for Parallel Object-Oriented Specification Language. The key feature of POOSL is that it distinguishes *statically interconnected process objects* from *dynamically moving data objects*. Process objects, or for short processes, are concurrent entities that communicate using one-way synchronous message passing over static channels. Process objects can be composed to form clusters of collaborating objects. A cluster is hierarchically built from process objects and other clusters by *parallel composition*, *channel hiding* and *channel renaming*. Data objects, on the other hand, are sequential entitities used to model internal data of processes and to model data exchanged between different processes. Processes do *not* share any data objects and if data objects are passed form one process to the other, actual *deep copies* are made. This contrasts traditional object-oriented languages where object references are passed instead of the objects themselves. The strict separation between data and processes creates the possibility to model the static structure of a system in an elegant and intuitive way, something which is of utter importance for the specification of software/hardware systems.

We mentioned that POOSL is a *formal* language, which means that the language should be equipped with a formal syntax as well as with a formal semantics. The formal syntax of previous versions of POOSL are given in [Voe94, Voe95]. In this report we give a formal syntax and develop a formal semantics of a stable version of the language.

With respect to the semantics, we have chosen for a Plotkin-style structural operational semantics. An operational semantics emphasizes *how* a specification is executed on some abstract machine. The term *structural* states that the semantics is defined in terms of the *syntactic structure* of specifications.

## 1.2 Motivation

The reasons to develop a semantics, and especially a structural operational semantics, are diverse. Designing a good (specification) language is a very complex task. When one attempts to combine different language concepts, unexpected and counterintuitive interactions arise [Ten91]. Since constructing a *formal* semantics requires a thorough understanding of every "corner" of the language, these interactions can be detected as early as possible and different alternatives can be evaluated systematically. Therefore, a language and its formal semantics should be developed simultaneously, thereby using the semantics as a *tool* which guides the language design. More about this subject is described in Chapter 4.

Next to its use for language design, a semantics, and because of its relative simplicity especially an operational semantics, can be helpful to users of the language. Since reference manuals and language standards are, in general, expressed informally, ambiguities may arise about the precise meaning of some construct. A formal semantics can then be consulted to resolve these ambiguities. Further, a rigorous semantic description can be of assistance in creating manuals or standards in the first place.

Correctness-preserving transformations play in important part in the design methodology mentioned in Section 1.1. A transformation takes a specification and transforms it into another one. A *correctness-preserving* transformation is a special kind of transformation whose *correctness* is known in advance, which means that it establishes a predefined *correctness relation* between the involved specifications. A formal proof of correctness can only be made if the semantics of specifications and the semantics of correctness relations are made precise.

The *correctness* of transformations is often based on *equivalent externally observable behaviour.* Since *denotational semantics* typically emphasize describing systems in terms of their external behaviour, they are often considered a good basis for the support of transformations. However, as languages become more complicated, deciding on an appropriate denotational semantics becomes more and more difficult, which especially applies to parallel (object-oriented) languages. On the other hand, operational semantics, and in particular *structural* operational semantics, have proven to be very fruitful. By defining correctness relations directly on top of an operational semantics, much of the need for denotational semantics has been side-stepped [Hen90].

To be able to validate formal specifications against informal requirements, to analyze the (dynamic) behaviour of specifications, or to implement specifications, a simulator tool or a compiler tool providing (prototype) implementations would be of great use. Since an operational semantics describes *how* specifications are executed rather than just *what* the results of the execution should be, tool implementers can greatly benefit from such a semantic description. This is nicely demonstrated in [Eij89] where a set of simulator functions for Hippo (a LOTOS simulator) is systematically derived from the operational semantics of LOTOS.

Formal verification is a mathematical proof that a specification meets a certain property. Currently, a wide variety of tools are available which automate formal verification. In order to make use of these verification tools, POOSL descriptions have to be translated into so-called labeled transition systems which serve as the basic input models for a large amount of verification tools. Since labeled transition systems and operational semantics are closely related, the latter can be of great help in the construction of translation tools.

## 1.3 Report Organization

The plan of the report is as follows:

- Chapter 2 deals with data objects. The chapter starts with an informal explanation. Then the formal syntax together with a number of context conditions are given. Next, we develop a computational semantics of the data part of POOSL. The realization of the formal semantics of data objects has greatly been influenced by [PAR85] in which an operational semantics of the parallel object-oriented language POOL is given. We conclude Chapter 2 with an example in which the semantics of a complex-number expression is calculated.

- Chapter 3 concerns the process part of the language. We start with an informal explanation, a formal syntax, and a number of context conditions. Then we develop a computational interleaving semantics and we define *observational equivalence* on specifications. In the last part of the chapter we give a proof of the equivalence of a simple hand-shake protocol and a 1-place buffer.

- In Chapter 4 a brief review of the development process of POOSL is given. The chapter describes three encountered problems, possible design alternatives, and the chosen solutions.

- The report finishes with Chapter 5 in which we derive our conclusions.

# 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 contains 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.

Data objects are grouped into *data classes*. A data 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. Because the precise form has not been decided upon, we will not consider inheritance in this report.

POOSL has four predefined classes of commonly used data types, namely *Boolean*, *Integer*, *Real*, and *Char*. Instances of these predefined classes are called *primitive* (data) objects.

The set of messages of these objects correspond to the usual operations of the object's data type.

Next to these primitive objects five other primitive objects exist, named *nil*, *bunk*, *iunk*, *runk*, and *cunk*. *nil* can be considered to be element of every class. Besides an equality (==) message, this object does not recognize any message and the execution aborts when another message is sent to it. *bunk*, *iunk*, *runk*, and *cunk* represent the unknown objects of classes *Boolean*, *Integer*, *Real*, and *Char*. An unknown object recognizes the same messages as the other objects of its class. The calculated results follow obvious rules such as $bunk\ or\ true = true$, $iunk < 6 = bunk$, and $1.567 \times runk = runk$. The unknown objects are introduced to allow the specification of non-deterministic, yet executable, behaviour. Non-determinism is a very powerful tool to achieve abstraction in specifications.

Every object (primitive as well as non-primitive) recognizes a special message called *equality* (==). Through the equality message it is decided whether or not two expressions refer to the same object.

## 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] and is based on the abstract syntax of POOL [PAR85]. We assume that the following sets of syntactic elements are given:

| | | |
|---|---|---|
| *IVar* | instance variables | $x, y, \cdots$ |
| *LVar* | local variables, parameters | $u, v, w, \cdots$ |
| *CName* | data class names | $C, \cdots$ |
| *MName* | method names | $m, \cdots$ |

First, we define the set *PDObj* of Primitive Data Objects with typical elements $\gamma, \cdots$. This set contains boolean objects ($\mathbb{B}$), integer objects ($\mathbb{Z}$), real objects ($\mathbb{R}$), character objects (*Char*), *nil*, *bunk*, *iunk*, *runk*, and *cunk*.

$$PDObj = \mathbb{B} \cup \{bunk\} \cup \mathbb{Z} \cup \{iunk\} \cup \mathbb{R} \cup \{runk\} \cup Char \cup \{cunk\} \cup \{nil\}$$

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

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

The first two expressions are *instance variables*, and *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 message-send expression is evaluated, first the destination expression is evaluated, then the parameters are evaluated from left to right, and finally the message is sent to the destination object. This latter 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 primitive objects. $\underline{\gamma}$ stands for the *direct naming* (textual representation) of primitive object $\gamma$. 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 latter 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

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

The first type of statement is an *expression*. Executing such a statement means that the expression is evaluated and the result is discarded. The effect of the execution is the *side-effect* of the expression evaluation.

Next, we have two *assignment* statements: the first to an *instance variable* and the second to a *local variable* or *parameter*. Upon execution of an assignment statement, the expression is evaluated and the result, a primitive object or a reference to an object of a user-defined class, is assigned to the variable.

*Sequential composition*, the *if-statement*, and the *do-statement* have their usual meaning. If the guard $E$ of the if-statement or the do-statement evaluates to *bunk*, a non-deterministic choice is taken whether the value should be interpreted as *true* or as *false*.

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 ***non-primitive data classes***, comparable with a set of ***system classes*** in Smalltalk. A system is built from a number of data class definitions.

The set *Classdef* of data class definitions, ranging over *CD*, $\cdots$, is defined as

$$\begin{array}{lll} CD ::= & \text{data class} & C \\ & \text{instance variables} & x_1 \cdots x_n \\ & \text{instance methods} & MD_1 \cdots MD_k \end{array}$$

Within a data class definition the functionality of the classes' instances is specified. First, the name of the class is given. 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 $MD, \cdots$

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

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. Another example is the equality (==) message. Through this message it is determined whether two expressions refer to the same object. In this report we will assume that there are classes which use *deepCopy* and equality. These primitives are also defined for every primitive object. Other primitive methods will not be considered here.

## 2.3 Context Conditions

In the previous section we gave the syntax of the data part of POOSL in BNF notation. There are, however, a number of (syntactic) requirements, often called *context conditions*, which have to be satisfied and which cannot be described in BNF. In this section we will informally describe the context conditions with respect to a system *Sys* of classes. The conditions are the following:

(a.) All class names in *Sys* are different.

(b.) All instance variables in a class definition are different.

(c.) All method names of a class are different.

(d.) All parameters and local variables in a method definition are different.

(e.) Every variable used in a method body is either an instance variable of the corresponding class, a method parameter, or a local method variable.

(f.) The class in any new expression is contained in *Sys*.

## 2.4 A Computational Semantics

### 2.4.1 Informal Explanation

A computational semantics is a special kind of operational semantics and is specified by a *transition system*. Transition systems where first used by Hennessey and Plotkin [HP79, Plo81, Plo83] and they were also used by Apt in [Apt81, Apt83]. A transition system is an ensemble $(Conf, \rightarrow)$ where *Conf* is a set of ***configurations*** and where $\rightarrow$ denotes a ***transition relation***. In general, a configuration is of the form $\langle S, I\rangle$ representing a system $S$ together with some amount of information $I$. $S$ is the syntactic part of the configuration and often denotes a statement. The information part $I$ often refers to a state. A configuration represents that system $S$ is to be executed in the context of information $I$. Transition relation $(\rightarrow)$ describes how this execution takes place. The intuitive meaning of transitions $\langle S, I\rangle \rightarrow \langle S', I'\rangle$ is that system $S$ with information $I$ can lead to system $S'$ with information $I'$ in a single *computation* (or execution) step.

The transition relation is defined by a syntax directed deductive system consisting of *rules* and *axioms*. A rule is of the general form

$$\frac{\langle S_1, I_1\rangle \rightarrow \langle S_1', I_1'\rangle, \cdots, \langle S_n, I_n\rangle \rightarrow \langle S_n', I_n'\rangle}{\langle S, I\rangle \rightarrow \langle S', I'\rangle} \text{ if } condition$$

A rule has zero or more *premises* and one *conclusion*. A rule tells us how we can deduce a new transition (the conclusion) from the old ones (the premises). A rule may have a *condition* which has to be fulfilled whenever the rule is to be applied.

Rules without premises are called *axioms*. An axiom tells us what is considered to be a *basic transition*. Usually, the solid line is omitted when a rule is an axiom. So, an axiom has the general form

$$\langle S, I\rangle \rightarrow \langle S', I'\rangle \text{ if } \textit{condition}$$

The transition relation $\rightarrow$ describes the individual steps of an execution. If we apply the relation repeatedly, starting with configuration $\langle S_1, I_1\rangle$, we obtain sequences of configurations, called *derivation sequences*,

$$\langle S_1, I_1\rangle, \langle S_2, I_2\rangle, \langle S_3, I_3\rangle, \cdots$$

such that for all $i \geq 1$ $\langle S_i, I_i\rangle \rightarrow \langle S_{i+1}, I_{i+1}\rangle$. Some of these sequences will be infinite and others will be finite. The finite sequences are of the form

$$\langle S_1, I_1\rangle, \langle S_2, I_2\rangle, \langle S_3, I_3\rangle, \cdots, \langle S_k, I_k\rangle$$

where configuration $\langle S_k, I_k\rangle$ is either a *terminal* or a *stuck* configuration, i.e., there exists no configuration $\langle S, I\rangle$ such that $\langle S_k, I_k\rangle \rightarrow \langle S, I\rangle$. A terminal configuration represents the calculated information obtained by *successful termination*. A stuck configuration represents an *unsuccessful termination*.

We can now give a meaning to a configuration $\langle S_1, I_1\rangle$ by defining its semantics as the set of *all* terminal configurations of *all* possible derivation sequences.

In the following two subsections we will give an operational semantics of the data part of POOSL. Section 2.4.2 will start with a number of definitions. In Section 2.4.3 the transition system is being developed. Section 2.4.4 defines the semantics of configurations in terms of a semantic function $\mathcal{M}$. In Section 2.4.5 the transition system is extended to deal with the primitive *deepCopy* messages. Section 2.4.6 we give an example of the calculation of the semantics of a data expression in POOSL.

### 2.4.2 Definitions

Before we can define our operational semantics we have to give some definitions.

We start defining the set *NDObj* of Non-Primitive Data Objects and let it range over $\alpha$, $\cdots$. Non-primitive data objects are represented by 'capped' integer values. In fact, these 'capped' integer values are really *object identifiers* and not the objects themselves. Most of the time, however, we will blur this distinction and call them objects instead.

$$NDObj = \{\hat{n} \mid n \in \mathbb{N}\}$$

Together with the primitive data objects *PDObj*, this constitutes the set *DObj* of Data Objects, with typical elements $\beta, \cdots$,

$$DObj = NDObj \cup PDObj$$

We define a set of global states $\Sigma$ ranging over $\sigma, \cdots$ as follows:

$$\Sigma = \{\sigma \in (NDObj \cup \{proc\}) \hookrightarrow (IVar \hookrightarrow DObj) \mid Dom(\sigma) \text{ is finite}\}$$

Here, we use $\hookrightarrow$ to indicate that a global state is a partial function. $Dom(\sigma)$ denotes the domain of function $\sigma$. We will denote elements of $NDObj \cup \{proc\}$ by $\delta, \cdots$ and elements of $IVar \hookrightarrow DObj$ by $\phi, \cdots$.

Later in Chapter 3 we will see that every data expression or statement is executed within the context of a single *process* object. Such a process has instance variables, which refer to data objects known to the process. A global state $\sigma \in \Sigma$ stores the values of all these instance variables as well as of those of all non-primitive data objects (indirectly) known to the process. Domain element *proc* identifies the process itself. The other domain elements refer to the non-primitive data objects.

We define a function *MaxId* which retrieves the greatest object identifier contained in a global state. *MaxId* is needed to describe the creation of data objects. If a state does not contain any object identifiers, the function returns 0. Let $\sigma \in \Sigma$. Then

$$MaxId(\sigma) = \begin{cases} 0 & \text{if } Dom(\sigma \restriction NDObj) = \varnothing \\ Max\{n \mid \hat{n} \in Dom(\sigma)\} & \text{if } Dom(\sigma \restriction NDObj) \neq \varnothing \end{cases}$$

Here $\sigma \restriction NDObj$ denotes function $\sigma$ restricted to set *NDObj*.

Next, we define a set *Stack* of (local) stacks, with typical elements $s, \cdots$,

$$Stack = (\{proc\} \times (LVar \hookrightarrow DObj))^*(NDObj \times (LVar \hookrightarrow DObj))^*$$

A stack is a (possibly empty) list of stack elements. An element of a stack denotes a *local variable environment.* Such an element consists of two components. The first component denotes the owner of the environment, which is either the involved process object or some non-primitive data object. The second component stores the values of local (method) variables of the owner. The first component of the top of a stack denotes the object which is currently executing one of its methods. Note that the bottom elements of each stack are owned by the involved process object, whereas the top elements are owned by non-primitive data-objects. We let the set of stack elements $(NDObj \cup \{proc\}) \times (LVar \hookrightarrow DObj)$ range over $e, \cdots$. We further let $Lvar \hookrightarrow DObj$ range over $\chi, \cdots$.

We shall need the following operations upon stacks: For stack elements

$$e_1, \cdots, e_n, e \in ((NDObj \cup \{proc\}) \times (LVar \hookrightarrow DObj))$$

and stack

$$\langle e_1, \cdots, e_n \rangle \in Stack$$

we define

$$\begin{array}{lcll}
pop(\langle e_1, \cdots, e_n \rangle) & = & \langle e_1, \cdots, e_{n-1} \rangle & \text{if } n > 1 \\
push(e, \langle e_1, \cdots, e_n \rangle) & = & \langle e_1, \cdots, e_n, e \rangle & \text{if } \langle e_1, \cdots, e_n, e \rangle \in Stack \\
top(\langle e_1, \cdots, e_n \rangle) & = & e_n & \text{if } n > 1 \\
\mid \langle e_1, \cdots, e_n \rangle \mid & = & n &
\end{array}$$

$\mid s \mid$ denotes the depth (the amount of elements) of stack $s$.

We denote the first and second component of a stack element $e$ by $e(1)$ respectively $e(2)$. So if $e = \langle \delta, \chi \rangle$ then $e(1) = \delta$ and $e(2) = \chi$.

To store for each non-primitive data object its class name, we define the set *Type* of type functions ranging over $\tau, \cdots$

$$Type = NDObj \hookrightarrow CName$$

Armed with these definitions we are able to define our set *Conf* of configurations.

$$Conf = Stat^e \times \Sigma \times Stack \times Type \times Systems$$

A configuration consists of a syntactic part and an information part. The syntactic part is composed of a statement and a system of classes. The information part (the $I$ part described in Section 2.4.1) is composed of a state, a stack, and a type.

The set $Stat^e$, with typical elements $S^e, \cdots$, is an extension of the set $Stat$. This extension is on its turn based on an extended set of expressions $Exp^e$, with typical elements $E^e, \cdots$. The extended sets are defined as follows:

$$\begin{array}{lll}
E^e & ::= & x \\
& \mid & u \\
& \mid & \text{new}(C) \\
& \mid & \text{self} \\
& \mid & E^e\ m(E_1^e, \cdots, E_n^e) \\
& \mid & \underline{\beta} \\
& \mid & S^e;\, E
\end{array}
\qquad
\begin{array}{lll}
S^e & ::= & E^e \\
& \mid & x := E^e \\
& \mid & u := E^e \\
& \mid & S_1^e;\, S_2 \\
& \mid & \text{if } E^e \text{ then } S_1 \text{ else } S_2 \text{ fi} \\
& \mid & \text{do } E \text{ then } S \text{ od} \\
& \mid & \downarrow E^e
\end{array}$$

Here, $\underline{\beta}$ denotes the direct naming of object $\beta$. This construction is incorporated to facilitate the semantic description. $\downarrow E^e$ indicates that a message is outstanding and that the result of the message, which is the value of expression $E^e$, is to be inserted at the place of the $\downarrow$.

### 2.4.3 The Transition System

In this section we will define the transition system of the data part of POOSL. The transition relation

$$\rightarrow \subset \ Conf \times Conf$$

will be defined by the following axioms and rules:

#### Axioms

1. *Object creation*

$$\langle \text{new}(C), \sigma, s, \tau, Sys \rangle \ \rightarrow \ \langle \underline{\hat{n}}, \sigma', s, \tau', Sys \rangle$$

if

$$\begin{array}{l} Sys \ \equiv^{1} \langle CD_1 \cdots CD_j \cdots CD_l \rangle \\ CD_j \ \equiv \ \text{data class } C \text{ instance variables } x_1 \cdots x_p \text{ instance methods } MD_1 \cdots MD_l \end{array}$$

and where

$$\begin{array}{ll} \sigma' & = \sigma\{\phi \ / \ \hat{n}\} \\ Dom(\phi) & = \{x_1, \cdots, x_p\} \\ \phi(x_i) & = nil \\ n & = MaxId(\sigma) + 1 \\ \tau' & = \tau\{C \ / \ \hat{n}\} \end{array}$$

Here we have used the *variant notation* for functions. If $f$ is a (partial) function from $X$ to $Y$, $x, p \in X$ and $y \in Y$, then $f\{y \ / \ x\}$ is defined by

$$f\{y \ / \ x\}(p) = \begin{cases} f(p) & \text{if } p \neq x \text{ and } p \in Dom(f) \\ y & \text{if } p = x \\ \underline{\text{undef}} & \text{if } p \neq x \text{ and } p \notin Dom(f) \end{cases}$$

We write $f\{y/x\}(p) = \underline{\text{undef}}$ to mean that $p \notin Dom(f\{y/x\})$.

2. *Assignment to instance variables*

$$\langle x := \underline{\beta}, \sigma, s, \tau, Sys \rangle \ \rightarrow \ \langle \underline{\beta}, \sigma', s, \tau, Sys \rangle$$

if

$$| \, s \, | > 0$$

and where

$$\begin{array}{ll} \sigma' & = \sigma\{\sigma(\delta)\{\beta \ / \ x\} \ / \ \delta\} \\ \delta & = (top(s))(1) \end{array}$$

[1] We use $\equiv$ to denote syntactic identity.

3. *Assignment to local variables*

$$\langle u := \underline{\beta}, \sigma, s, \tau, Sys\rangle \rightarrow \langle \underline{\beta}, \sigma, s', \tau, Sys\rangle$$

if

$$\mid s \mid > 0$$

and where

$$\begin{aligned} s' &= push(e', pop(s)) \\ e' &= \langle e(1), e(2)\{\beta \ / \ u\}\rangle \\ e &= top(s) \end{aligned}$$

4. *Instance variables*

$$\langle x, \sigma, s, \tau, Sys\rangle \rightarrow \langle \underline{\beta}, \sigma, s, \tau, Sys\rangle$$

if

$$\begin{gathered} \mid s \mid > 0 \\ \sigma(top(s)(1))(x) \neq \underline{\text{undef}} \end{gathered}$$

and where

$$\beta = \sigma(top(s)(1))(x)$$

5. *Local variables*

$$\langle u, \sigma, s, \tau, Sys\rangle \rightarrow \langle \underline{\beta}, \sigma, s, \tau, Sys\rangle$$

if

$$\begin{gathered} \mid s \mid > 0 \\ (top(s)(2))(u) \neq \underline{\text{undef}} \end{gathered}$$

and where

$$\beta = (top(s))(2)(u)$$

6. *Self*

$$\langle \text{self}, \sigma, s, \tau, Sys\rangle \rightarrow \langle \underline{\alpha}, \sigma, s, \tau, Sys\rangle$$

if

$$\begin{gathered} \mid s \mid > 0 \\ top(s)(1) \neq proc \end{gathered}$$

and where

$$\alpha = top(s)(1)$$

7. *Discarding a value*

$$\langle \underline{\beta}\,; S, \sigma, s, \tau, Sys\rangle \;\rightarrow\; \langle S, \sigma, s, \tau, Sys\rangle$$

8. *Method call*

$$\langle \underline{\alpha}\;\; m(\underline{\beta_1}, \cdots, \underline{\beta_k}), \sigma, s, \tau, Sys\rangle \;\rightarrow\; \langle \downarrow E, \sigma, s', \tau, Sys\rangle$$

if

$$\begin{array}{ll} Sys & \equiv \langle CD_1 \cdots CD_j \cdots CD_l\rangle \\ CD_j & \equiv \text{data class } C \text{ instance variables } \cdots \; MD_1 \cdots MD \cdots MD_l \\ C & \equiv \tau(\alpha) \\ MD & \equiv \mathrm{m}(u_1, \cdots, u_k) \;\; \mid v_1 \cdots v_n \mid \;\; E \end{array}$$

and where

$$\begin{array}{ll} s' & = push(e, s) \\ e & = \langle \alpha, \chi\rangle \\ \chi(u_i) & = \beta_i \\ \chi(v_j) & = nil \end{array}$$

9. *Returning the result*

$$\langle \downarrow \underline{\beta}, \sigma, s, \tau, Sys\rangle \;\rightarrow\; \langle \underline{\beta}, \sigma, s', \tau, Sys\rangle$$

if

$$\mid s \mid > 0$$

and where

$$s' = pop(s)$$

10. *Conditional*

$$\langle \text{if } \underline{\beta} \text{ then } S_1 \text{ else } S_2 \text{ fi}, \sigma, s, \tau, Sys\rangle \;\rightarrow\; \begin{cases} \langle S_1, \sigma, s, \tau, Sys\rangle & \text{if } \beta = true \\ & \text{or } \beta = bunk \\ \langle S_2, \sigma, s, \tau, Sys\rangle & \text{if } \beta = false \\ & \text{or } \beta = bunk \end{cases}$$

11. *Do-statement*

$$\begin{array}{l} \langle \text{do } E \text{ then } S \text{ od}, \sigma, s, \tau, Sys\rangle \;\rightarrow \\ \langle \text{if } E \text{ then}(S\,;\; \text{do } E \text{ then } S \text{ od}) \text{ else } \underline{nil} \text{ fi}, \sigma, s, \tau, Sys\rangle \end{array}$$

12. *Primitive-class Integer (and iunk), operators*

$$\langle \underline{\gamma}\ op(\underline{\gamma_1}), \sigma, s, \tau, Sys\rangle \rightarrow \langle \underline{\gamma_2}, \sigma, s, \tau, Sys\rangle \quad \text{if } \gamma, \gamma_1 \in \mathbb{Z} \cup \{iunk\}$$

where

$$\begin{aligned} \gamma_2 &= op(\gamma, \gamma_1) \\ op &= +, -, *, div, mod, \cdots \end{aligned}$$

and

$$\langle \underline{\gamma}\ op(), \sigma, s, \tau, Sys\rangle \rightarrow \langle \underline{\gamma_1}, \sigma, s, \tau, Sys\rangle \quad \text{if } \gamma \in \mathbb{Z} \cup \{iunk\}$$

where

$$\begin{aligned} \gamma_1 &= op(\gamma) \\ op &= sqr, sqrt, asChar, \cdots \end{aligned}$$

13. *Primitive-class Integer (and iunk), relators*

$$\langle \underline{\gamma}\ rel(\underline{\gamma_1}), \sigma, s, \tau, Sys\rangle \rightarrow \langle \underline{\gamma_2}, \sigma, s, \tau, Sys\rangle \quad \text{if } \gamma, \gamma_1 \in \mathbb{Z} \cup \{iunk\}$$

where

$$\begin{aligned} \gamma_2 &= rel(\gamma, \gamma_1) \\ rel &= <, \leq, >, \geq, \cdots \end{aligned}$$

14. *Primitive-class Boolean (and bunk), operators*

$$\langle \underline{\gamma}\ op(\underline{\gamma_1}), \sigma, s, \tau, Sys\rangle \rightarrow \langle \underline{\gamma_2}, \sigma, s, \tau, Sys\rangle \quad \text{if } \gamma, \gamma_1 \in \mathbb{B} \cup \{bunk\}$$

where

$$\begin{aligned} \gamma_2 &= op(\gamma, \gamma_2) \\ op &= and, or, nand, xor, \cdots \end{aligned}$$

and

$$\langle \underline{\gamma}\ op(), \sigma, s, \tau, Sys\rangle \rightarrow \langle \underline{\gamma_1}, \sigma, s, \tau, Sys\rangle \quad \text{if } \gamma \in \mathbb{B} \cup \{bunk\}$$

where

$$\begin{aligned} \gamma_1 &= op(\gamma) \\ op &= not, \cdots \end{aligned}$$

15. *Primitive-class Real (and runk), operators*

$$\langle \underline{\gamma}\ op(\underline{\gamma_1}), \sigma, s, \tau, Sys\rangle \rightarrow \langle \underline{\gamma_2}, \sigma, s, \tau, Sys\rangle \quad \text{if } \gamma, \gamma_1 \in \mathbb{R} \cup \{runk\}$$

where

$$\begin{aligned} \gamma_2 &= op(\gamma, \gamma_1) \\ op &= +, -, *, /, \cdots \end{aligned}$$

and

$$\langle \underline{\gamma}\ op(), \sigma, s, \tau, Sys\rangle \rightarrow \langle \underline{\gamma_1}, \sigma, s, \tau, Sys\rangle \quad \text{if } \gamma \in \mathbb{R} \cup \{runk\}$$

where

$$\begin{aligned} \gamma_1 &= op(\gamma) \\ op &= round, abs, \cdots \end{aligned}$$

16. *Primitive-class Real (and runk), relators*

$$\langle \underline{\gamma}\ rel(\underline{\gamma_1}), \sigma, s, \tau, Sys\rangle \rightarrow \langle \underline{\gamma_2}, \sigma, s, \tau, Sys\rangle \quad \text{if } \gamma, \gamma_1 \in \mathbb{R} \cup \{runk\}$$

where

$$\begin{aligned} \gamma_2 &= rel(\gamma, \gamma_1) \\ rel &= <, \leq, >, \geq, \cdots \end{aligned}$$

17. *Primitive-class Character (and cunk), operators*

$$\langle \underline{\gamma}\ op(), \sigma, s, \tau, Sys\rangle \rightarrow \langle \underline{\gamma_1}, \sigma, s, \tau, Sys\rangle \quad \text{if } \gamma \in Char \cup \{cunk\}$$

where

$$\begin{aligned} \gamma_1 &= op(\gamma) \\ op &= asciiValue, asUppercase, isLetter, isDigit, \cdots \end{aligned}$$

18. *Primitive-class Character (and cunk), relators*

$$\langle \underline{\gamma}\ rel(\underline{\gamma_1}), \sigma, s, \tau, Sys\rangle \rightarrow \langle \underline{\gamma_2}, \sigma, s, \tau, Sys\rangle \quad \text{if } \gamma, \gamma_1 \in Char \cup \{cunk\}$$

where

$$\begin{aligned} \gamma_2 &= rel(\gamma, \gamma_1) \\ rel &= <, \leq, >, \geq, \cdots \end{aligned}$$

19. *Primitive method Equality, primitive objects*

$$\langle \underline{\gamma} == (\underline{\beta}), \sigma, s, \tau, Sys\rangle \rightarrow \begin{cases} \langle \underline{true}, \sigma, s, \tau, Sys\rangle & \text{if } \beta = \gamma \\ \langle \underline{false}, \sigma, s, \tau, Sys\rangle & \text{if } \beta \neq \gamma \end{cases}$$

20. *Primitive method Equality, non-primitive objects*

$$\langle \underline{\alpha} == (\underline{\beta}), \sigma, s, \tau, Sys\rangle \rightarrow \langle \underline{\gamma}, \sigma, s, \tau, Sys\rangle$$

if

$$\begin{array}{ll} Sys & \equiv \langle CD_1 \cdots CD_j \cdots CD_l\rangle \\ CD_j & \equiv \text{data class } C \text{ instance variables } \cdots \; MD_1 \cdots MD \cdots MD_l \\ C & \equiv \tau(\alpha) \\ MD & \equiv ==(u) \;\; \text{primitive} \end{array}$$

and where

$$\gamma = \begin{cases} true & \text{if } \alpha = \beta \\ false & \text{if } \alpha \neq \beta \end{cases}$$

## Rules

a. *Method call 1*

$$\frac{\langle E^e, \sigma, s, \tau, Sys\rangle \rightarrow \langle E^{e\prime}, \sigma', s', \tau', Sys\rangle}{\begin{array}{l}\langle E^e \; m(E_1^e, \cdots, E_n^e), \sigma, s, \tau, Sys\rangle \rightarrow \\ \langle E^{e\prime} \; m(E_1^e, \cdots, E_n^e), \sigma', s', \tau', Sys\rangle\end{array}}$$

b. *Method call 2*

$$\frac{\langle E^e, \sigma, s, \tau, Sys\rangle \rightarrow \langle E^{e\prime}, \sigma', s', \tau', Sys\rangle}{\begin{array}{l}\langle \underline{\beta} \; m(\underline{\beta_1}, \cdots, \underline{\beta_{i-1}}, E^e, \cdots, E_n^e), \sigma, s, \tau, Sys\rangle \rightarrow \\ \langle \underline{\beta} \; m(\underline{\beta_1}, \cdots, \underline{\beta_{i-1}}, E^{e\prime}, \cdots, E_n^e), \sigma', s', \tau', Sys\rangle\end{array}}$$

c. *Assignment to instance variables*

$$\frac{\langle E^e, \sigma, s, \tau, Sys\rangle \rightarrow \langle E^{e\prime}, \sigma', s', \tau', Sys\rangle}{\langle x := E^e, \sigma, s, \tau, Sys\rangle \rightarrow \langle x := E^{e\prime}, \sigma', s', \tau', Sys\rangle}$$

d. *Assignment to local variables*

$$\frac{\langle E^e, \sigma, s, \tau, Sys\rangle \rightarrow \langle E^{e\prime}, \sigma', s', \tau', Sys\rangle}{\langle u := E^e, \sigma, s, \tau, Sys\rangle \rightarrow \langle u := E^{e\prime}, \sigma', s', \tau', Sys\rangle}$$

e. *Method execution*

$$\frac{\langle E^e, \sigma, s, \tau, Sys\rangle \;\rightarrow\; \langle E^{e\prime}, \sigma', s', \tau', Sys\rangle}{\langle \downarrow E^e, \sigma, s, \tau, Sys\rangle \;\rightarrow\; \langle \downarrow E^{e\prime}, \sigma', s', \tau', Sys\rangle}$$

f. *Sequential composition*

$$\frac{\langle S^e, \sigma, s, \tau, Sys\rangle \;\rightarrow\; \langle S^{e\prime}, \sigma', s', \tau', Sys\rangle}{\langle S^e\, ;\, S_1, \sigma, s, \tau, Sys\rangle \;\rightarrow\; \langle S^{e\prime}; \, S_1, \sigma', s', \tau', Sys\rangle}$$

g. *Conditional*

$$\frac{\langle E^e, \sigma, s, \tau, Sys\rangle \;\rightarrow\; \langle E^{e\prime}, \sigma', s', \tau', Sys\rangle}{\begin{array}{l}\langle \text{if } E^e \text{ then } S_1 \text{ else } S_2 \text{ fi}, \sigma, s, \tau, Sys\rangle \;\rightarrow \\ \langle \text{if } E^{e\prime} \text{ then } S_1 \text{ else } S_2 \text{ fi}, \sigma', s', \tau', Sys\rangle\end{array}}$$

### 2.4.4 The Semantic Function $\mathcal{M}$

Now that we have defined the transition relation $\rightarrow$, we will specify what we consider to be the meaning of a configuration. We define the *semantics* of a configuration as the set of *all* terminal configurations of *all* its possible derivation sequences. A configuration is terminal if it is of the form $\langle \underline{\beta}, \sigma, s, \tau, Sys\rangle$. Formally, we define a *semantic function* $\mathcal{M}$ as follows:

$$\mathcal{M} : Conf \rightarrow \mathbb{P}(Conf)$$

where

$$\mathcal{M}(\langle S^e, \sigma, s, \tau, Sys\rangle) = \{\langle \underline{\beta}, \sigma', s', \tau', Sys\rangle \;\mid\; \langle S^e, \sigma, s, \tau, Sys\rangle \;\rightarrow^*\; \langle \underline{\beta}, \sigma', s', \tau', Sys\rangle\}$$

Here, $\mathbb{P}(Conf)$ denotes the *powerset* of *Conf*, i.e., the set of all subsets of *Conf*. $\rightarrow^*$ is the *reflexive-transitive closure* of relation $\rightarrow$ and denotes that a configuration can lead to another configuration in zero or more execution steps.

### 2.4.5 Primitive deepCopy Messages

The transition system defined in Subsection 2.4.3 does not incorporate the definition of so-called *deepCopy* messages. A deepCopy of a non-primitive object creates a *new* object of the same class. The objects referred to by the instance variables of this newly created object are again deepCopies of the objects referred to by the instance variables of the original object. DeepCopies of primitive objects are the objects themselves.

To define the semantics of deepCopy messages we need to introduce a function *copy* and a collection of functions $relabel_{+m}$. Function *copy* takes a triple $\langle\beta,\sigma,\tau\rangle$ of an object (identifier) $\beta$, a state $\sigma$ and a type $\tau$ and delivers a copy of object $\beta$. This copy is again represented by a triple of an object (identifier), a state and a type and is of the form $\langle\beta,\sigma \restriction \mathcal{D},\tau \restriction \mathcal{D}\rangle$. $\mathcal{D}$ is the set of identifiers of all objects that are (indirectly) known to object $\beta$. If $\beta$ denotes a non-primitive object, it is also contained in $\mathcal{D}$. Since primitive objects do not know any other objects, $\mathcal{D}$ is empty if $\beta$ is primitive. The collection of functions $relabel_{+m}$ is used to relabel all object identifiers of some triple $\langle\beta,\sigma,\tau\rangle$ by increasing them by $m$.

To calculate a proper object-copy, we let the input triple $\langle\beta,\sigma,\tau\rangle$ of function *copy* be a so-called *Sys*-structure.

**Definition 2.1**
Let $Sys \in Systems$. A triple $\langle\beta,\sigma,\tau\rangle \in DObj \times \Sigma \times Type$ is a *Sys*-structure if and only if

1. All data object identifiers in $\sigma$ are also known in $\tau$ and vice versa. So $Dom(\sigma \restriction NDObj) = Dom(\tau)$.

2. Every class name, which is referred to in $\tau$, is defined in *Sys*, and the instance variables of all data objects in $\sigma$ conform to their respective class definition. Expressed formally:
   $\forall\ \alpha \in Dom(\tau):\ \ Sys \equiv \langle CD_1 \cdots CD_n\rangle$ and
   $\exists\ i \in \{1,\cdots,n\}\ :\ CD_i \equiv$ data class $\tau(\alpha)$ instance variables $x_1 \cdots x_p\ \cdots$ such that $Dom(\sigma(\alpha)) = \{x_1,\cdots,x_p\}$.

3. $\sigma$ is closed. This means that every data object, referred to by objects (or *proc*) in $\sigma$, is also in $\sigma$:
   $\forall\ \delta \in Dom(\sigma)\ :\ \forall\ x \in Dom(\sigma(\delta))\ :\ \sigma(\delta)(x) \in NDObj \rightarrow \sigma(\delta)(x) \in Dom(\sigma)$

4. If object $\beta$ is non-primitive, then it is part of $\sigma$. So $\beta \in NDObj$ implies $\beta \in Dom(\sigma)$.

We let $Strucs_{Sys}$ denote the set of all *Sys*-structures. □

To compute identifier set $\mathcal{D}$ we introduce a collection of functions $\mathcal{F}_{\sigma,\beta}$.

**Definition 2.2**
Let $\langle\beta,\sigma,\tau\rangle \in Strucs_{Sys}$ and let $V \subseteq Dom(\sigma)$. Then $\mathcal{F}_{\sigma,\beta} : \mathbb{P}(Dom(\sigma)) \rightarrow \mathbb{P}(Dom(\sigma))$ is defined by

$$\mathcal{F}_{\sigma,\beta}(V) = \begin{cases} \emptyset & \text{if } \beta \in PDObj \\ \{\beta\} \cup V \cup & \\ \{\alpha \in NDObj \mid \exists \delta \in V : \exists x \in IVar : \alpha = \sigma(\delta)(x)\} & \text{if } \beta \in NDObj \end{cases}$$

□

For $n \in \mathbb{N}$ we will write $\mathcal{F}^n_{\sigma,\beta}(V)$ to denote the $n$-fold application of $\mathcal{F}_{\sigma,\beta}$ to $V$, so

$$\begin{array}{lcl} \mathcal{F}^0_{\sigma,\beta}(V) & = & V \\ \mathcal{F}^{n+1}_{\sigma,\beta}(V) & = & \mathcal{F}_{\sigma,\beta}(\mathcal{F}^n_{\sigma,\beta}(V)) \end{array}$$

We will now show how functions $\mathcal{F}_{\sigma,\beta}$ and $\mathcal{F}^n_{\sigma,\beta}$ can be applied to calculate the required object identifier set $\mathcal{D}$. Let $\langle\beta,\sigma,\tau\rangle$ be a *Sys*-structure and assume that $\beta \in NDObj$. Then

$$\begin{array}{lcl} \bigcup\{\mathcal{F}^n_{\sigma,\beta}(\varnothing) \mid 0 \leq n \leq 0\} & = & \varnothing \\ \bigcup\{\mathcal{F}^n_{\sigma,\beta}(\varnothing) \mid 0 \leq n \leq 1\} & = & \{\beta\} \\ \bigcup\{\mathcal{F}^n_{\sigma,\beta}(\varnothing) \mid 0 \leq n \leq 2\} & = & \{\beta\} \cup \{\text{each object directly known to object } \beta\} \\ \bigcup\{\mathcal{F}^n_{\sigma,\beta}(\varnothing) \mid 0 \leq n \leq 3\} & = & \{\beta\} \cup \{\text{each object directly known to object } \beta\} \cup \\ & & \quad \{\text{each object directly known to some} \\ & & \quad\ \text{object that is directly known by } \beta\} \\ \ldots & = & \ldots \\ \bigcup\{\mathcal{F}^n_{\sigma,\beta}(\varnothing) \mid 0 \leq n\} & = & \{\beta\} \cup \{\text{each object (indirectly) known to object } \beta\} \end{array}$$

In case $\beta \in PDObj$, $\bigcup\{\mathcal{F}^n_{\sigma,\beta}(\varnothing) \mid 0 \leq n\} = \varnothing$

So clearly we have that $\bigcup\{\mathcal{F}^n_{\sigma,\beta}(\varnothing) \mid 0 \leq n\}$ is *precisely* the set of all object (identifiers) that are (indirectly) known to $\beta$, that is $\mathcal{D} = \bigcup\{\mathcal{F}^n_{\sigma,\beta}(\varnothing) \mid 0 \leq n\}$.

For readers with a more mathematical background it may be interesting to observe that that $\langle\mathbb{P}(Dom(\sigma)), \subseteq\rangle$ is a complete lattice with least element $\varnothing$, and that each $\mathcal{F}_{\sigma,\beta}$ is a continuous function on $\langle\mathbb{P}(Dom(\sigma)), \subseteq\rangle$. By elementary fixed point theory it then follows that $\mathcal{D} = FIX(\mathcal{F}_{\sigma,\beta})$, where $FIX(\mathcal{F}_{\sigma,\beta})$ denotes the least fixed point of $\mathcal{F}_{\sigma,\beta}$.

We are now able to formulate a definition of function *copy*.

**Definition 2.3**
Let $\langle\beta,\sigma,\tau\rangle \in Strucs_{Sys}$ be a *Sys*-structure. Then

$$copy(\langle\beta,\sigma,\tau\rangle) = \langle\beta, \sigma \restriction \mathcal{D}, \tau \restriction \mathcal{D}\rangle$$

where

$$\mathcal{D} = \bigcup\{\mathcal{F}^n_{\sigma,\beta}(\varnothing) \mid 0 \leq n\}$$

□

The elements of the range of function *copy* are characterized by a number of interesting properties:

**Proposition 2.1**
Let $\langle\beta,\sigma,\tau\rangle \in Strucs_{Sys}$ and let $copy(\langle\beta,\sigma,\tau\rangle) = \langle\beta,\sigma',\tau'\rangle$. Then

(a) $\langle \beta, \sigma', \tau' \rangle \in Strucs_{Sys}$.

(b) $copy(\langle \beta, \sigma', \tau' \rangle) = \langle \beta, \sigma', \tau' \rangle$.

(c) $proc \notin Dom(\sigma')$.

□

According to (a) of Proposition 2.1, a *copy* of a *Sys*-structure is again a *Sys*-structure. This means that it contains at least all information about objects that are (indirectly) known by $\beta$. Item (b) states that this *Sys*-structure is *minimal* in the sense that it does not contain any information about objects that are not (indirectly) known by $\beta$. By item (c) we know that it does not contain any information about the involved process object also. In conclusion, (a), (b) and (c) state that $\langle \beta, \sigma', \tau' \rangle$ contains *precisely* the information needed to characterize object $\beta$.

**Proof** of Proposition 2.1

(a) For (a) of Proposition 2.1 we have to prove that $\langle \beta, \sigma', \tau' \rangle$ satisfies requirements (1) $\cdots$ (4) of Definition 2.1. Now requirements (1) and (2) easily follow from the definition of *copy*. For (3) let $\delta \in Dom(\sigma')$, let $x \in Dom(\sigma'(\delta))$ and assume that $\alpha = \sigma'(\delta)(x) \in NDObj$. We have to show that $\alpha \in Dom(\sigma')$. Since $\sigma' = \sigma \restriction \mathcal{D}$ we have that $\delta \in \mathcal{D}$. This means that there exists an $n \in \mathbb{N}$ such that $\delta \in \mathcal{F}^n_{\sigma,\tau}(\emptyset)$. But then by Definition 2.2 $\alpha \in \mathcal{F}^{n+1}_{\sigma,\tau}(\emptyset)$ and thus $\alpha \in \mathcal{D}$. For (4) we observe that that $\beta \in NDObj$ implies that $\beta \in \mathcal{F}^1_{\sigma,\tau}(\emptyset)$ and thus that $\beta \in \mathcal{D}$. Further, since $\beta \in Dom(\sigma)$, we also have $\beta \in Dom(\sigma \restriction \mathcal{D})$ and thus $\beta \in Dom(\sigma')$.

(b) For (b) we have to prove that $copy(\langle \beta, \sigma', \tau' \rangle = \langle \beta, \sigma', \tau' \rangle$. We know that $copy (\langle \beta, \sigma', \tau' \rangle) = \langle \beta, \sigma' \restriction \mathcal{D}', \tau' \restriction \mathcal{D}' \rangle$ where $\mathcal{D}' = \bigcup \{ \mathcal{F}^n_{\sigma',\beta}(\emptyset) \mid 0 \leq n \}$. If we can show that $\mathcal{D}' = \mathcal{D}$ we are ready. For then $\langle \beta, \sigma' \restriction \mathcal{D}', \tau' \restriction \mathcal{D}' \rangle = \langle \beta, \sigma' \restriction \mathcal{D}, \tau' \restriction \mathcal{D} \rangle = \langle \beta, \sigma \restriction \mathcal{D} \restriction \mathcal{D}, \tau \restriction \mathcal{D} \restriction \mathcal{D} \rangle = \langle \beta, \sigma \restriction \mathcal{D}, \tau \restriction \mathcal{D} \rangle = \langle \beta, \sigma', \tau' \rangle$. To prove that $\mathcal{D} = \mathcal{D}'$ it suffices to show that $\mathcal{F}^n_{\sigma',\beta}(\emptyset) = \mathcal{F}^n_{\sigma,\beta}(\emptyset)$ for each $n \in \mathbb{N}$. This can be proved by an easy induction on $n$.

(c) By induction it is easy to show that for all $n \in \mathbb{N}$ $proc \notin \mathcal{F}^n_{\sigma,\beta}(\emptyset)$. Therefore $proc \notin \mathcal{D}$, and thus $proc \notin Dom(\sigma \restriction \mathcal{D}) = Dom(\sigma')$.

This concludes the proof of Proposition 2.1 □

As explained above, each *Sys*-structure $\langle \beta, \sigma', \tau' \rangle$ satisfying property (b) of Proposition 2.1[2], contains *precisely* the information needed to characterize object $\beta$. *Sys*-structures of this kind are called *minimal*. They will later be used extensively to describe *data object passing* between the various processes. The set of all minimal *Sys*-structures is denoted $Strucs_{Sys,min}$.

[2]Note that if $\langle \beta, \sigma', \tau' \rangle$ is a *Sys*-structure satisfying property (b), it also satisfies properties (a) and (c).

**Definition 2.4**
Let $\langle \beta, \sigma, \tau \rangle \in \mathit{Strucs}_{Sys}$. Then

$$\langle \beta, \sigma, \tau \rangle \in \mathit{Strucs}_{Sys,min} \text{ if and only if } \mathit{copy}(\langle \beta, \sigma, \tau \rangle) = \langle \beta, \sigma, \tau \rangle$$

□

Next, we define a collection of functions $\mathit{relabel}_{+m}$. These functions are used to relabel all object identifiers of a minimal *Sys*-structure by increasing them by $m$.

**Definition 2.5**
Let $\langle \beta, \sigma, \tau \rangle$ be a minimal *Sys*-structure, and let $m \in \mathbb{N}$ be a natural number. We define a collection of relabelling functions $\mathit{relabel}_{+m} : \mathit{Strucs}_{Sys,min} \to \mathit{Strucs}_{Sys,min}$ as follows:

$$\mathit{relabel}_{+m}(\langle \beta, \sigma, \tau \rangle) = \langle \beta', \sigma', \tau' \rangle$$

where

1. $\mathit{Dom}(\sigma') = \mathit{Dom}(\tau') = \{\widehat{k+m} \mid \hat{k} \in \mathit{Dom}(\sigma)\}$
2. $\sigma'(\widehat{k+m})(x) = \begin{cases} \widehat{p+m} & \text{if } \sigma(\hat{k})(x) = \hat{p} \in \mathit{NDObj} \\ \sigma(\hat{k})(x) & \text{if } \sigma(\hat{k})(x) \in \mathit{PDObj} \end{cases}$
   for all $\hat{k} \in \mathit{Dom}(\sigma)$ and $x \in \mathit{Dom}(\sigma(\hat{k}))$
3. $\tau'(\widehat{k+m}) = \tau(\hat{k})$ for all $\hat{k} \in \mathit{Dom}(\sigma)$

□

This definition states that relabelling a minimal *Sys*-structure consists of replacing all non-primitive data object identifiers $\hat{k}$ by objects identifiers $\widehat{k+m}$. All primitive objects remain unchanged. The relabelling of a minimal *Sys*-structure yields another minimal *Sys*-structure. The proof of this fact is of a similar complexity as the proof of Proposition 2.1.

Finally, we are able to give the semantics of deepCopy messages. We will extend the transition system defined in Section 2.4.3 with two axioms. The first axiom (21) deals with deepCopies of primitive objects and the other (22) with deepCopies of non-primitive objects.

21. *Primitive method deepCopy, primitive objects*

$$\langle \underline{\gamma}\ \mathit{deepCopy}(), \sigma, s, \tau, \mathit{Sys} \rangle \;\to\; \langle \underline{\gamma}, \sigma, s, \tau, \mathit{Sys} \rangle$$

22. *Primitive method deepCopy, non-primitive objects*

$$\langle \underline{\alpha}\ deepCopy(), \sigma, s, \tau, Sys\rangle \rightarrow \langle \underline{\alpha'''}, \sigma''', s, \tau''', Sys\rangle$$

if $\langle \alpha, \sigma, \tau\rangle$ is a *Sys*-structure and if

$$\begin{array}{ll} Sys & \equiv \langle CD_1 \cdots CD_j \cdots CD_l\rangle \\ CD_j & \equiv \text{data class } C \text{ instance variables } \cdots\ MD_1 \cdots MD \cdots MD_l \\ C & \equiv \tau(\alpha) \\ MD & \equiv \mathbf{deepCopy}()\ \ \text{primitive} \end{array}$$

where

$$\begin{array}{l} copy(\langle \alpha, \sigma, \tau\rangle) = \langle \alpha', \sigma', \tau'\rangle \\ relabel_{+MaxId(\sigma)}(\langle \alpha', \sigma', \tau'\rangle) = \langle \alpha'', \sigma'', \tau''\rangle \\ \alpha''' = \alpha'' \\ \sigma''' = \sigma \cup \sigma'' \\ \tau''' = \tau \cup \tau'' \end{array}$$

Axiom 22 states that a deepCopy of object $\alpha$ is being made in three steps. In the first step a *copy* is made. This copy is being relabelled by adding the number $MaxId(\sigma)$ to all its object identifiers. $MaxId(\sigma)$ is the largest object identifier which is contained in the original state $\sigma$, so the newly created identifiers are all "fresh". As a final step, the relabelled object is added to the original state, and the (new) object identifier $\alpha'''$ associated with $\alpha$ is the evaluated result of the deepCopy operation.

The theory of this subsection is not just developed to describe deepCopy messages. Later, we will need the same theory to describe data object passing between process objects.

### 2.4.6 Example: Complex Numbers

In this subsection we give an example of the calculation of the semantics of a configuration representing the addition of two complex numbers. A definition of a system *Sys* which contains a class *Complex* of complex numbers is as follows:

$Sys\ \equiv\ \langle$

```
data class            Complex
instance variables    re
                      im
instance methods      init(r, i)     add(comp)
                        re := r;       | i r res |
                        im := i;       r ← self real + (comp real );
                        self           i ← self imag + (comp imag );
```

```
                         res ← new(Complex) init(r, i);
            real         res
              re

            imag
              im

    )^3
```

Calculating the sum of complex numbers, say (3+4i) and (8+9i), can be performed by evaluating

$$(\text{new}(\mathit{Complex})\ \mathit{init}(3,4))\ \mathit{add}(\text{new}(\mathit{Complex})\ \mathit{init}(8,9))$$

Syntactic entities 3, 4, 8, and 9 denote the direct naming of natural numbers 3, 4, 8 and 9.

The evaluation of this expression is reflected by a derivation sequence, starting with configuration

$$\langle(\text{new}(\mathit{Complex})\ \mathit{init}(3,4))\ \mathit{add}(\text{new}(\mathit{Complex})\ \mathit{init}(8,9)), \varnothing, \langle\rangle, \varnothing, \mathit{Sys}\rangle$$

In this configuration we have put brackets around expression new(*Complex*) *init*(3, 4), although they are not part of the abstract syntax defined in Section 2.2. Such brackets are used in ambiguous situations to explain the intended structure of expressions or statements.

To ease readability we represent functions in a way explained by the following example:

Assume

$$F : A \hookrightarrow (B \hookrightarrow C)$$

is defined by

$$F = \{(a_1, \{(b_1, c_1), (b_2, c_1), (b_3, c_2)\}), (a_2, \{(b_2, c_3), (b_4, c_1)\}), (a_3, \varnothing)\}$$

then we represent $F$ as

$$F = \{a_1.b_1 \rightarrow c_1, a_1.b_2 \rightarrow c_1, a_1.b_3 \rightarrow c_2, a_2.b_2 \rightarrow c_3, a_2.b_4 \rightarrow c_1, a_3 \rightarrow \varnothing\}$$

For the first step of the derivation we have to find a configuration, say *conf*, such that

$$\langle(\text{new}(\mathit{Complex})\ \mathit{init}(3,4))\ \mathit{add}(\text{new}(\mathit{Complex})\ \mathit{init}(8,9)), \varnothing, \langle\rangle, \varnothing, \mathit{Sys}\rangle \ \rightarrow\ \mathit{conf}$$

Since (new(*Complex*) *init*(3, 4)) *add*(new(*Complex*) *init*(8, 9)) is of the form $E^e\ m(E_1^e, \cdots, E_n^e)$ (choose $E^e \equiv$ new(*Complex*) *init*(3, 4), $m \equiv \mathit{add}$, $n = 1$, and $E_1^e \equiv$ new(*Complex*) *init*(3, 9)), we can apply [Rule a] and deduce

[3]Note that in stead of self *real* and self *imag* we could have written *re* respectively *im*.

$$conf = \langle E^{e\prime}\ add(\text{new}(Complex)\ init(8,9)), \sigma', s', \tau', Sys\rangle$$

if

$$\langle \text{new}(Complex)\ init(3,4), \varnothing, \langle\rangle, \varnothing, Sys\rangle \ \rightarrow\ \langle E^{e\prime}, \sigma', s', \tau', Sys\rangle$$

So, to calculate *conf*, we first have to calculate the latter transition. By applying [rule a] again we see that

$$\langle E^{e\prime}, \sigma, s', \tau', Sys\rangle = \langle E^{e\prime\prime}\ init(3,4), \sigma'', s'', \tau'', Sys\rangle$$

if

$$\langle \text{new}(Complex), \varnothing, \langle\rangle, \varnothing, Sys\rangle \ \rightarrow\ \langle E^{e\prime\prime}, \sigma'', s'', \tau'', Sys\rangle$$

We can now apply [axiom 1] to deduce

$$\langle E^{e\prime\prime}, \sigma'', s'', \tau'', Sys\rangle = \langle \underline{\hat{1}}, \{\hat{1}.re \rightarrow nil, \hat{1}.im \rightarrow nil\}, \langle\rangle, \{\hat{1} \rightarrow Complex\}, Sys\rangle$$

This implies that

$$\langle E^{e\prime}, \sigma, s', \tau', Sys\rangle = \langle \underline{\hat{1}}\ \ init(3,4), \{\hat{1}.re \rightarrow nil, \hat{1}.im \rightarrow nil\}, \langle\rangle, \{\hat{1} \rightarrow Complex\}, Sys\rangle$$

and thus

$$\begin{aligned} conf = \langle & (\underline{\hat{1}}\ \ init(3,4))\ add(\text{new}(Complex)\ init(8,9)), \\ & \{\hat{1}.re \rightarrow nil, \hat{1}.im \rightarrow nil\}, \langle\rangle, \{\hat{1} \rightarrow Complex\}, Sys\rangle \end{aligned}$$

The other transition steps of the derivation sequence are deduced in a similar way. The result of a partial derivation is as follows:

$$\begin{aligned} & \langle (\text{new}(Complex)\ init(3,4))\ add(\text{new}(Complex)\ init(8,9)), \\ & \varnothing, \langle\rangle, \varnothing, Sys\rangle \end{aligned}$$

$\rightarrow$ {1. by [rules a,a] and [axiom 1] }

$$\begin{aligned} & \langle (\underline{\hat{1}}\ \ init(3,4))\ \ add(\text{new}(Complex)\ init(8,9)), \\ & \{\hat{1}.re \rightarrow nil, \hat{1}.im \rightarrow nil\}, \langle\rangle, \{\hat{1} \rightarrow Complex\}, Sys\rangle \end{aligned}$$

$\rightarrow$ {2. by [rule a] and [axiom 8] }

$$\begin{aligned} & \langle \downarrow (re := r;\ im := i; \text{self})\ add(\text{new}(Complex)\ init(8,9)), \\ & \{\hat{1}.re \rightarrow nil, \hat{1}.im \rightarrow nil\}, \\ & \langle\langle \hat{1}, \{r \rightarrow 3, i \rightarrow 4\}\rangle\rangle, \{\hat{1} \rightarrow Complex\}, Sys\rangle \end{aligned}$$

$\rightarrow$ {3. by [rules a,e,f,c] and [axiom 5] }

$$\begin{aligned} & \langle \downarrow (re := 3;\ im := i; \text{self})\ add(\text{new}(Complex)\ init(8,9)), \\ & \{\hat{1}.re \rightarrow nil, \hat{1}.im \rightarrow nil\}, \\ & \langle\langle \hat{1}, \{r \rightarrow 3, i \rightarrow 4\}\rangle\rangle, \{\hat{1} \rightarrow Complex\}, Sys\rangle \end{aligned}$$

$\rightarrow$ {4. by [rules a,e,f] and [axiom 2] }

$\langle\downarrow (\underline{3}; \mathit{im} := i; \text{self})\ \mathit{add}(\text{new}(\mathit{Complex})\ \mathit{init}(8,9)),$
$\{\hat{1}.\mathit{re} \rightarrow 3, \hat{1}.\mathit{im} \rightarrow \mathit{nil}\},$
$\langle\langle\hat{1}, \{r \rightarrow 3, i \rightarrow 4\}\rangle\rangle, \{\hat{1} \rightarrow \mathit{Complex}\}, \mathit{Sys}\rangle$

$\rightarrow$ {5. by [rules a,e] and [axiom 7] }

$\langle\downarrow (\mathit{im} := i; \text{self})\ \mathit{add}(\text{new}(\mathit{Complex})\ \mathit{init}(8,9)),$
$\{\hat{1}.\mathit{re} \rightarrow 3, \hat{1}.\mathit{im} \rightarrow \mathit{nil}\},$
$\langle\langle\hat{1}, \{r \rightarrow 3, i \rightarrow 4\}\rangle\rangle, \{\hat{1} \rightarrow \mathit{Complex}\}, \mathit{Sys}\rangle$

$\rightarrow$ {6. by [rules a,e,f,c] and [axiom 5] }

$\langle\downarrow (\mathit{im} := 4; \text{self})\ \mathit{add}(\text{new}(\mathit{Complex})\ \mathit{init}(8,9)),$
$\{\hat{1}.\mathit{re} \rightarrow 3, \hat{1}.\mathit{im} \rightarrow \mathit{nil}\},$
$\langle\langle\hat{1}, \{r \rightarrow 3, i \rightarrow 4\}\rangle\rangle, \{\hat{1} \rightarrow \mathit{Complex}\}, \mathit{Sys}\rangle$

$\rightarrow$ {7. by [rules a,e,f] and [axiom 2] }

$\langle\downarrow (4; \text{self})\ \mathit{add}(\text{new}(\mathit{Complex})\ \mathit{init}(8,9)),$
$\{\hat{1}.\mathit{re} \rightarrow 3, \hat{1}.\mathit{im} \rightarrow 4\},$
$\langle\langle\hat{1}, \{r \rightarrow 3, i \rightarrow 4\}\rangle\rangle, \{\hat{1} \rightarrow \mathit{Complex}\}, \mathit{Sys}\rangle$

$\rightarrow$ {8. by [rules a,e] and [axiom 7] }

$\langle\downarrow (\text{self})\ \mathit{add}(\text{new}(\mathit{Complex})\ \mathit{init}(8,9)),$
$\{\hat{1}.\mathit{re} \rightarrow 3, \hat{1}.\mathit{im} \rightarrow 4\},$
$\langle\langle\hat{1}, \{r \rightarrow 3, i \rightarrow 4\}\rangle\rangle, \{\hat{1} \rightarrow \mathit{Complex}\}, \mathit{Sys}\rangle$

$\rightarrow$ {9. by [rules a,e] and [axiom 6] }

$\langle\downarrow (\underline{1})\ \mathit{add}(\text{new}(\mathit{Complex})\ \mathit{init}(8,9)),$
$\{\hat{1}.\mathit{re} \rightarrow 3, \hat{1}.\mathit{im} \rightarrow 4\},$
$\langle\langle\hat{1}, \{r \rightarrow 3, i \rightarrow 4\}\rangle\rangle, \{\hat{1} \rightarrow \mathit{Complex}\}, \mathit{Sys}\rangle$

$\rightarrow$ {10. by [rule a] and [axiom 9] }

$\langle\underline{1}\ \mathit{add}(\text{new}(\mathit{Complex})\ \mathit{init}(8,9)),$
$\{\hat{1}.\mathit{re} \rightarrow 3, \hat{1}.\mathit{im} \rightarrow 4\}, \langle\rangle, \{\hat{1} \rightarrow \mathit{Complex}\}, \mathit{Sys}\rangle$

$\rightarrow^*$ {11. and by many other rules and axioms }

$\langle\underline{\hat{3}},$
$\{\hat{1}.\mathit{re} \rightarrow 3, \hat{1}.\mathit{im} \rightarrow 4, \hat{2}.\mathit{re} \rightarrow 8, \hat{2}.\mathit{im} \rightarrow 9, \hat{3}.\mathit{re} \rightarrow 11, \hat{3}.\mathit{im} \rightarrow 13\}, \langle\rangle,$
$\{\hat{1} \rightarrow \mathit{Complex}, \hat{2} \rightarrow \mathit{Complex}, \hat{3} \rightarrow \mathit{Complex}\}, \mathit{Sys}\rangle$

We therefore have

$$\begin{aligned}&\mathcal{M}(((\text{new}(\mathit{Complex})\ \mathit{init}(3,4))\ \mathit{add}(\text{new}(\mathit{Complex})\ \mathit{init}(8,9)),\varnothing,\langle\rangle,\varnothing,\mathit{Sys}\rangle) =\\ &\{(\langle\underline{\hat{3}},\{\hat{1}.\mathit{re}\rightarrow 3,\hat{1}.\mathit{im}\rightarrow 4,\hat{2}.\mathit{re}\rightarrow 8,\hat{2}.\mathit{im}\rightarrow 9,\hat{3}.\mathit{re}\rightarrow 11,\hat{3}.\mathit{im}\rightarrow 13\},\langle\rangle,\\ &\{\hat{1}\rightarrow \mathit{Complex},\hat{2}\rightarrow \mathit{Complex},\hat{3}\rightarrow \mathit{Complex}\},\mathit{Sys}\rangle)\}\end{aligned}$$

So, the result of sum of complex number (3+4i) and (8+9i) is an object denoted as $\hat{3}$. This object is of class *Complex* and represents value $(11 + 13i)$ (instance variable *re* refers to 11 and *im* refers to 13), precisely as we would expect.

In this case the set of terminal configurations is *singleton*. Note that in general this in not necessarily true. If the execution of a statement loops or blocks, for example, the set of terminal configurations may be *empty*. Further, if the execution involves non-deterministic *bunk* objects, the set may contain more than one element.

# Chapter 3

# Processes, Clusters and System Specifications

## 3.1 Informal Explanation

This chapter describes the process-oriented part of POOSL. This part is based upon the language of data objects described in the previous chapter. A specification in POOSL consists of a fixed set of *process objects* and *process-object clusters*, which are composed by *parallel composition*, *channel hiding* and *channel renaming*. For convenience we will often use the terms *processes* and *clusters* to denote process objects respectively process-object clusters. Processes and clusters are statically interconnected in a topology of channels, through which they can communicate by exchanging messages. Message exchange is based upon the *synchronous (rendez-vous) pair-wise message-passing mechanism* of CCS [Mil80, Mil89].

The grain of concurrency in POOSL is the process. Processes communicate by sending each other messages. 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 as 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. Processes send and receive messages by execution *message-send* respectively *message-receive* statements. These statements are combined by *sequential composition*, *choice operators*, *guarded commands*, *conditional statements* and *do-statements* to describe the temporal communication behaviour of processes.

A process object can call one of its *methods*. Methods are comparable with procedures of imperative programming languages such as C or Pascal. However, procedures of imperative programming languages are in general expected to terminate and can therefore only express *finite* behaviour. Methods in POOSL, on the other hand, can be used to express *infinite* (non-terminating) behaviour. Such infinite behaviour is specified by defining methods in a (mutual) *tail-recursive* manner (see also Section 4.3). The concept of tail recursion has proven to be very useful for the specification of hardware/software systems with complex dynamic (communication) behaviour.

In almost any complex hardware/software system messages can be identified that disrupt the normal course of behaviour. In general, such a message requires an immediate response. Therefore, a system should be able to accept such a message at any time, no matter what other activities are going on. To specify such behaviour, POOSL has a special *disrupt* operator. This operator is similar to the *disabling* operator of LOTOS [EVD89].

Processes contain internal data in the form of data objects (also called traveling 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 activities 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 (whence the term traveling object). 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. The concept of traveling object was first introduced in [Ver92].

Next to processes, POOSL supports clusters. A cluster is hierarchically built from processes and other clusters and acts as an abstraction of these. The constituents of a cluster are composed by parallel composition, channel hiding and channels renaming. These combinators are based upon similar combinators originally used in CCS [Mil80, Mil89].

Processes and clusters are grouped in classes, just as data objects. Members of classes are called instances. A process is an instance of a *process class*, and a cluster is an instance of a *cluster class.* Each class is parameterized by *expression parameters.* Expression parameters refer to data objects and are used to initialize an instance of a class. Future versions of POOSL should support some form of *inheritance* among process classes or cluster classes. The precise form, however, has not been 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:

| | | |
|---|---|---|
| *CName*$^p$ | process class names | $C^p, \cdots$ |
| *CName*$^c$ | cluster class names | $C^c, \cdots$ |
| *Chan* | communication channels | $ch, \cdots$ |
| *EPar* | expression parameters | $P, \cdots$ |

We define

| | | |
|---|---|---|
| *CName*$^{pc}$ | *CName*$^p \cup$ *CName*$^c$ | $C^{pc}, \cdots$ |
| *Var* | *IVar* $\cup$ *LVar* | $p, \cdots$ |

Next we define the set *Stat*$^p$ of process statements. These statements are used to specify the behaviour of process objects.

$$\begin{aligned} S^p ::= \; & S \\ \mid \; & ch!m(E_1, \cdots, E_n) \\ \mid \; & ch?m(p_1, \cdots, p_m \mid E) \\ \mid \; & m(E_1, \cdots, E_n)(p_1, \cdots, p_m) \\ \mid \; & S_1^p;\ S_2^p \\ \mid \; & S_1^p \text{ or } S_2^p \\ \mid \; & [E]S^p \\ \mid \; & \text{if } E \text{ then } S_1^p \text{ else } S_2^p \text{ fi} \\ \mid \; & \text{do } E \text{ then } S^p \text{ od} \\ \mid \; & S_1^p >> S_2^p \end{aligned}$$

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 $E_1, \cdots, E_n$, which refer to data objects, on channel $ch$. A message-receive statement $ch?m(p_1, \cdots, p_n \mid E)$ indicates that a process is willing to receive message $m$ with parameters $p_1, \cdots, p_m$ under condition $E$. $E$ is a boolean expression which may depend on the input parameters $p_1, \cdots, p_m$. It may, however, also depend on other local variables or on instance variables.

For a message to be transferred, exactly two process objects are needed. One of the objects should be executing a message-send statement, and the other a message-receive statement.

These statements must refer to the same channel and the same message. Further, the number of parameters has to be equal. Finally, the reception condition of the receiving object should allow message reception.

A rendez-vous procedure is performed in the following way: First, the message parameters $E_1, \cdots, E_n$ of the message-send statement are evaluated from left to right. Then deep copies of these parameters are bound to the input parameters $p_1, \cdots, p_m$ of the message-receive statement. After that, conditional expression $E$ is evaluated at the receiving process. If the expression evaluates to *true* or *bunk*, the rendez-vous is successfully terminated. In all other cases, the states of both processes are restored to the states just before the rendez-vous procedure started and no message is passed. This basically implies that the check whether a message may be passed is performed transparently for both processes.

The fourth statement is a *method call.* By means of such a method-call statement a process object can call one of 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 and output parameters are initialized to *nil.* Then the method body is executed. If this execution successfully terminates, the output parameters of the method are bound to variables $p_1, \cdots, p_n$.

The fifth sort of statement is *sequential composition*, which is indicated with a semicolon. Sequential composition has its usual meaning.

Next we have the *choice statement* $S_1^p$ or $S_2^p$, indicating that a process can choose between two alternatives. A process always leaves both alternatives open, at least until one of the alternatives can actually be executed. So, a choice is never made *a priori.* As soon as one of the statements has performed an execution step, the other is discarded. In general, the process environment will permit only one of the alternatives. If both alternatives are open, the choice is made *non-deterministically.* The choice statement has the same meaning as the summation combinator (+) of CCS.

$[E]S^p$ is a *guarded command.* $E$ is a boolean expression, called the *guard* of statement $S^p$. A guarded command is executed as follows: First, expression $E$ is evaluated. If it evaluates to *true* or *bunk*, an attempt is made to perform an execution step of rest statement $S^p$. If this attempt succeeds this execution step actually takes place and the execution of $S^p$ is continued. Otherwise, the state of the executing process object is restored to the state just before the expression was evaluated, and the guarded command blocks, i.e., is not executed. In general, guarded commands are used in combination with a choice statement. If a blocking guarded command is used in isolation, the executing process has a danger for complete blocking.

The *if*-statement and the *do*-statement have their usual meaning.

Finally we have the *disrupt*-statement. $S_1^p >> S_2^p$ denotes that statement $S_1^p$ is executed until it is disrupted by $S_2^p$. If an execution step of $S_2^p$ can be performed, this step actually takes place and the execution of $S_2^p$ continues. Statement $S_1^p$ is thereby completely discarded. Upon succesful termination of $S_1^p$, the complete disrupt-statement successfully terminates.

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

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

Such a system is a set of process and cluster classes. It is built from a number of class definitions. The set of all class definitions $Classdef^p$ ranges over $CD^p, \cdots$ and is defined as

| | | | |
|---|---|---|---|
| $CD^p$ | $::=$ | process class | $C^p\langle y_1, \cdots, y_r \rangle$ |
| | | instance variables | $x_1 \cdots x_n$ |
| | | communication channels | $ch_1 \cdots ch_k$ |
| | | message interface | $l_1^a \cdots l_m^a$ |
| | | initial method call | $m(E_1, \cdots, E_q)()$ |
| | | instance methods | $MD_1^p \cdots MD_k^p$ |
| | $\mid$ | cluster class | $C^c\langle P_1, \cdots, P_r \rangle$ |
| | | communication channels | $ch_1 \cdots ch_k$ |
| | | message interface | $l_1^a \cdots l_m^a$ |
| | | behaviour specification | $BSpec^p$ |

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

A specification of a process class starts with the *name* of that class together with a number of *instance variables* between angle brackets. Then the set of *all* instance variables of the class is specified. These variables model the *private* or *internal* data of each instance of the class. Upon initialization of an instance of the class, the instance variables specified between angle brackets are bound to a set of externally supplied data objects. All the other instance variables are initialized to *nil*. 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 abstract send or receive actions, also called abstract communication actions. Such an action 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 $\mathcal{L}^a$ of all abstract actions, has typical elements $l^a, \cdots$ and is defined as

$$\begin{aligned} l^a &::= ch!m[\underline{n}] \\ &\mid\ ch?m[\underline{n}] \end{aligned}$$

The first clause states that a process or cluster, at some point in time, can send message $m$, together with $n$ data objects, to channel $ch$. The second, complementary, abstract action states that message $m$, with $n$ data objects can be received from channel $ch$. $\underline{n}$ denotes the textual representation of natural number $n$. The actions are called *abstract* because they contain no information about the precise transferred data objects; only the amount of objects is indicated [1]. A message interface serves as an abstract description, often called a *signature*, 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 $m(E_1, \cdots, E_q)()$. After initializing its instance variables, a process object always starts its activities by calling its initial method. This method does not contain any output parameters.

The last part of a 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 process method definitions, with typical elements $MD^p, \cdots$, is defined as

$$\begin{aligned} MD^p \;::=\; & \mathrm{m}(u_1, \cdots, u_m)(v_1, \cdots, v_n) \\ & \mid w_1 \cdots w_k \mid S^p \end{aligned}$$

A method definition contains a header with name $m$, input parameters $u_1, \cdots, u_m$ and output parameters $v_1, \cdots, v_n$. 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. Method $m$ is invoked through an $m(E_1, \cdots, E_m)(p_1, \cdots, p_n)$ statement. Such a statement is executed by first evaluating $E_1, \cdots, E_n$ from left to right and by binding the results to input parameters $u_1, \cdots, u_m$. The output parameters $v_1, \cdots, v_n$ and local variables $w_1, \cdots, w_k$ are initialized to *nil*. Then statement $S^p$ is executed. If this statement terminates successfully, the output parameters $v_1, \cdots, v_n$ are bound to variables $p_1, \cdots, p_n$ and the method execution terminates.

The second kind of classes are called cluster classes. A cluster class is built from other classes, which are either process classes or cluster classes themselves. A class definition of a cluster class consists of a cluster class name with a number of *expression parameters* between angle brackets, a number of communication channels, and a message interface. The behaviour of a cluster class is specified by means of a *parameterized behaviour specification.* The set *BSpecifications*$^p$ of all parameterized behaviour specifications has typical elements *BSpec*$^p$ and is defined as follows:

$$\begin{aligned} BSpec^p \;::=\; & C^p(PE_1, \cdots, PE_r) \\ \mid\; & C^c(PE_1, \cdots, PE_r) \\ \mid\; & BSpec_1^p \parallel BSpec_2^p \end{aligned}$$

[1]In practice it may be convenient to specify names of message parameters. One then for example specifies $c!m(x)$ in stead of $c!m[1]$

$$\begin{array}{ll} | & BSpec^p \setminus L \\ | & BSpec^p[f] \end{array}$$

Here each $PE_i$ denotes a parameterized expression. The set of all parameterized expressions is *ParExp* and ranges over $PE, \cdots$. *ParExp* is defined as

$$\begin{array}{rcl} PE & ::= & P \\ & | & E \\ & | & PE\ m(PE_1, \cdots, PE_n) \\ & | & PS; PE \end{array}$$

where $PS$ is a typical element of the set *ParStat* of parameterized statements.

$$\begin{array}{rcl} PS & ::= & E \\ & | & x := PE \\ & | & u := PE \\ & | & PS_1; PS_2 \\ & | & \text{if } PE \text{ then } PS_1 \text{ else } PS_2 \text{ fi} \\ & | & \text{do } PE \text{ then } PS \text{ od} \end{array}$$

$L \subseteq Chan$ denotes a set of channels and $f$ is a so-called *channel renaming function.* The collection of all channel renaming functions is denoted *ChanRen* and ranges over $f, \cdots$.

$$ChanRen = Chan \rightarrow Chan$$

The first two parameterized behaviour specifications $C^p(PE_1, \cdots, PE_r)$ and $C^c(PE_1, \cdots, PE_r)$ denote a single parameterized instance of some process class $C^p\langle y_1, \cdots, y_r\rangle$ and some cluster class $C^c\langle P_1, \cdots, P_r\rangle$ respectively.

The next kind of specifications $BSpec^p_1 \parallel BSpec^p_2$ expresses the *parallel composition* of specifications $BSpec^p_1$ and $BSpec^p_2$. Assume, for example, that the class definition of a class, say $C^c\langle P_1, P_2\rangle$, contains behaviour specification $C^p_1(P_1) \parallel C^p_2(P_2)$. This specification expresses the behaviour of two (parameterized) instances, one of some process class $C^p_1\langle y_1\rangle$ and the other of some process class $C^p\langle y_2\rangle$, which execute in parallel and which (perhaps) communicate through their common channels. The channel set of class $C^c\langle P_1, P_2\rangle$ is the union of the channel sets of classes $C^p\langle y_1\rangle$ and $C^p\langle y_2\rangle$. An instance of class $C^c\langle P_1, P_2\rangle$ can send and receive any message which can be sent and received by instances of either $C^p_1\langle y_1\rangle$ or $C^p_2\langle y_2\rangle$. The parallel composition combinator is comparable with the composition combinator of CCS.

The fourth kind of behaviour specification is called *channel hiding.* A channel hiding $BSpec^p \setminus L$ expresses a specification $BSpec^p$ from which the channels in $L$ are made *unobservable.* This means that other (external) instances cannot communicate through channels in $L$ with instances contained in specification $BSpec^p$. Assume, for example, that the method definition of a class $C^c\langle P_1, P_2\rangle$ contains a behaviour specification $(C_1^p(P_1) \parallel C_2^p(P_2))\backslash\{ch\}$. This means that, even though classes $C_1^p\langle y_1\rangle$ and $C_2^p\langle y_2\rangle$ may contain channel $ch$, class $C^c\langle P_1, P_2\rangle$ may not. Channel $ch$ may only be used for the communication between (parameterized) instances $C_1^p(P_1)$ and $C_2^p(P_2)$. The channel hiding constructor is similar to the *restriction* combinator of CCS.

The last sort of specification expresses a *channel renaming.* The channel renaming $BSpec^p$ $[f]$ denotes a specification $BSpec^p$ from which the channels are renamed as dictated by $f$. We shall often write $ch_1'/ch_1, \cdots, ch_n'/ch_n$ for the renaming 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 several instances of the same class are used within different environments and have to communicate through different channels.

We are now ready to define what we consider to be a specification of a system. A system specification consists of four parts. The first part is a behaviour specification *BSpec* which expresses how the actual system is composed from instances of classes defined in $Sys^p$. A behaviour specification is a parameterized behaviour specification which contains no expression parameters. We will let *BSpecifications* denote the set of all behaviour specifications, and we let it range over $BSpec, \cdots$

$$\begin{array}{rcl} BSpec & ::= & C^p(E_1, \cdots, E_r) \\ & | & C^c(E_1, \cdots, E_r) \\ & | & BSpec_1 \parallel BSpec_2 \\ & | & BSpec \backslash L \\ & | & BSpec[f] \end{array}$$

$C^p(E_1, \cdots, E_r)$ and $C^c(E_1, \cdots, E_r)$ denote an instance of some process class $C^p\langle y_1, \cdots, y_r\rangle$ respectively of some cluster class $C^c\langle P_1, \cdots, P_r\rangle$, initialized to expressions $E_1, \cdots, E_r$. These expressions are called *initialization* expressions, since they initialize the instance. Upon initialization of instance $C^p(E_1, \cdots, E_r)$, expressions $E_1, \cdots, E_r$ are evaluated from left to right and the results are bound to the corresponding instance variables $y_1, \cdots, y_r$. All other instance variables are initialized to *nil*. Then the initial method is called. Upon instantiation of the instance of cluster class $C^c\langle P_1, \cdots, P_r\rangle$, expression parameters $P_1, \cdots, P_r$ are syntactically substituted by $E_1, \cdots, E_r$. Then all constituents of the behaviour specification are initialized.

The second part of a system specification is an empty list. The meaning of this part will become clear in Section 3.4.2. [2]

The third part is a system $Sys^p$ which contains the set of all process classes and cluster classes, and the last part is a system *Sys* of non-primitive classes of data objects. Formally, we define the set of all system specifications *SSpecifications*, with typical elements

[2]In practice we often omit the empty list and write $\langle BSpec, Sys^p, Sys\rangle$ in stead of $\langle BSpec, \langle\rangle, Sys^p, Sys\rangle$.

$SSpec, \cdots$, as

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

## 3.3 Context Conditions

In this section we will describe the *syntactic requirements*, often called context conditions, which have to be satisfied by specification $\langle BSpec, \langle\rangle, Sys^p, Sys \rangle$ to be *valid*. The set of conditions is partitioned into the following three groups:

**Conditions concerning** $Sys$

(1.) All context conditions defined in Section 2.3 have to be satisfied.

**Conditions concerning** $Sys^p$

(2.) All class names in $Sys^p$ are different.

(3.) All instance variables in a class definition are different.

(4.) All method names within a single process class are different.

(5.) All parameters and local variables defined in a method definition are different.

(6.) Every variable used in a method body is either an instance variable of the corresponding process class, a method parameter, or a local method-variable.

(7.) No method body has expression *self* as its constituent.

(8.) An expression contained in an initial method call does not contain any local variables or *self* expressions.

(9.) All instance variables specified between angle brackets in a process class definition are different. Further, each of these variables is member of the set of all instance variables of the class.

(10.) The set of communication channels as well as the message interface defined in a class definition conform to the corresponding instance methods or behaviour specification. (This condition is formally defined in Section 3.4.2.)

(11.) For every method call statement there exists a corresponding method definition.

(12.) For every $C^{pc}(PE_1, \cdots, PE_r)$ used as part of the behaviour expression of some cluster class, there exists a corresponding class definition. Further, $C^{pc}(PE_1, \cdots, PE_r)$ does not contain any variables or *self* expressions.

(13.) Every expression parameter used in a parameterized behaviour expression is defined as expression parameter of the corresponding cluster class.

(14.) Parameterized behaviour specifications of cluster classes are not defined (mutual) recursively.

**Conditions concerning the combination of *BSpec*, *Sys*$^p$**

(15.) For every $C^{pc}(E_1, \cdots, E_r)$ used as part of *BSpec*, there exists a corresponding class definition in *Sys*$^p$. Further no expression $E_i$ contains variables or *self*.

In the rest of this report it will be assumed that *SSpecifications* denotes the set of ***valid*** system specifications.

## 3.4 A Computational Interleaving Semantics

### 3.4.1 Informal Explanation

The process-oriented part of POOSL will be formalized by means of a ***computational interleaving semantics.*** The semantics is specified by a ***labeled-transition system.*** A labeled-transition system is very similar to a "normal" transition system as described in Section 2.4.1. It is represented by an ensemble $(Conf^p, Act, \{\xrightarrow{a} \mid a \in Act\})$ where $Conf^p$ is a set of configurations, $Act$ is a set of (atomic) ***actions***, and $\{\xrightarrow{a} \mid a \in Act\}$ is a set of ***labeled-transition relations.*** If $\langle S, I\rangle$ and $\langle S', I'\rangle$ are configurations, then the intuitive meaning of $\langle S', I'\rangle \xrightarrow{a} \langle S', I'\rangle$ is that system $S$ with information $I$ can lead to system $S'$ and information $I'$ by performing action $a$.

The execution of a system of collaborating instances is modeled as the ***interleaving*** of all atomic actions, that is, as a sequential execution of these actions.

### 3.4.2 Definitions

The labeled-transition system will be based upon configurations of the form

$$\langle BSpec^e, \langle\langle\sigma_1, ps_1, \tau_1\rangle, \cdots, \langle\sigma_n, ps_n, \tau_n\rangle\rangle, Sys^p, Sys\rangle$$

Together $BSpec^e$, $Sys^p$, and $Sys$, form the syntactic part of the configuration. $BSpec^e$ is an ***extended behaviour specification.*** The set $BSpecifications^e$ denotes the set of all extended behaviour-specifications and is defined as

$$\begin{array}{rcl} BSpec^e & ::= & C^p(E_1, \cdots, E_r) \\ & \mid & C^c(E_1, \cdots, E_r) \\ & \mid & [S^{p,e}]_{C^p(E_1,\cdots,E_r)} \\ & \mid & [BSpec^e]_{C^c(E_1,\cdots,E_r)} \end{array}$$

$| \; BSpec^e_1 \parallel BSpec^e_2$
$| \; BSpec^e \setminus L$
$| \; BSpec^e[f]$

$C^p(E_1, \cdots, E_r)$ and $C^c(E_1, \cdots, E_r)$ denote an instance of some process class $C^p\langle y_1, \cdots, y_r\rangle$ and some cluster class $C^c\langle P_1, \cdots, P_r\rangle$ respectively, initialized to $E_1, \cdots, E_r$. $[S^{p,e}]_{C^p(E_1,\cdots,E_r)}$ denotes that statement $S^{p,e}$ still has to be executed by process instance $C^p(E_1, \cdots, E_r)$. $[BSpec^e]_{C^c(E_1,\cdots,E_r)}$ indicates that behaviour expression $BSpec^e$ remains to be executed by cluster $C^c(E_1, \cdots, E_r)$. The other constructs denote *parallel composition*, *channel hiding*, and *channel renaming* respectively.

$\langle\langle\sigma_1, ps_1, \tau_1\rangle , \cdots , \langle\sigma_n, ps_n, \tau_n\rangle\rangle$, is the *information* part of the configuration. $\langle\langle\sigma_1, ps_1, \tau_1\rangle , \cdots , \langle\sigma_n, ps_n, \tau_n\rangle\rangle$ is a list of *process environments*, one for each *initialized process* object of $BSpec^e$. A process environment $\langle\sigma, ps, \tau\rangle$ is composed of a global state $\sigma$, a local process stack $ps$, and a type $\tau$.

We let *Env* denote the set of all process environments and let it range over $env, \cdots$. *Env* is defined as

$$Env = \Sigma \times PStack \times Type$$

Here, $\Sigma$ and *Type* are as defined in Section 2.4.2. *PStack* denotes the set of process stacks which is a subset of *Stack* as defined in Section 2.4.2. *PStack* ranges over $ps, \cdots$ and is defined as

$$PStack = (\{proc\} \times (LVar \hookrightarrow DObj))^*$$

$Env^*$ will be used to denote the set of *all* lists of process environments. It ranges over $envs, \cdots$. Further, we will write $envs_1 \cdot envs_2$ for the concatenation of lists $envs_1$ and $envs_2$, and we write $\langle\rangle$ for the empty list.

We will now define the set $Conf^p$ of configurations. $Conf^p \subseteq BSpecifications^e \times Env^* \times Systems^p \times Systems$ is inductively defined by means of the following set of axioms and rules:

1. $\langle C^p(E_1, \cdots, E_r), \langle\rangle, Sys^p, Sys\rangle \in Conf^p$
   if there exists a process class $C^p\langle y_1, \cdots, y_r\rangle$ in $Sys^p$ and if no $E_i$ contains variables or *self*.

   The condition is derived from context condition (15.) given in Section 3.3.

2. $\langle C^c(E_1, \cdots, E_r), \langle\rangle, Sys^p, Sys\rangle \in Conf^p$
   if there exists a cluster class $C^c\langle P_1, \cdots, P_r\rangle$ in $Sys^p$ and if no $E_i$ contains variables or *self*.

   The condition is again derived from context condition (15.) given in Section 3.3.

3. $\langle [S^{p,e}]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \in Conf^p$
if $C^p(y_1, \cdots, y_r)$ is a process class in $Sys^p$, if no $E_i$ contains variables or *self*, and if $AASort(S^{p,e}) \subseteq AASort(C^p, Sys^p)$

The first part of the condition is derived from context condition (15.). The second part requires that $S^{p,e}$ does not contain any send or receive statements, that do not conform to the message interface of the process object. Function *AASort*, defined later in this section, calculates the set of all associated abstract communication actions.

4. $\dfrac{\langle BSpec^e, envs, Sys^p, Sys\rangle \in Conf^p}{\langle [BSpec^e]_{C^c(E_1,\cdots,E_r)}, envs, Sys^p, Sys\rangle \in Conf^p}$
if $C^c(P_1, \cdots, P_r)$ is a cluster class in $Sys^p$ with behaviour expression $BSpec^p$, if no $E_i$ contains variables or *self*, and if $Reset(BSpec^e) \equiv BSpec^p[\![E_1/P_1, \cdots, E_r/P_r]\!]$

Again, the first part of the condition is derived from context condition (15.). The second part ensures that $BSpec^e$ is built from the same instances, in the same way, as $BSpec^p[\![E_1/P_1, \cdots, E_r/P_r]\!]$. Intuitively this means that the latter behaviour specification is a "decent" descendant of the former one. $BSpec^p[\![E_1/P_1, \cdots, E_r/P_r]\!]$ denotes the syntactic substitution of each $P_i$ by $E_i$ in $BSpec^p$. The precise definition is given later in this section. Function *Reset* calculates, given an (extended) behaviour specification, the corresponding initial specification. *Reset* is defined later in this section.

5. $\dfrac{\langle BSpec_1^e, envs_1, Sys^p, Sys\rangle \in Conf^p \;,\; \langle BSpec_2^e, envs_2, Sys^p, Sys\rangle \in Conf^p}{\langle BSpec_1^e \parallel BSpec_2^e, envs_1 \cdot envs_2, Sys^p, Sys\rangle \in Conf^p}$

6. $\dfrac{\langle BSpec^e, envs, Sys^p, Sys\rangle \in Conf^p}{\langle BSpec^e \setminus L, envs, Sys^p, Sys\rangle \in Conf^p}$

7. $\dfrac{\langle BSpec^e, envs, Sys^p, Sys\rangle \in Conf^p}{\langle BSpec^e[f], envs, Sys^p, Sys\rangle \in Conf^p}$

$Sys^p$ and $Sys$ are assumed to satisfy context conditions (1.) $\cdots$ (15.) of Section 3.3.

We can now see why a system specification $\langle BSpec, \langle\rangle, Sys^p, Sys\rangle$ contains the empty-list part. The seemingly superfluous list, allows us to consider a system specification as a special kind of configuration, thereby simplifying our theory. By induction it is easy to show that indeed $SSpecifications \subseteq Conf^p$.

As explained above, $S^{p,e}$ denotes a statement that still has to be executed by some process object. $S^{p,e}$, an element of set $Stat^{p,e}$, is defined as

$$
\begin{array}{rcl}
S^{p,e} & ::= & S \\
& | & ch!m(E_1,\cdots,E_n) \\
& | & ch?m(p_1,\cdots,p_m \mid E) \\
& | & m(E_1,\cdots,E_n)(p_1,\cdots,p_m) \\
& | & S_1^{p,e};\ S_2^p \\
& | & S_1^p \text{ or } S_2^p \\
& | & [E]S^p \\
& | & \text{if } E \text{ then } S_1^p \text{ else } S_2^p \text{ fi} \\
& | & \text{do } E \text{ then } S^p \text{ od} \\
& | & S_1^{p,e} >> S_2^p \\
& | & \downarrow^{m(p_1,\cdots,p_m)} S^{p,e} \\
& | & \surd
\end{array}
$$

Here, $\downarrow^{m(p_1,\cdots,p_m)} S^{p,e}$ indicates that the method body of method $m$ is being executed and that after successful termination of this execution the process has to resume its execution at the place of the $\downarrow^{m(p_1,\cdots,p_m)}$, after the output parameters are bound to variables $p_1,\cdots,p_m$ and the local method variables have been popped from the stack. This statement is introduced to facilitate our semantics. $\surd$ denotes successful process termination.

Now that we have defined the set $Conf^p$ of configurations, we will show how the set $Act$ of actions looks like. We will distinguish the following three kinds of actions:

1. The *internal* action, also known as the silent action, which is denoted as $\tau$. This action reflects an internal computation which cannot be observed by the system environment.

2. *Send* actions of the form $ch!m[data]$ indicating that the system can send message $m$, together with data $data$, on channel $ch$.

3. *Receive* actions of the form $ch?m[data]$ indicating that the system is willing to receive message $m$ with data $data$ from channel $ch$.

When two processes exchange a message, a *deepCopy* of every message parameter is passed from the sending process to the receiving one. The *data* part of the above send and receive actions therefore consists of a list of deepCopies, one for each message parameter. Every deepCopy will be represented by a minimal *Sys*-structure (see Section 2.4.5), and every *data* by a list of such structures.

For each $n \in \mathbb{N}$ we will let $Struc^n_{Sys,min}$ denote the set of lists of minimal *Sys*-structures consisting of $n$ elements. The set of all lists of minimal *Sys*-structures will be denoted by $Struc^*_{Sys,min}$ and will range over $data,\cdots$.

The set $Act$ of actions can now be defined as

$$Act = \mathcal{L} \cup \{\tau\}$$

where $\mathcal{L}$, the set of all communication actions, is

$$\begin{aligned}\mathcal{L} = &\{ch?m[data] \mid ch \in Chan \;\wedge\; m \in MName \;\wedge\; data \in Struc^*_{Sys,min}\} \cup \\ &\{ch!m[data] \mid ch \in Chan \;\wedge\; m \in MName \;\wedge\; data \in Struc^*_{Sys,min}\}\end{aligned}$$

Elements of $\mathcal{L}$ will be denoted as $l, \cdots$, and elements of $Act$ as $a, \cdots$.

We now define what we mean by functions *Abstract Action Sort* (*AASort*) and *Channel Sort* (*ChSort*). Both functions are defined on a number of constructs. In general the *AASort* of a construct is the set of all associated abstract communication actions. *ChSort* calculates the set of associated channels. For configurations both functions have a particular meaning. The *AASort* of a configuration is the set of abstract communication actions corresponding to the *communication actions* which can ever be performed in future by that configuration. The *ChSort* calculates the set of *channels* which can possibly ever be used for communication.

**Definition 3.1**
The abstract action sort is defined on $Stat^{p,e}$, on $BSpecifications^e \times Systems^p$, on $BSpecifications^p \times Systems^p$ and on $CName^{pc} \times Systems^p$ as follows:

$$\begin{array}{lcl}
AASort(S) & = & \varnothing \\
AASort(ch!m(E_1, \cdots, E_n)) & = & \{ch!m[\underline{n}]\} \\
AASort(ch?m(p_1, \cdots, p_n \mid E) & = & \{ch?m[\underline{n}]\} \\
AASort(m(E_1, \cdots, E_m)(p_1, \cdots, p_n)) & = & \varnothing \\
AASort(S_1^{p,e};\; S_2^p) & = & AASort(S_1^{p,e}) \cup AASort(S_2^p) \\
AASort(S_1^p \text{ or } S_2^p) & = & AASort(S_1^p) \cup AASort(S_2^p) \\
AASort([E]S^p) & = & AASort(S^p) \\
AASort(\text{if } E \text{ then } S_1^p \text{ else } S_2^p \text{ fi}) & = & AASort(S_1^p) \cup AASort(S_2^p) \\
AASort(\text{do } E \text{ then } S^p \text{ od}) & = & AASort(S^p) \\
AASort(S_1^{p,e} >> S_2^p) & = & AASort(S_1^{p,e}) \cup AASort(S_2^p) \\
AASort(\downarrow^{m(p_1, \cdots, p_n)} S^{p,e}) & = & AASort(S^{p,e}) \\
AASort(\surd) & = & \varnothing
\end{array}$$

$$\begin{array}{lcl}
AASort(C^p(E_1, \cdots, E_r), Sys^p) & = & AASort(C^p, Sys^p) \\
AASort(C^c(E_1, \cdots, E_r), Sys^p) & = & AASort(C^c, Sys^p) \\
AASort([S^{p,e}]_{C^p(E_1, \cdots, E_r)}, Sys^p) & = & AASort(C^p, Sys^p) \\
AASort([BSpec^e]_{C^c(E_1, \cdots, E_r)}, Sys^p) & = & AASort(C^c, Sys^p) \\
AASort(BSpec_1^e \parallel BSpec_2^e, Sys^p) & = & AASort(BSpec_1^e, Sys^p) \cup \\
 & & AASort(BSpec_2^e, Sys^p)
\end{array}$$

$$\begin{array}{ll}
AASort(BSpec^e \setminus L, Sys^p) & = \{l^a \in AASort(BSpec^e, Sys^p) \mid Chan(l^a) \notin L\} \\
AASort(BSpec^e[f], Sys^p) & = \{f(l^a) \mid l^a \in AASort(BSpec^e, Sys^p)\}
\end{array}$$

$$\begin{array}{ll}
AASort(C^p(PE_1, \cdots, PE_r), Sys^p) = & AASort(C^p, Sys^p) \\
AASort(C^c(PE_1, \cdots, PE_r), Sys^p) = & AASort(C^c, Sys^p) \\
AASort(BSpec_1^p \parallel BSpec_2^p, Sys^p) = & AASort(BSpec_1^p, Sys^p) \cup \\
 & AASort(BSpec_2^p, Sys^p) \\
AASort(BSpec^p \setminus L, Sys^p) = & \{l^a \in AASort(BSpec^p, Sys^p) \mid Chan(l^a) \notin L\} \\
AASort(BSpec^p[f], Sys^p) = & \{f(l^a) \mid l^a \in AASort(BSpec^p, Sys^p)\}
\end{array}$$

$$AASort(C^{pc}, Sys^p) = \begin{cases} \{l_1^a, \cdots, l_m^a\} & \text{if } Sys^p \equiv \langle CD_1^p \cdots CD^p \cdots CD_s^p \rangle \text{ ,} \\ & CD^p \equiv \cdots \text{ class } C^{pc} \langle \cdots \rangle \\ & \cdots \text{ message interface } l_1^a \cdots l_m^a \ \cdots \\ \emptyset & \text{otherwise} \end{cases}$$

□

**Definition 3.2**
The channel sort is defined on $Stat^{p,e}$, on $BSpecifications^e \times Systems^p$, on $BSpecifications^p \times Systems^p$ and on $CName^{pc} \times Systems^p$ as follows:

$$\begin{array}{ll}
ChSort(S^{p,e}) & = \{Chan(l^a) \mid l^a \in AASort(S^{p,e})\} \\
ChSort(BSpec^e, Sys^p) & = \{Chan(l^a) \mid l^a \in AASort(BSpec^e, Sys^p)\} \\
ChSort(BSpec^p, Sys^p) & = \{Chan(l^a) \mid l^a \in AASort(BSpec^p, Sys^p)\} \\
ChSort(C^{pc}, Sys^p) & = \{Chan(l^a) \mid l^a \in AASort(C^{pc}, Sys^p)\}
\end{array}$$

□

**Definition 3.3**
Let $conf^p = \langle BSpec^e, envs, Sys^p, Sys \rangle$ be a configuration. The $AASort$ and $ChSort$ of $conf^p$ are given by

$$\begin{array}{ll}
AASort(conf^p) & = AASort(BSpec^e, Sys^p) \\
ChSort(conf^p) & = ChSort(BSpec^e, Sys^p)
\end{array}$$

□

$Chan(l^a)$, used in Definition 3.1, retrieves the channel from abstract communication action $l^a$. $Chan$ is also defined on $\mathcal{L}$

$$\begin{array}{ll}
Chan(ch!m[data]) & = ch \\
Chan(ch?m[data]) & = ch \\
Chan(ch!m[\underline{i}]) & = ch \\
Chan(ch?m[\underline{i}]) & = ch
\end{array}$$

The application of a channel renaming function $f$ to an (abstract) communication action yields a new action which channel is relabelled as dictated by $f$. For channel renaming function $f$ we define

$$\begin{aligned}
f(ch!m[data]) &= f(ch)!m[data] \\
f(ch?m[data]) &= f(ch)?m[data] \\
f(ch!m[\underline{i}]) &= f(ch)!m[\underline{i}] \\
f(ch?m[\underline{i}]) &= f(ch)?m[\underline{i}]
\end{aligned}$$

Another convenient function is $Abs$. This function takes a communication action from $\mathcal{L}$ and calculates the corresponding abstraction. $Abs$ is defined as

$$\begin{aligned}
Abs(ch!m[data]) &= ch!m[\mid data \mid] \text{[3]} \\
Abs(ch?m[data]) &= ch?m[\boxed{\mid data \mid}]
\end{aligned}$$

On $\mathcal{L}$ and $\mathcal{L}^a$, we define the complement operator $\bar{\cdot}$ which maps a send action to a corresponding receive action and vice versa.

$$\begin{aligned}
\overline{ch!m[data]} &= ch?m[data] \\
\overline{ch?m[data]} &= ch!m[data] \\
\overline{ch!m[\underline{i}]} &= ch?m[\underline{i}] \\
\overline{ch?m[\underline{i}]} &= ch!m[\underline{i}]
\end{aligned}$$

The next definitions concern syntactic substitution of expression parameters in parameterized expression, in parameterized statements and in parameterized behaviour specifications. We define the set $SSubst$ of syntactic substitution functions as

$$SSubst = EPar \to EPar \cup Exp$$

and let it range over $g, \cdots$.

We shall often write $E_1/P_1, \cdots, E_r/P_r$ to denote a function $g$ for which $g(P_i) = E_i$ for $i = 1, \cdots, r$ and $g(P) = P$ if $P \neq P_i$ for all $i = 1, \cdots, r$.

**Definition 3.4**
Syntactic substitution of expression parameters is defined as:

[3]Here $\mid data \mid$ denotes the amount of elements of list $data$.

$$\begin{array}{lcl}
P[\![g]\!] & \equiv & g(P) \\
E[\![g]\!] & \equiv & E \\
(PE\ m(PE_1, \cdots, PE_n))[\![g]\!] & \equiv & (PE[\![g]\!])\ m(PE_1[\![g]\!], \cdots, PE_n[\![g]\!]) \\
(PS; PE)[\![g]\!] & \equiv & (PS[\![g]\!]); (PE[\![g]\!]) \\
(x := PE)[\![g]\!] & \equiv & x := (PE[\![g]\!]) \\
(u := PE)[\![g]\!] & \equiv & u := (PE[\![g]\!]) \\
(PS_1; PS_2)[\![g]\!] & \equiv & (PS_1[\![g]\!]); (PS_2[\![g]\!]) \\
(\text{if } PE \text{ then } PS_1 \text{ else } PS_2 \text{ fi})[\![g]\!] & \equiv & \text{if } PE[\![g]\!] \text{ then } PS_1[\![g]\!] \text{ else } PS_2[\![g]\!] \text{ fi} \\
(\text{do } PE \text{ then } PS \text{ od})[\![g]\!] & \equiv & \text{do } PE[\![g]\!] \text{ then } PS[\![g]\!] \text{ od} \\
C^{pc}(PE_1, \cdots, PE_r)[\![g]\!] & \equiv & C^{pc}(PE_1[\![g]\!], \cdots, PE_r[\![g]\!]) \\
(BSpec_1^p \parallel BSpec_2^p)[\![g]\!] & \equiv & (BSpec_1^p[\![g]\!]) \parallel (BSpec_2^p[\![g]\!]) \\
(BSpec^p \setminus L)[\![g]\!] & \equiv & (BSpec^p[\![g]\!]) \setminus L \\
(BSpec^p[f])[\![g]\!] & \equiv & (BSpec^p[\![g]\!])[f]
\end{array}$$

□

The definition of the set of configuration *Conf*$^p$ uses function *Reset*. There it was applied to (extended) behaviour specifications, but we will define it on configurations too. The *Reset* of a behaviour specification is the corresponding *initial* behaviour specification, i.e., the behaviour specification describing the (part of the) system in its initial state, i.e., before initialization has taken place. The *Reset* of a configuration is the corresponding initial *specification*, i.e., the specification from which it is (probably) derived. We say probably, because not every configuration has a specification as ancestor. However, later we will prove that *if* a configuration *has* an ancestor specification, *then* this must be the configurations' *Reset*.

**Definition 3.5**
The *Reset* of an extended behaviour specification is inductively defined as

$$\begin{array}{ll}
Reset(C^p(E_1, \cdots, E_n)) & \equiv C^p(E_1, \cdots, E_n) \\
Reset([S^{p,e}]_{C^p(E_1, \cdots, E_n)}) & \equiv C^p(E_1, \cdots, E_n) \\
Reset(C^c[E_1, \cdots, E_n]) & \equiv C^c(E_1, \cdots, E_n) \\
Reset([BSpec^e]_{C^c(E_1, \cdots, E_n)}) & \equiv C^c(E_1, \cdots, E_n) \\
Reset(BSpec_1^e \parallel BSpec_2^e) & \equiv Reset(BSpec_1^e) \parallel Reset(BSpec_2^e) \\
Reset(BSpec^e \setminus L) & \equiv Reset(BSpec^e) \setminus L \\
Reset(BSpec^e[f]) & \equiv Reset(BSpec^e)[f]
\end{array}$$

□

**Definition 3.6**
The *Reset* of a configuration $conf^p = \langle BSpec^e, envs, Sys^p, Sys \rangle$ is defined as

$$Reset(conf^p) = \langle Reset(BSpec^e), \langle\rangle, Sys^p, Sys \rangle$$

□

We conclude this section by considering context condition (10) of Section 3.3. We defined this condition as

(10) The set of communication channels as well as the message interface defined in a class definition conform to the corresponding instance methods or behaviour specification.

Since the precise meaning of this definition is not clear at all, and since the condition is important for our semantics to be correct, we will formally rephrase it. We will do this in terms of functions *AASort* and *ChSort* defined in this section.

(10) Let $CD^p$ be a class definition of $Sys^p$. Then

(i) if $CD^p \equiv$ process class $C^p\langle\cdots\rangle$ $\cdots$ communication channels $ch_1 \cdots ch_p$ message interface $l_1^a \cdots l_m^a$ $\cdots$ instance methods $MD_1^p \cdots MD_k^p$, such that for each $i = 1 \cdots k$ $MD_i^p \equiv \cdots S_i^p$, then

$$\bigcup_{i=1}^{k} AASort(S_i^p) = \{l_1^a, \cdots, l_m^a\}$$
$$\bigcup_{i=1}^{k} ChSort(S_i^p) = \{ch_1, \cdots, ch_p\}$$

(ii) if $CD^p \equiv$ cluster class $C^c\langle\cdots\rangle$ $\cdots$ communication channels $ch_1 \cdots ch_p$ message interface $l_1^a \cdots l_m^a$ behaviour specification $BSpec^p$, then

$$AASort(BSpec^p, Sys^p) = \{l_1^a, \cdots, l_m^a\}$$
$$ChSort(BSpec^p, Sys^p) = \{ch_1, \cdots, ch_p\}$$

### 3.4.3 The Labeled-Transition System

In this section we will define the set of labeled transitions $\{\xrightarrow{a} \mid a \in Act\}$. It is defined by the following axioms and rules:

#### Axioms

1'. *Internal computation*

$$\langle [S]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{\tau}$$
$$\langle [\surd]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle$$

if

$$\langle \underline{\beta}, \sigma', ps', \tau', Sys\rangle \in \mathcal{M}(\langle S, \sigma, ps, \tau, Sys\rangle)$$

2'. *Message send*

$$\langle [ch!m(E_1, \cdots, E_n)]_{C^p(E_1',\cdots,E_r')}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{ch!m[data]}$$
$$\langle [\surd]_{C^p(E_1',\cdots,E_r')}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle$$

if

$$\begin{array}{l}\langle \underline{\beta_1}, \sigma_1, ps_1, \tau_1, Sys\rangle \in \mathcal{M}(\langle E_1, \sigma, ps, \tau, Sys\rangle)\\ \langle \underline{\beta_2}, \sigma_2, ps_2, \tau_2, Sys\rangle \in \mathcal{M}(\langle E_2, \sigma_1, ps_1, \tau_1, Sys\rangle)\\ \qquad\qquad\vdots\\ \langle \underline{\beta_n}, \sigma', ps', \tau', Sys\rangle \in \mathcal{M}(\langle E_n, \sigma_{n-1}, ps_{n-1}, \tau_{n-1}, Sys\rangle)\end{array}$$

and

$$\langle \beta_1, \sigma', \tau'\rangle, \cdots, \langle \beta_n, \sigma', \tau'\rangle \text{ are } Sys\text{-structures}$$

and where

$$data = \langle copy(\langle \beta_1, \sigma', \tau'\rangle), \cdots, copy(\langle \beta_n, \sigma', \tau'\rangle)\rangle$$

3'. *Message reception*

$$\langle [ch?m(p_1, \cdots, p_n \mid E)]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{ch?m[data]} \langle [\surd]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle$$

if

$$\begin{array}{l}\mid ps \mid > 0\\ data = \langle\langle \beta_1, \sigma_1, \tau_1\rangle, \cdots, \langle \beta_n, \sigma_n, \tau_n\rangle\rangle \in Struc^n_{Sys,min}\\ \gamma \in \{true, bunk\}\end{array}$$

and where

$$\begin{cases} \langle \beta'_1, \sigma'_1, \tau'_1\rangle = relabel_{+MaxId(\sigma)}(\langle \beta_1, \sigma_1, \tau_1\rangle)\\ \bar{\sigma}_1 = \sigma \cup \sigma'_1 \ , \ \bar{\tau}_1 = \tau \cup \tau'_1\end{cases}$$
$$\begin{cases} \langle \beta'_2, \sigma'_2, \tau'_2\rangle = relabel_{+MaxId(\bar{\sigma}_1)}(\langle \beta_2, \sigma_2, \tau_2\rangle)\\ \bar{\sigma}_2 = \bar{\sigma}_1 \cup \sigma'_2 \ , \ \bar{\tau}_2 = \bar{\tau}_1 \cup \tau'_2\end{cases}$$
$$\vdots$$
$$\begin{cases} \langle \beta'_n, \sigma'_n, \tau'_n\rangle = relabel_{+MaxId(\bar{\sigma}_{n-1})}(\langle \beta_n, \sigma_n, \tau_n\rangle)\\ \bar{\sigma}_n = \bar{\sigma}_{n-1} \cup \sigma'_n \ , \ \bar{\tau}_n = \bar{\tau}_{n-1} \cup \tau'_n\end{cases}$$

$$\tilde{\sigma}_1, \chi_1 = \begin{cases} \bar{\sigma}_n\{\bar{\sigma}_n(proc)\{\beta'_1/p_1\}/proc\}, top(ps)(2) & \text{if } p_1 \in IVar\\ \bar{\sigma}_n, top(ps)(2)\{\beta'_1/p_1\} & \text{if } p_1 \in LVar\end{cases}$$
$$\tilde{\sigma}_2, \chi_2 = \begin{cases} \tilde{\sigma}_1\{\tilde{\sigma}_1(proc)\{\beta'_2/p_2\}/proc\}, \chi_1 & \text{if } p_2 \in IVar\\ \tilde{\sigma}_1, \chi_1\{\beta'_2/p_2\} & \text{if } p_2 \in LVar\end{cases}$$
$$\vdots$$
$$\tilde{\sigma}_n, \chi_n = \begin{cases} \tilde{\sigma}_{n-1}\{\tilde{\sigma}_{n-1}(proc)\{\beta'_n/p_n\}/proc\}, \chi_{n-1} & \text{if } p_n \in IVar\\ \tilde{\sigma}_{n-1}, \chi_{n-1}\{\beta'_n/p_n\} & \text{if } p_n \in LVar\end{cases}$$

$$\langle \underline{\gamma}, \sigma', ps', \tau', Sys\rangle \in \mathcal{M}(\langle E, \tilde{\sigma}_n, push(\langle proc, \chi_n\rangle, pop(ps)), \bar{\tau}_n, Sys\rangle)$$

4'. *Method call*

$$\langle [m(E_1,\cdots,E_m)(p_1,\cdots,p_n)]_{C^p(E'_1,\cdots,E'_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{\tau}$$
$$\langle [\downarrow^{m(p_1,\cdots,p_n)} S^p]_{C^p(E'_1,\cdots,E'_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle$$

if

$$Sys^p \equiv \langle CD^p_1 \cdots CD^p \cdots CD^p_q\rangle$$
$$CD^p \equiv \text{process class } C^p\langle\cdots\rangle \ \cdots \ \text{instance methods } MD^p_1 \cdots MD^p \cdots MD^p_s$$
$$MD^p \equiv \mathrm{m}(u_1,\cdots,u_m)\ (v_1,\cdots,v_n)\ \mid w_1\cdots w_o \mid S^p$$

$$\langle \underline{\beta_1}, \sigma_1, ps_1, \tau_1, Sys\rangle \in \mathcal{M}(\langle E_1, \sigma, ps, \tau, Sys\rangle)$$
$$\langle \underline{\beta_2}, \sigma_2, ps_2, \tau_2, Sys\rangle \in \mathcal{M}(\langle E_2, \sigma_1, ps_1, \tau_1, Sys\rangle)$$
$$\vdots$$
$$\langle \underline{\beta_m}, \sigma_m, ps_m, \tau_m, Sys\rangle \in \mathcal{M}(\langle E_n, \sigma_{n-1}, ps_{n-1}, \tau_{n-1}, Sys\rangle)$$

and where

$$\sigma' = \sigma_m \text{ and } \tau' = \tau_m$$
$$ps' = push(\langle proc, \chi\rangle, ps_m)$$
$$\chi(u_i) = \beta_i$$
$$\chi(v_j) = nil$$
$$\chi(w_k) = nil$$

5'. *Conditional, first branch*

$$\langle [\text{if } E \text{ then } S^p_1 \text{ else } S^p_2 \text{ fi}]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{\tau}$$
$$\langle [S^p_1]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle$$

if

$$\langle \underline{\gamma}, \sigma', ps', \tau', Sys\rangle \in \mathcal{M}(\langle E, \sigma, ps, \tau, Sys\rangle) \text{ for some } \gamma \in \{true, bunk\}$$

6'. *Conditional, alternative branch*

$$\langle [\text{if } E \text{ then } S^p_1 \text{ else } S^p_2 \text{ fi}]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{\tau}$$
$$\langle [S^p_2]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle$$

if

$$\langle \underline{\gamma}, \sigma', ps', \tau', Sys\rangle \in \mathcal{M}(\langle E, \sigma, ps, \tau, Sys\rangle) \text{ for some } \gamma \in \{false, bunk\}$$

7'. *Do-statement*

$$\langle [\text{do}\, E\, \text{then}\, S^p\, \text{od}]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{\tau}$$
$$\langle [\text{if}\, E\, \text{then}(S^p;\ \text{do}\, E\, \text{then}\, S^p\, \text{od})\, \text{else}\, \underline{nil}\, \text{fi}]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle$$

8'. *Process initialization*

$$\langle C^p(E_1,\cdots,E_r), \langle\rangle, Sys^p, Sys\rangle \xrightarrow{\tau}$$
$$\langle [m(E_1',\cdots,E_q')()]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma', ps_r, \tau_r\rangle\rangle, Sys^p, Sys\rangle$$

if

$$Sys^p \equiv \langle CD_1^p \cdots CD^p \cdots CD_s^p\rangle$$
$$CD^p \equiv \text{process class } C^p\langle y_1,\cdots,y_r\rangle \text{ instance variable names } x_1\cdots x_n$$
$$\cdots \text{ initial method call } m(E_1',\cdots,E_q')() \cdots$$

$$\langle \underline{\beta_1}, \sigma_1, ps_1, \tau_1, Sys\rangle \in \mathcal{M}(\langle E_1, \sigma, ps, \tau, Sys\rangle)$$
$$\langle \underline{\beta_2}, \sigma_2, ps_2, \tau_2, Sys\rangle \in \mathcal{M}(\langle E_2, \sigma_1, ps_1, \tau_1, Sys\rangle)$$
$$\vdots$$
$$\langle \underline{\beta_r}, \sigma_r, ps_r, \tau_r, Sys\rangle \in \mathcal{M}(\langle E_r, \sigma_{r-1}, ps_{r-1}, \tau_{r-1}, Sys\rangle)$$

and where

$$\begin{aligned}
Dom(\sigma) &= \{proc\}\\
\sigma(proc) &= \varnothing\\
ps &= \langle\rangle\\
\tau &= \varnothing\\
\sigma' &= \sigma_r\{\phi/proc\}\\
Dom(\phi) &= \{x_1,\cdots,x_n\}\\
\phi(y_i) &= \beta_i\\
\phi(x_j) &= nil \text{ if } y_i \not\equiv x_j \text{ for all } y_i
\end{aligned}$$

9'. *Cluster initialization*

$$\langle C^c(E_1,\cdots,E_r), \langle\rangle, Sys^p, Sys\rangle \xrightarrow{\tau} \langle [BSpec^p[\![g]\!]]_{C^c(E_1,\cdots,E_r)}, \langle\rangle, Sys^p, Sys\rangle$$

if

$$Sys^p \equiv \langle CD_1^p \cdots CD^p \cdots CD_r^p\rangle$$
$$CD^p \equiv \text{cluster class } C^c\langle P_1,\cdots,P_r\rangle \cdots \text{ behaviour specification } BSpec^p$$

and where

$$g = E_1/P_1,\cdots,E_r/P_r$$

## Rules

a'. *Sequential composition 1*

$$\frac{\begin{array}{l}\langle [S_1^{p,e}]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [S_1^{p,e'}]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle\end{array}}{\begin{array}{l}\langle [S_1^{p,e};\ S_2^p]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [S_1^{p,e'};\ S_2^p]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle\end{array}}$$

if

$$S^{p,e'} \not\equiv \surd$$

b'. *Sequential composition 2*

$$\frac{\begin{array}{l}\langle [S_1^{p,e}]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [\surd]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle\end{array}}{\begin{array}{l}\langle [S_1^{p,e}\ ;\ S_2^p]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [S_2^p]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle\end{array}}$$

c'. *Choice 1*

$$\frac{\begin{array}{l}\langle [S_1^p]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [S^{p,e}]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle\end{array}}{\begin{array}{l}\langle [S_1^p \text{ or } S_2^p]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [S^{p,e}]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle\end{array}}$$

d'. *Choice 2*

$$\frac{\begin{array}{l}\langle [S_2^p]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [S^{p,e}]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle\end{array}}{\begin{array}{l}\langle [S_1^p \text{ or } S_2^p]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [S^{p,e}]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle\end{array}}$$

e'. *Guarded command*

$$\frac{\begin{array}{l}\langle [S^p]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [S^{p,e}]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma'', ps'', \tau''\rangle\rangle, Sys^p, Sys\rangle\end{array}}{\begin{array}{l}\langle [[E]S^p]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [S^{p,e}]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma'', ps'', \tau''\rangle\rangle, Sys^p, Sys\rangle\end{array}}$$

if

$$\langle \underline{\gamma}, \sigma', ps', \tau', Sys\rangle \in \mathcal{M}(\langle E, \sigma, ps, \tau, Sys\rangle) \text{ with } \gamma \in \{\mathit{true}, \mathit{bunk}\}$$

f'. *Method execution*

$$\frac{\begin{array}{l}\langle [S^{p,e}]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [S^{p,e'}]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle\end{array}}{\begin{array}{l}\langle [\downarrow^{m(p_1,\cdots,p_n)} S^{p,e}]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [\downarrow^{m(p_1,\cdots,p_n)} S^{p,e'}]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle\end{array}}$$

if

$S^{p,e'} \not\equiv \surd$
not ($n = 0$ and $S^{p,e'} \equiv \downarrow^{m'()} S^{p,e''}$ for some $m', S^{p,e''}$)

g'. *Tail-recursive method call*

$$\frac{\begin{array}{l}\langle [S^{p,e}]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [\downarrow^{m'()} S^{p,e'}]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle\end{array}}{\begin{array}{l}\langle [\downarrow^{m()} S^{p,e}]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [\downarrow^{m'()} S^{p,e'}]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma', ps'', \tau'\rangle\rangle, Sys^p, Sys\rangle\end{array}}$$

if

$\mid ps' \mid > 1$

and where

$ps'' = push(top(ps'), pop(pop(ps')))$

h'. *Method termination 1*

$$\frac{\begin{array}{l}\langle [S^{p,e}]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [\surd]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle\end{array}}{\begin{array}{l}\langle [\downarrow^{m(p_1,\cdots,p_n)} S^{p,e}]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [\surd]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma'', ps'', \tau''\rangle\rangle, Sys^p, Sys\rangle\end{array}}$$

if

$\mid ps' \mid > 1$
$Sys^p \equiv \langle CD_1^p \cdots CD^p \cdots CD_q^p\rangle$
$CD^p \equiv$ process class $C^p\langle\cdots\rangle$ $\cdots$ instance methods $MD_1^p \cdots MD^p \cdots MD_s^p$
$MD^p \equiv \mathrm{m}(u_1, \cdots, u_m)\ (v_1, \cdots, v_n)\ \mid w_1 \cdots w_o \mid S^p$

and where

$$\sigma_1,\chi_1 = \begin{cases} \sigma'\{\sigma'(proc)\{top(ps')(2)(v_1)/p_1\}/proc\}, top(pop(ps'))(2) & \text{if } p_1 \in IVar \\ \sigma', top(pop(ps'))(2)\{top(ps')(2)(v_1)/p_1\} & \text{if } p_1 \in LVar \end{cases}$$

$$\sigma_2,\chi_2 = \begin{cases} \sigma_1\{\sigma_1(proc)\{top(ps')(2)(v_2)/p_2\}/proc\}, \chi_1 & \text{if } p_2 \in IVar \\ \sigma_1, \chi_1\{top(ps')(2)(v_2)/p_2\} & \text{if } p_2 \in LVar \end{cases}$$

$$\vdots$$

$$\sigma_n,\chi_n = \begin{cases} \sigma_{n-1}\{\sigma_{n-1}(proc)\{top(ps')(2)(v_n)/p_n\}/proc\}, \chi_{n-1} & \text{if } p_n \in IVar \\ \sigma_{n-1}, \chi_{n-1}\{top(ps')(2)(v_n)/p_n\} & \text{if } p_n \in LVar \end{cases}$$

$\sigma'' = \sigma_n$ and $ps'' = push(\langle proc, \chi_n\rangle, pop(pop(ps')))$ and $\tau'' = \tau'$

i'. *Method termination 2*

$$\frac{\begin{array}{l}\langle [S^{p,e}]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [\surd]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle\end{array}}{\begin{array}{l}\langle [\downarrow^{m()}\ S^{p,e}]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [\surd]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma', \langle\rangle, \tau'\rangle\rangle, Sys^p, Sys\rangle\end{array}}$$

if

$$\begin{array}{l} \mid ps' \mid = 1 \\ Sys^p \equiv \langle CD_1^p \cdots CD^p \cdots CD_q^p\rangle \\ CD^p \equiv \text{process class } C^p\langle\cdots\rangle \ \cdots \ \text{instance methods } MD_1^p \cdots MD^p \cdots MD_s^p \\ MD^p \equiv \mathrm{m}(u_1, \cdots, u_m)\ ()\ \mid w_1 \cdots w_o \mid S^p \end{array}$$

j'. *Disrupt command, normal execution*

$$\frac{\begin{array}{l}\langle [S_1^{p,e}]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [S_1^{p,e'}]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle\end{array}}{\begin{array}{l}\langle [S_1^{p,e} >> S_2^p]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [S_1^{p,e'} >> S_2^p]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle\end{array}}$$

if

$$S_1^{p,e'} \not\equiv \surd$$

k'. *Disrupt command, successful termination*

$$\frac{\begin{array}{l}\langle [S_1^{p,e}]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [\surd]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle\end{array}}{\begin{array}{l}\langle [S_1^{p,e} >> S_2^p]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [\surd]_{CP(E_1,\cdots,E_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle\end{array}}$$

l'. *Disrupt command, disrupt occurrence*

$$\frac{\begin{array}{l}\langle [S_2^p]_{CP(E_1,\ldots,E_r)}, \langle\langle \sigma, ps'', \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [S_2^{p,e'}]_{CP(E_1,\ldots,E_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle\end{array}}{\begin{array}{l}\langle [S_1^{p,e} >> S_2^p]_{CP(E_1,\ldots,E_r)}, \langle\langle \sigma, ps, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [S_2^{p,e'}]_{CP(E_1,\ldots,E_r)}, \langle\langle \sigma', ps', \tau'\rangle\rangle, Sys^p, Sys\rangle\end{array}}$$

if

$| ps | \geq n$

and where

$n$ equals the amount of $\downarrow^{m(p_1,\ldots,p_k)}$ symbols contained in $S_1^{p,e}$
$ps'' = pop^n(ps)$

m'. *Parallel composition 1*

$$\frac{\langle BSpec_1^e, envs_1, Sys^p, Sys\rangle \xrightarrow{a} \langle BSpec_1^{e'}, envs_1', Sys^p, Sys\rangle}{\begin{array}{l}\langle BSpec_1^e \parallel BSpec_2^e, envs_1 \cdot envs_2, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle BSpec_1^{e'} \parallel BSpec_2^e, envs_1' \cdot envs_2, Sys^p, Sys\rangle\end{array}}$$

n'. *Parallel composition 2*

$$\frac{\langle BSpec_2^e, envs_2, Sys^p, Sys\rangle \xrightarrow{a} \langle BSpec_2^{e'}, envs_2', Sys^p, Sys\rangle}{\begin{array}{l}\langle BSpec_1^e \parallel BSpec_2^e, envs_1 \cdot envs_2, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle BSpec_1^e \parallel BSpec_2^{e'}, envs_1 \cdot envs_2', Sys^p, Sys\rangle\end{array}}$$

o'. *Parallel composition 3*

$$\frac{\begin{array}{l}\langle BSpec_1^e, envs_1, Sys^p, Sys\rangle \xrightarrow{\bar{l}} \langle BSpec_1^{e'}, envs_1', Sys^p, Sys\rangle \\ \langle BSpec_2^e, envs_2, Sys^p, Sys\rangle \xrightarrow{l} \langle BSpec_2^{e'}, envs_2', Sys^p, Sys\rangle\end{array}}{\begin{array}{l}\langle BSpec_1^e \parallel BSpec_2^e, envs_1 \cdot envs_2, Sys^p, Sys\rangle \xrightarrow{\tau} \\ \langle BSpec_1^{e'} \parallel BSpec_2^{e'}, envs_1' \cdot envs_2', Sys^p, Sys\rangle\end{array}}$$

p'. *Channel hiding*

$$\frac{\langle BSpec^e, envs, Sys^p, Sys\rangle \xrightarrow{a} \langle BSpec^{e'}, envs', Sys^p, Sys\rangle}{\langle BSpec^e \setminus L, envs, Sys^p, Sys\rangle \xrightarrow{a} \langle BSpec^{e'} \setminus L, envs', Sys^p, Sys\rangle}$$

if

$a = \tau$ or $Chan(a) \notin L$

q'. *Channel renaming*

$$\frac{\langle BSpec^e, envs, Sys^p, Sys\rangle \xrightarrow{a} \langle BSpec^{e\prime}, envs', Sys^p, Sys\rangle}{\langle BSpec^e[f], envs, Sys^p, Sys\rangle \xrightarrow{a'} \langle BSpec^{e\prime}[f], envs', Sys^p, Sys\rangle}$$

where

$$a' = \begin{cases} \tau & \text{if } a = \tau \\ f(a) & \text{if } a \neq \tau \end{cases}$$

r'. *Cluster execution*

$$\frac{\langle BSpec^e, envs, Sys^p, Sys\rangle \xrightarrow{a} \langle BSpec^{e\prime}, envs', Sys^p, Sys\rangle}{\begin{array}{l}\langle [BSpec^e]_{C^c(E_1,\cdots,E_r)}, envs, Sys^p, Sys\rangle \xrightarrow{a} \\ \langle [BSpec^{e\prime}]_{C^c(E_1,\cdots,E_r)}, envs', Sys^p, Sys\rangle\end{array}}$$

### 3.4.4 Some Properties of the Transition System

In this subsection we will prove some important properties of the labeled transition system of Subsection 3.4.3.

**Proposition 3.1**
Let $conf_1^p \in Conf^p$ and assume that $conf_1^p \xrightarrow{a} conf_2^p$ for some action $a \in Act$. Then we have

(i) $conf_2^p \in Conf^p$

(ii) $Reset(conf_1^p) = Reset(conf_2^p)$.

(iii) if $a \in \mathcal{L}$ then $Abs(a) \in AASort(conf_1^p)$

(iv) if $a \in \mathcal{L}$ then $Chan(a) \in ChSort(conf_1^p)$

(v) $AASort(conf_2^p) = AASort(conf_1^p)$

(vi) $ChSort(conf_2^p) = ChSort(conf_1^p)$

□

Item (i) of Proposition 3.1 states that every labeled relation $\xrightarrow{a}$ is indeed defined on $Conf^p$. Note that, since the definition rules of $Conf^p$ are conditional, this is by no means a trivial property. Item (ii) states that the *Reset* of a configuration is equal to the *Reset* of any of its derivates. From (iii) and (iv) it follows that the abstraction (*Abs*) of any non-silent action performed by some configuration is part of the *AASort* of that configuration, and that the corresponding channel is part of the configurations' channel sort. Finally (v) and

(vi) state that the *AASort* and *ChSort* of a configuration equal the *AASort* respectively *ChSort* of any derivate configuration.

**Proof** of Proposition 3.1
Items (iv) and (vi) directly follow from (iii) and (v) respectively. We will prove (i), (ii), (iii), and (v) simultaneously by transition induction on the shape of the derivation tree of $conf_1^p \xrightarrow{a} conf_2^p$. We argue by cases on the applied axioms and rules. Let $conf_1^p = \langle BSpec_1^e, envs, Sys^p, Sys\rangle$ and $conf_2^p = \langle BSpec_2^e, envs', Sys^p, Sys\rangle$ and assume $conf_1^p \xrightarrow{a} conf_2^p$.

**Case**[Axiom 1']
Then $BSpec_1^e \equiv [S]_{C^p(E_1,\cdots,E_r)}$, $BSpec_2^e \equiv [\surd]_{C^p(E_1,\cdots,E_r)}$ and $a = \tau$. (i) directly follows from the definition of $Conf^p$ and from $conf_1^p \in Conf^p$. (ii) is an immediate result of the definition of *Reset*. (iii) holds vacuously since $\tau \notin \mathcal{L}$. Further $AASort(conf_2^p) = AASort(BSpec_2^e, Sys^p) = AASort(C^p, Sys^p) = AASort(BSpec_1^e, Sys^p) = AASort(conf_1^p)$, and thus (v) is also satisfied.

**Case**[Axiom 2']
Then $BSpec_1^e \equiv [ch!m(E_1,\cdots,E_n)]_{C^p(E_1',\cdots,E_r')}$, $BSpec_2^e \equiv [\surd]_{C^p(E_1',\cdots,E_r')}$, and $a = ch!m[data]$ with $\mid data \mid = n$. (i), (ii), and (v) are proved as in case [Axiom 1']. For (iii) we notice that $Abs(a) = ch!m[\underline{n}] \in \{ch!m[\underline{n}]\} = AASort(ch!m(E_1,\cdots,E_n))$. Then, using the condition of rule (3.) of the definition of $Conf^p$, we derive $Abs(a) \in AASort(C^p, Sys^p)$, and thus $Abs(a) \in AASort(BSpec_1^e, Sys^p) = AASort(conf_1^p)$, so the result follows.

**Case**[Axiom 3']
Analogous to case [Axiom 2'].

**Case**[Axiom 4']
Then $BSpec_1^e \equiv [m(E_1,\cdots,E_m)(p_1,\cdots,p_n)]_{C^p(E_1',\cdots,E_r')}$, $BSpec_2^e \equiv [\downarrow^{m(p_1,\cdots,p_n)} S^p]_{C^p(E_1',\cdots,E_r')}$ with $S^p$ being the body of method $m$, and $a = \tau$. (ii), (iii), and (v) are proved as in case [Axiom 1']. For (i) we have to show that the condition of rule (3.) of the definition of $Conf^p$ is satisfied. The first part of the definition directly follows from $conf_1^p \in Conf^p$. For the second part we use context condition 10 together with the definition of *AASort*, and deduce $AASort(S^p) \subseteq AASort(C^p, Sys^p)$. From this the result follows since $AASort(\downarrow^{m(p_1,\cdots,p_n)} S^p) = AASort(S^p) \subseteq AASort(C^p, Sys^p)$.

**Cases**[Axioms 5',6',7']
Are proved analogous to case [Axiom 4'].

**Case**[Axiom 8']
Then $BSpec_1^e \equiv C^p(E_1,\cdots,E_r)$, $BSpec_2^e \equiv [m(E_1',\cdots,E_q')()]_{C^p(E_1,\cdots,E_r)}$, and $a = \tau$. (i), (ii), and (iii) are proved as in case [Axiom 1']. Item (v) holds since $AASort(conf_2^p) =$

$AASort(BSpec_2^e, Sys^p) = AASort(C^p, Sys^p) = AASort(BSpec_1^e, Sys^p) = AASort(conf_1^p)$.

**Case**[Axiom 9']
Then $BSpec_1^e \equiv C^c(E_1, \cdots, E_r)$, $BSpec_2^e \equiv [BSpec^p[\![E_1/P_1, \cdots, E_r/P_r]\!]]_{C^c(E_1,\cdots,E_r)}$ where $BSpec^p$ is the behaviour specification and where $P_1, \cdots, P_r$ are the expression parameters of cluster class with name $C^c$, $a = \tau$, and $envs = envs' = \langle\rangle$. (ii) is an immediate consequence of the definition of *Reset*. (iii) holds vacuously. (v) is satisfied since $AASort(Conf_2^p) = AASort(BSpec_2^e, Sys^p) = AASort(C^c, Sys^p) = AASort(BSpec_1^e, Sys^p) = AASort(conf_1^p)$. For (i) we first have to show that $\langle BSpec^p[\![E_1/P_1, \cdots, E_r/P_r]\!], \langle\rangle, Sys^p, Sys\rangle \in Conf^p$. Using context condition (15.) it easily follows that $BSpec^p[\![E_1/P_1, \cdots, E_r/P_r]\!] \in BSpecifications$, and thus that $\langle BSpec^p[\![E_1/P_1, \cdots, E_r/P_r]\!], \langle\rangle, Sys^p, Sys\rangle \in SSpecifications$. From this the result follows because $SSpecification \subseteq Conf^p$. The next thing we have to show for (i) is that the condition of rule (4.) of the definition of $Conf^p$ holds. The first part of the condition follows from $conf_1^p \in Conf^p$. For the second part we have to show that $Reset(BSpec^p [\![ E_1/P_1 , \cdots , E_r/P_r ]\!]) \equiv BSpec^p[\![E_1/P_1, \cdots, E_r/P_r]\!]$. But this follows from the fact that $BSpec^p[\![E_1/P_1, \cdots, E_r/P_r]\!] \in BSpecifications$ by applying Lemma 3.1.

**Case**[Rule a']
Then $BSpec_1^e \equiv [S_1^{p,e};\ S_2^p]_{C^p(E_1,\cdots,E_r)}$, $BSpec_2^e \equiv [S_1^{p,e'};\ S_2^p]_{C^p(E_1,\cdots,E_r)}$ where $S_1^{p,e} \not\equiv \surd$, and $\langle\ [S_1^{p,e}]_{C^p(E_1,\cdots,E_r)}\ ,\ env\ ,\ Sys^p\ ,\ Sys\rangle \xrightarrow{a} \langle\ [S_1^{p,e'}]_{C^p(E_1,\cdots,E_r)}\ ,\ env'\ ,\ Sys^p\ ,\ Sys\rangle$. Items (ii) and (v) are proved as in case [Axiom 1']. By induction we have $a \in \mathcal{L} \Rightarrow Abs(a) \in AASort(\langle [S_1^{p,e}]_{C^p(E_1,\cdots,E_r)}\ ,\ env\ ,\ Sys^p, Sys\rangle)$, but then clearly $a \in \mathcal{L} \Rightarrow Abs(a) \in AASort(conf_1^p)$, so (iii) holds. For (i) we have to show that the condition of rule (3.) of the definition of $Conf^p$ is satisfied. Again, the first part of the condition follows from $conf_1^p \in Conf^p$. For the second part first notice that since $conf_1^p \in Conf^p$, $AASort(S_1^{p,e};\ S_2^p) \subseteq AASort(C^p, Sys^p)$. But then $AASort(S_1^{p,e}) \subseteq AASort(C^p, Sys^p)$ and thus $\langle [S_1^{p,e}]_{C^p(E_1,\cdots,E_r)}, envs, Sys^p, Sys\rangle \in Conf^p$. By induction we then have $\langle [S_1^{p,e'}]_{C^p(E_1,\cdots,E_r)}, env', Sys^p, Sys\rangle \in Conf^p$ and thus $AASort(S_1^{p,e'}) \subseteq AASort(C^p, Sys^p)$. But then $AASort(S_1^{p,e'};\ S_2^p) \subseteq AASort(C^p, Sys^p)$, and consequently $conf_2^p \in Conf^p$.

**Case**[Rules b',c',d',e',f',g',h',i',j',k',l']
Are all proved analogous to case [Rule a'].

**Case**[Rule m']
Then $conf_1^p = \langle BSpec_1^e \parallel BSpec_2^e, envs_1 \cdot envs_2, Sys^p, Sys\rangle$, $conf_2^p = \langle BSpec_1^{e'} \parallel BSpec_2^e, envs_1' \cdot envs_2, Sys^p, Sys\rangle$, and $\langle BSpec_1^e, envs_1, Sys^p, Sys\rangle \xrightarrow{a} \langle BSpec_1^{e'}, envs_1', Sys^p, Sys\rangle$. By induction we have $a \in \mathcal{L} \Rightarrow Abs(a) \in AASort(\langle BSpec_1^e, envs_1, Sys^p, Sys\rangle)$, $AASort\ (\langle\ BSpec_1^e\ ,\ envs_1\ ,\ Sys^p\ ,\ Sys\rangle) = AASort(\langle BSpec_1^{e'}, envs_1', Sys^p, Sys\rangle)$, and $Reset\ (\langle\ BSpec_1^e\ ,\ envs_1\ ,\ Sys^p\ ,\ Sys\rangle) = Reset\ (\langle BSpec_1^{e'}, envs_1', Sys^p, Sys\rangle)$. From this, (ii), (iii), and (v) follows easily. For (i) observe that for $conf_1^p$ to be member of $Conf^p$, $\langle BSpec_1^e, envs_1, Sys^p, Sys\rangle$ and $\langle BSpec_2^e, envs_2, Sys^p, Sys\rangle$ both must be member of $Conf^p$ (see rule (5.) of the definition of $Conf^p$). But then by induction $\langle BSpec_1^{e'}, envs_1', Sys^p, Sys\rangle \in Conf^p$, and thus (using rule

(5.) again) $conf_2^p \in Conf^p$.

**Case**[Rules n',o',p',q']
Are proved in a similar way as case [Rule m'].

**Case**[Rule r']
Then $conf_1^p = \langle [BSpec^e]_{C^c(E_1,\cdots,E_r)}, envs, Sys^p, Sys\rangle$, $conf_2^p = \langle [BSpec^{e\prime}]_{C^c(E_1,\cdots,E_r)}$ , $envs'$ , $Sys^p$ , $Sys\rangle$, and $\langle BSpec^e, envs, Sys^p, Sys\rangle \xrightarrow{a} \langle BSpec^{e\prime}, envs', Sys^p, Sys\rangle$. Since $conf_1^p \in Conf^p$ we have that $Reset(BSpec^e) \equiv BSpec^p[\![E_1/P_1,\cdot,E_r/P_r]\!]$ where $BSpec^p$ denotes the behaviour specification and where $P_1,\cdots,P_n$ denote the expression parameters of the cluster class with name $C^c$. Item (ii) directly follows from the definition of *Reset*. By induction, $a \in \mathcal{L} \Rightarrow Abs(a) \in AASort(\langle BSpec^e, envs, Sys^p, Sys\rangle)$. Further we have $AASort(\langle BSpec^e, envs, Sys^p, Sys\rangle) = AASort(BSpec^e, Sys^p) = \{$ according to Lemma 3.2 $\}$ $AASort(Reset(BSpec^e), Sys^p) = AASort(BSpec^p[\![E_1/P_1,\cdots,E_r/P_r]\!], Sys^p) = \{$ according to Lemma 3.3 $\}$ $AASort$ $(BSpec^p$ , $Sys^p) = \{$ context condition (10.) $\}$ $AASort(C^c, Sys^p)$ $= AASort(conf_1^p)$, and thus (iii) follows. (v) is true because $AASort(conf_1^p) = AASort(C^c$ , $Sys^P) = AASort(conf_2^p)$. Since $conf_1^p \in Conf^p$, we have using rule (4.) of the definition of $Conf^p$ that $\langle BSpec^e$ , $envs$ , $Sys^p$ , $Sys\rangle \in Conf^p$. By induction we then have $\langle BSpec^{e\prime}, envs', Sys^p$ , $Sys\rangle \in Conf^p$ and $Reset$ $(\langle BSpec^e, envs, Sys^p, Sys\rangle) = Reset$ $(\langle BSpec^{e\prime}, envs', Sys^p, Sys\rangle)$. Now, (i) follows from rule (4.) of the definition of $Conf^p$.

This concludes the proof of Proposition 3.1. □

The proof of Proposition 3.1 is based on the following three lemmas:

**Lemma 3.1**
Let *BSpec* be a behaviour specification. Then
$Reset(BSpec) \equiv BSpec$.

**Proof**
The proof is an easy induction on the structure of *BSpec*. □

**Lemma 3.2**
Let $BSpec^e$ be an extended behaviour specification. Then $AASort(BSpec^e, Sys^p) =$
$AASort(Reset(BSpec^e), Sys^p)$

**Proof**
The proof proceeds by structural induction. □

**Lemma 3.3**
Let $BSpec^p$ be a parameterized behaviour specification. Then $AASort(BSpec^p, Sys^p) =$
$AASort(BSpec^p[\![g]\!], Sys^p)$

**Proof**
Again, the proof is an easy induction on the structure of *BSpec*. □

In the previous sections we have seen that a *system specification* describes a system in its *initial* state, i.e. before initialization has been taken place. A *configuration*, on the other hand, is also able to describe a system *during execution* or, if you want, during *simulation.* A system always starts in a configuration reflected by some system specification, and it proceeds its execution through a sequence of consecutive configurations. Sometimes it is convenient to be able to retrieve the initial specification, given some configuration. For this purpose, we have introduced function *Reset* in Subsection 3.4.2. One would expect that starting from the *Reset* of a configuration, it would be possible to return to that configuration. However, from the fact that there exist configurations that can never be reached from any system specification, it is easy to see that this cannot be true. For such an "unreachable" configuration, the *Reset* then is not the initial specification, but that specification built from the same instances, in the same way as the configuration. For a "reachable" configuration, on the other hand, it is true that it can be reached from its corresponding *Reset*. Furthermore the *Reset* is the *only* specification from which a configuration can be reached. These latter two properties, justifying the well-definedness of function *Reset*, are captured in Proposition 3.2.

**Proposition 3.2**
Let $conf^p$ be a configuration and let $SSpec \overset{\cdot}{\rightarrow}^* conf^p$ [4] for some system specification $SSpec$. Then $SSpec = Reset(conf^p)$.

**Proof**
The proof proceeds by induction on the syntactic structure of $conf^p$.

**Case** $conf^p = \langle C^p(E_1, \cdots, E_r), \langle\rangle, Sys^p, Sys\rangle$.
Assume $SSpec \overset{\cdot}{\rightarrow}^n conf^p$ for some $n \geq 1$. Then there exists a $conf^{p\prime}$ such that $SSpec \overset{\cdot}{\rightarrow}^* conf^{p\prime} \overset{\cdot}{\rightarrow} conf^p$. However, there exists no axiom or rule which can produce $conf^p$, and thus we have a contradiction. So $n = 0$ and $SSpec = conf^p$. But then $Reset(conf^p) = Reset(SSpec) = \{$ use Lemma 3.1 $\}$ $SSpec$.

**Case** $conf^p = \langle C^c(E_1, \cdots, E_r), \langle\rangle, Sys^p, Sys\rangle$.
Is proved in an analogous way.

**Case** $conf^p = \langle [S^{p,e}]_{C^p(E_1,\cdots,E_r)}, \langle\rangle, Sys^p, Sys\rangle$.
By inspection of the axioms and rules, it is easy to verify that then $SSpec = \langle C^p(E_1, \cdots, E_r)$ , $\langle\rangle$ , $Sys^p, Sys\rangle$. But this implies that $Reset(conf^p) = SSpec$.

**Case** $conf^p = \langle [BSpec^e]_{C^c(E_1,\cdots,E_r)}, envs, Sys^p, Sys\rangle$.
Analogous to the previous case.

**Case** $conf^p = \langle BSpec_1^e \parallel BSpec_2^e, envs_1 \cdot envs_2, Sys^p, Sys\rangle$.

[4]We write $conf^p \overset{\cdot}{\rightarrow} conf^{p\prime}$ to mean that $conf^p \xrightarrow{a} conf^{p\prime}$ for some action $a$.

By inspection of the deductive proof system, it is not hard to see that then $SSpec = \langle BSpec_1 \parallel BSpec_2, \langle\rangle, Sys^p, Sys\rangle$ for some $BSpec_1$ and $BSpec_2$, and that $\langle BSpec_1 , \langle\rangle , Sys^p , Sys \rangle \dot{\rightarrow}^* \langle BSpec_1^e, envs_1, Sys^p, Sys\rangle$ and $\langle BSpec_2 , \langle\rangle , Sys^p , Sys\rangle \dot{\rightarrow}^* \langle BSpec_2^e , envs_2 , Sys^p , Sys\rangle$. By induction we then have $Reset(\langle BSpec_1^e, envs_1, Sys^p, Sys\rangle) = \langle BSpec_1, \langle\rangle, Sys^p, Sys\rangle$ and $Reset ( \langle BSpec_2^e, envs_2, Sys^p, Sys\rangle) = \langle BSpec_2, \langle\rangle, Sys^p, Sys\rangle$ and thus $Reset(BSpec_1^e) = BSpec_1$ and $Reset(BSpec_2^e) = BSpec_2$. So $Reset(\langle BSpec_1^e \parallel BSpec_2^e, envs_1 \cdot envs_2, Sys^p, Sys\rangle = \langle Reset(BSpec_1^e) \parallel Reset(BSpec_2^e), \langle\rangle, Sys^p, Sys\rangle = \langle BSpec_1 \parallel BSpec_2, \langle\rangle, Sys^p, Sys\rangle = SSpec$.

**Cases** $conf^p = \langle BSpec^e \setminus L, envs, Sys^p, Sys\rangle$ or $conf^p = \langle BSpec^e[f], envs, Sys^p, Sys\rangle$. These cases are proved analogous to the previous case.

This concludes the proof of Proposition 3.2. □

## 3.4.5 Observational Equivalence and Semantic Function $\mathcal{M}_\approx$

In the introduction we mentioned that *correctness-preserving transformations* play an important role in our object-oriented methodology for the design of hardware/software systems. A correctness-preserving transformation takes a specification $Spec_1$ and transforms it into a specification $Spec_2$. Specification $Spec_2$ should be *correct* with respect to $Spec_1$. This basically means that the specifications are related by some predefined (binary) *correctness relation.* In general, it is required that the correctness relation is an *equivalence relation* which is also substitutive under some or all language constructors. A pair of related specifications is often called *equivalent* with respect to the corresponding correctness relation.

A currently well-known and practically applicable way to define a correctness relation is in terms of so-called *bisimulations*. A bisimulation establishes a kind of invariant holding between a pair of dynamic systems, and the technique is to prove two systems equivalent by establishing such an invariant. The notion of bisimulation was first introduced by Park in [Par81] and later developed in the context of CCS in [Mil83].

In this report we will only define one kind of correctness relation, namely *observational equivalence.* Observational equivalence is defined terms of *weak bisimulations.* Weak bisimulations are binary relations on configurations.

**Definition 3.7**
A binary relation $\mathcal{S}$ over configurations $\mathcal{S} \subseteq Conf^p \times Conf^p$ is a weak bisimulation if $(conf_1^p, conf_2^p)$ implies

(i) the $Sys$ parts of $conf_1^p$ and $conf_2^p$ are syntactically identical

and for all $a \in Act$

(ii) if $conf_1^p \xrightarrow{a} conf_1^{p\prime}$ then, for some $conf_2^{p\prime}$, $conf_2^p \overset{\hat{a}}{\Rightarrow} conf_2^{p\prime}$ and $(conf_1^{p\prime}, conf_2^{p\prime}) \in \mathcal{S}$

(iii) if $conf_2^p \xrightarrow{a} conf_2^{p\prime}$ then, for some $conf_1^{p\prime}$, $conf_1^p \overset{\hat{a}}{\Rightarrow} conf_1^{p\prime}$ and $(conf_1^{p\prime}, conf_2^{p\prime}) \in \mathcal{S}$. □

Here, for each $a \in Act$, $\overset{\hat{a}}{\Rightarrow}$ is a binary transition relation over configurations (often called *descendant* relation) and is defined by

**Definition 3.8**
Let $conf_1^p$, $conf_2^p$ be configurations and let $l$ be a communication action. Then

$$conf_1^p \overset{\hat{\tau}}{\Rightarrow} conf_2^p \text{ if } conf_1^p(\xrightarrow{\tau})^* conf_2^p$$
$$conf_1^p \overset{\hat{l}}{\Rightarrow} conf_2^p \text{ if } conf_1^p(\xrightarrow{\tau})^* \xrightarrow{l} (\xrightarrow{\tau})^* conf_2^p$$

□

Our notion of weak bisimulation is very similar to that of CCS [Mil89]. The difference is that CCS does not include (i) in its definition. We, however, need to incorporate (i) to ensure that every data class name referred to from within any communication action, unambiguously denotes a *single* data class.

The following definition uses weak bisimulations to define observational equivalence ($\approx$) upon configurations.

**Definition 3.9**
$conf_1^p$ and $conf_2^p$ are observational equivalent, written $conf_1^p \approx conf_2^p$, if $(conf_1^p, conf_2^p) \in \mathcal{S}$ for some weak bisimulation $\mathcal{S}$. So

$$\approx \; = \; \bigcup\{\mathcal{S} \mid \mathcal{S} \text{ is a weak bisimulation}\}$$ □

It is not hard to prove that $\approx$ is an equivalence relation. Further, it can be shown that $\approx$ is substitutive under parallel composition, channel hiding, and channel renaming.

We will now use relation $\approx$ to assign a meaning to configurations (and thus to specifications). Since $\approx$ is an equivalence relation, we can consider the meaning of a configuration (and implicitly of system specifications) to be the class of all observational-equivalent configurations, so,

$$\mathcal{M}_{\approx}(conf^p) \; = \; [conf^p]_{\approx}$$

Evidently, $\mathcal{M}_{\approx}(conf_1^p) = \mathcal{M}_{\approx}(conf_2^p)$ if and only if $conf_1^p \approx conf_2^p$, so, two configurations are *observational equivalent*, precisely if they have the same semantics.

Although relation $\approx$ and function $\mathcal{M}_{\approx}$ reflect a very strong notion of equivalence, we have good hope that they can be used to prove a set of interesting correctness-preserving transformations. For transformations which are not correct under observational equivalence, alternative equivalence relations and semantical functions will have to be developed. Defining a useful set of transformations, developing suitable equivalence relations, and proving correctness of transformations will be subject of future research.

### 3.4.6 Example: A Simple Handshake Protocol

In this subsection we will show how to calculate the behaviour of a simple handshake protocol, using the labeled-transition system defined in Subsection 3.4.3. Further, we will demonstrate that the protocol behaves as a 1-place buffer. The protocol, consisting of a *Sender* process and a *Receiver* process, is visualized in figure 3.1.

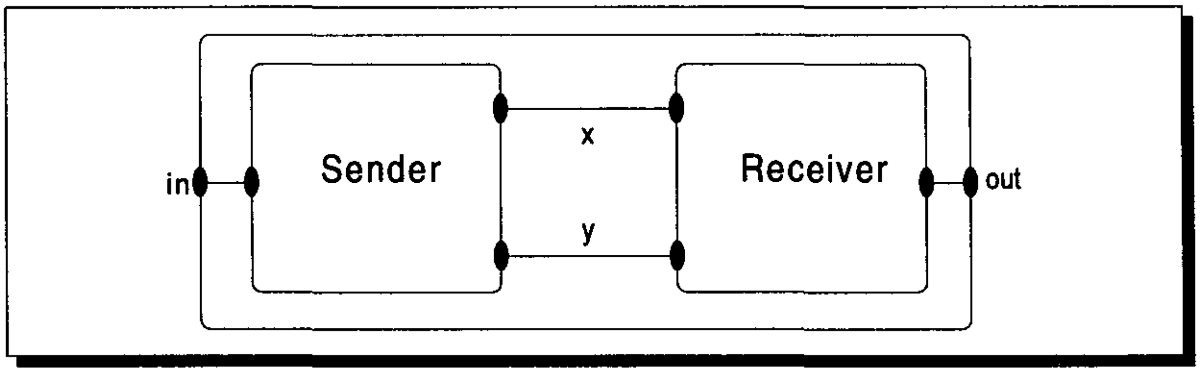

Figure 3.1: A Simple Handshake Protocol

The *Sender* can receive some *data* from channel *in* and this data is delivered by the *Receiver* at channel *out*. Externally the protocol behaves as a *1-place buffer*. The protocol is specified as follows:

$$\langle (\mathit{Sender} \parallel \mathit{Receiver}) \setminus \{x, y\}, \langle\rangle, \mathit{Sys}^p, \mathit{Sys}\rangle$$

For reasons of simplicity we assume that there are no non-primitive data classes, so $\mathit{Sys} = \langle\rangle$. $\mathit{Sys}^p$ consists of process classes *Sender* and *Receiver*. It is defined as $\mathit{Sys}^p = \langle$ [5]

| | |
|---|---|
| process class | *Sender* |
| instance variables | |
| communication channels | *in x y* |
| message interface | *in?receive($d_S$) x!transfer($d_S$) y?ack* |
| initial method call | *start* |
| instance methods | **start** $\mid d_S \mid$ |
| | *in?receive*($d_S$); *x!transfer*($d_S$); *y?ack*; **start** |

| | |
|---|---|
| process class | *Receiver* |
| instance variables | |
| communication channels | *x y out* |
| message interface | *x?transfer($d_R$) out!deliver($d_R$) y!ack* |

[5]The specifications and configurations use some obvious syntactic simplifications.

| | |
|---|---|
| initial method call | *start* |
| instance methods | **start** $\mid d_R \mid$ $x?transfer(d_R);\ out!deliver(d_R);\ y!ack;$ **start** $)$ |

Configuration $\langle (Sender \parallel Receiver) \setminus \{x, y\}, \langle\rangle, Sys^p, Sys\rangle$ reflects the state of the protocol where both the sender and the receiver are still uninitialized. The actual computation starts by initializing either the sender or the receiver. By [axiom 8'] we have

$$\langle Sender, \langle\rangle, Sys^p, Sys\rangle \xrightarrow{\tau}$$
$$\langle [\mathbf{start}]_{Sender}, \langle\langle \{proc \to \varnothing\}, \langle\rangle, \varnothing\rangle\rangle, Sys^p, Sys\rangle$$

Using [rule m'] we then deduce

$$\langle Sender \parallel Receiver, \langle\rangle, Sys^p, Sys\rangle \xrightarrow{\tau}$$
$$\langle [\mathbf{start}]_{Sender} \parallel Receiver, \langle\langle \{proc \to \varnothing\}, \langle\rangle, \varnothing\rangle\rangle, Sys^p, Sys\rangle$$

and thus by [rule p']

$$\langle (Sender \parallel Receiver) \setminus \{x, y\}, \langle\rangle, Sys^p, Sys\rangle \xrightarrow{\tau}$$
$$\langle [(\mathbf{start}]_{Sender} \parallel Receiver) \setminus \{x, y\}, \langle\langle \{proc \to \varnothing\}, \langle\rangle, \varnothing\rangle\rangle, Sys^p, Sys\rangle$$

By applying [axiom 4'] and [rules m',p'] we get

$$\langle [(\mathbf{start}]_{Sender} \parallel Receiver) \setminus \{x, y\}, \langle\langle \{proc \to \varnothing\}, \langle\rangle, \varnothing\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{\tau}$$
$$\langle ([\downarrow^{start} (in?receive(d_S);\ x!transfer(d_S);\ y?ack;\ \mathbf{start})]_{Sender} \parallel Receiver) \setminus \{x, y\},$$
$$\langle\langle \{proc \to \varnothing\}, \langle\langle proc, \{d_S \to nil\}\rangle\rangle, \varnothing\rangle\rangle, Sys^p, Sys\rangle$$

The latter configuration reflects the situation where the receiver is able to receive message *receive* and data $d_S$ from channel *in*, and where the receiver is still uninitialized. If we let the receiver perform its message reception we get, by applying [axiom 3'] and [rules b',m',p']

$$\langle ([\downarrow^{start} (in?receive(d_S);\ x!transfer(d_S);\ y?ack;\ \mathbf{start})]_{Sender} \parallel Receiver) \setminus \{x, y\},$$
$$\langle\langle \{proc \to \varnothing\}, \langle\langle proc, \{d_S \to nil\}\rangle\rangle, \varnothing\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{in?receive[data]}$$
$$\langle ([\downarrow^{start} (x!transfer(d_S);\ y?ack;\ \mathbf{start})]_{Sender} \parallel Receiver) \setminus \{x, y\},$$
$$\langle\langle \{proc \to \varnothing\}, \langle\langle proc, \{d_S \to \gamma\}\rangle\rangle, \varnothing\rangle\rangle, Sys^p, Sys\rangle$$

Here, $data = \langle\langle \gamma, \varnothing, \varnothing\rangle\rangle$ for some primitive data object $\gamma \in PDObj$. This can be seen as follows: According to [axiom 3'], *data* should be of the form $\langle\langle \beta_1, \sigma_1, \tau_1\rangle\rangle \in Struc^1_{Sys,min}$. Since $Sys = \langle\rangle$ we have by (2) of Definition 2.1 that $Dom(\tau_1) = \varnothing$, and therefore by (1) of the same definition that $Dom(\sigma_1 \restriction NDObj) = \varnothing$. By Proposition 2.1(c) we know that $proc \notin Dom(\sigma_1)$ and thus $Dom(\sigma_1) = \varnothing$. Now according to Definition 2.1(4) $\beta_1 \in NDObj$ implies $\beta_1 \in Dom(\sigma_1)$. Therefore $\beta_1 \in PDObj$ and thus thus *data* is of the form $\langle\langle \gamma, \varnothing, \varnothing\rangle\rangle$.

If we continue calculating the transitions between the involved configurations we construct a so-called *transition graph*, shown in Figure 3.2, representing the behaviour of the protocol.

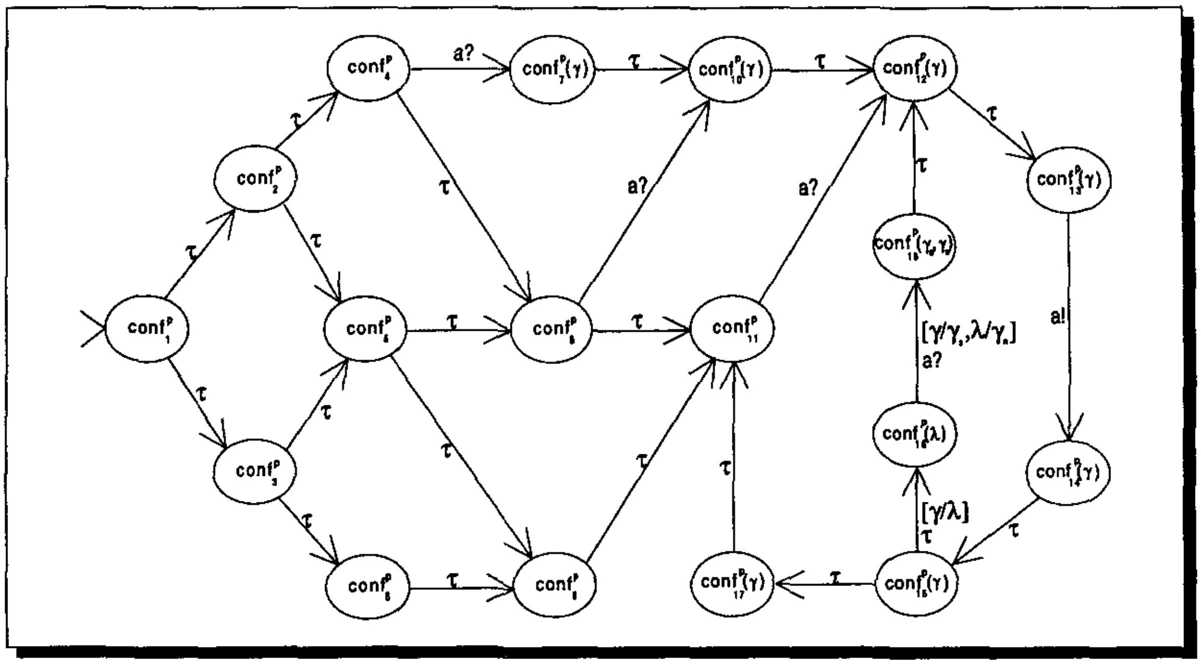

Figure 3.2: A Transition Graph of the Handshake Protocol

The transition graph consists of 18 configurations (represented by the nodes of the graph), some of which are parameterized. The > mark attached to node $conf_1^p$ denotes that this is the starting configuration. The substitution $[\gamma/\gamma_S, \lambda/\gamma_R]$ indicates that parameters $\gamma_S$ and $\gamma_R$ of $conf_{18}^p(\gamma_S, \gamma_R)$ have to be replaced by $\gamma$ and $\lambda$ respectively. Substitution $[\gamma/\lambda]$ has a similar meaning. Further, $a?$ denotes receive action $in?receive[\langle\langle\gamma, \varnothing, \varnothing\rangle\rangle]$ and $a!$ denotes send action $out!deliver[\langle\langle\gamma, \varnothing, \varnothing\rangle\rangle]$.

Note that, although the graph of Figure 3.2 has a *finite* amount of nodes, it represents an *infinite* transition graph. Node $conf_7^p(\gamma)$, for example, represents a collection of nodes, one for each $\gamma \in PDObj$. The configurations are given by

$$\begin{array}{ll}
conf_1^p & = \langle (Sender \parallel Receiver) \setminus \{x, y\}, \langle\rangle, Sys^p, Sys\rangle \\
conf_2^p & = \langle ([\mathbf{start}]_{Sender} \parallel Receiver) \setminus \{x, y\}, \langle env_S\varnothing\rangle, Sys^p, Sys\rangle \\
conf_3^p & = \langle (Sender \parallel [\mathbf{start}]_{Receiver}) \setminus \{x, y\}, \langle env_R\varnothing\rangle, Sys^p, Sys\rangle \\
conf_4^p & = \langle ([\downarrow^{start} (in?receive(d_S);\ x!transfer(d_S);\ y?ack;\ \mathbf{start})]_{Sender} \parallel Receiver) \\
& \quad \setminus \{x, y\}, \langle env_S(nil)\rangle, Sys^p, Sys\rangle \\
conf_5^p & = \langle ([\mathbf{start}]_{Sender} \parallel [\mathbf{start}]_{Receiver}) \setminus \{x, y\}, \langle env_S\varnothing, env_R\varnothing\rangle, Sys^p, Sys\rangle \\
conf_6^p & = \langle (Sender \parallel [\downarrow^{start} (x?transfer(d_R);\ out!deliver(d_R);\ y!ack;\ \mathbf{start})]_{Receiver}) \\
& \quad \setminus \{x, y\}, \langle env_R(nil)\rangle, Sys^p, Sys\rangle \\
conf_7^p(\gamma) & = \langle ([\downarrow^{start} (x!transfer(d_S);\ y?ack;\ \mathbf{start})]_{Sender} \parallel Receiver) \\
& \quad \setminus \{x, y\}, \langle env_S(\gamma)\rangle, Sys^p, Sys\rangle
\end{array}$$

$$\begin{array}{ll}
conf_8^p & = \langle(([\downarrow^{start}(in?receive(d_S);\ x!transfer(d_S);\ y?ack;\ \mathbf{start})]_{Sender} \parallel \\
& \quad [\mathbf{start}]_{Receiver}) \setminus \{x,y\}, \langle env_S(nil), env_{R\varnothing}\rangle, Sys^p, Sys\rangle \\
conf_9^p & = \langle(Sender \parallel [\downarrow^{start}(x?transfer(d_R);\ out!deliver(d_R);\ y!ack;\ \mathbf{start})]_{Receiver}) \\
& \quad \setminus\{x,y\}, \langle env_{S\varnothing}, env_R(nil)\rangle, Sys^p, Sys\rangle \\
conf_{10}^p(\gamma) & = \langle(([\downarrow^{start}(x!transfer(d_S);\ y?ack;\ \mathbf{start})]_{Sender} \parallel [\mathbf{start}]_{Receiver}) \\
& \quad \setminus\{x,y\}, \langle env_S(\gamma), env_{R\varnothing}\rangle, Sys^p, Sys\rangle \\
conf_{11}^p & = \langle(([\downarrow^{start}(in?receive(d_S);\ x!transfer(d_S);\ y?ack;\ \mathbf{start})]_{Sender}) \parallel \\
& \quad [\downarrow^{start}(x?transfer(d_R);\ out!deliver(d_R);\ y!ack;\ \mathbf{start})]_{Receiver}) \\
& \quad \setminus\{x,y\}, \langle env_S(nil), env_R(nil)\rangle, Sys^p, Sys\rangle \\
conf_{12}^p(\gamma) & = \langle(([\downarrow^{start}(x!transfer(d_S);\ y?ack;\ \mathbf{start})]_{Sender} \parallel \\
& \quad [\downarrow^{start}(x?transfer(d_R);\ out!deliver(d_R);\ y!ack;\ \mathbf{start})]_{Receiver}) \\
& \quad \setminus\{x,y\}, \langle env_S(\gamma), env_R(nil)\rangle, Sys^p, Sys\rangle \\
conf_{13}^p(\gamma) & = \langle(([\downarrow^{start}(y?ack;\ \mathbf{start})]_{Sender} \parallel [\downarrow^{start}(out!deliver(d_R);\ y!ack; \\
& \quad \mathbf{start})]_{Receiver}) \setminus \{x,y\}, \langle env_S(\gamma), env_R(\gamma)\rangle, Sys^p, Sys\rangle \\
conf_{14}^p(\gamma) & = \langle(([\downarrow^{start}(y?ack;\ \mathbf{start})]_{Sender} \parallel [\downarrow^{start}(y!ack;\ \mathbf{start})]_{Receiver}) \\
& \quad \setminus\{x,y\}, \langle env_S(\gamma), env_R(\gamma)\rangle, Sys^p, Sys\rangle \\
conf_{15}^p(\gamma) & = \langle(([\downarrow^{start}(\mathbf{start})]_{Sender} \parallel [\downarrow^{start}(\mathbf{start})]_{Receiver}) \\
& \quad \setminus\{x,y\}, \langle env_S(\gamma), env_R(\gamma)\rangle, Sys^p, Sys\rangle \\
conf_{16}^p(\gamma) & = \langle([\downarrow^{start}(in?receive(d_S);\ x!transfer(d_S);\ y?ack;\ \mathbf{start})]_{Sender} \parallel \\
& \quad [\downarrow^{start}(\mathbf{start})]_{Receiver}) \setminus \{x,y\}, \langle env_S(nil), env_R(\gamma)\rangle, Sys^p, Sys\rangle \\
conf_{17}^p(\gamma) & = \langle(([\downarrow^{start}(\mathbf{start})]_{Sender} \parallel [\downarrow^{start}(x?transfer(d_R);\ out!deliver(d_R);\ y!ack; \\
& \quad \mathbf{start})]_{Receiver}) \setminus \{x,y\}, \langle env_S(\gamma), env_R(nil)\rangle, Sys^p, Sys\rangle \\
conf_{18}^p(\gamma_S,\gamma_R) & = \langle([\downarrow^{start}(x!transfer(d_S);\ y?ack;\ \mathbf{start})]_{Sender} \parallel [\downarrow^{start}(\mathbf{start})]_{Receiver}) \\
& \quad \setminus\{x,y\}, \langle env_S(\gamma_S), env_R(\gamma_R)\rangle, Sys^p, Sys\rangle
\end{array}$$

Here, $env_{S\varnothing}$ and $env_S(\gamma)$ denote environments of the *Sender* process. In $env_{S\varnothing}$ no local variables are allocated on the stack (the stack is empty), whereas in $env_S(\gamma)$ variable $d_S$ is allocated and bound to primitive object $\gamma$. Environments $env_{R\varnothing}$ and $env_R(\gamma)$ have a similar meaning for the *Receiver* process. The environments are defined as

$$\begin{array}{ll}
env_{S\varnothing} & = \langle\{proc \to \varnothing\}, \langle\rangle, \varnothing\rangle \\
env_{R\varnothing} & = \langle\{proc \to \varnothing\}, \langle\rangle, \varnothing\rangle \\
env_S(\gamma) & = \langle\{proc \to \varnothing\}, \langle\langle proc, \{d_S \to \gamma\}\rangle\rangle, \varnothing\rangle \\
env_R(\gamma) & = \langle\{proc \to \varnothing\}, \langle\langle proc, \{d_R \to \gamma\}\rangle\rangle, \varnothing\rangle
\end{array}$$

In the beginning of this subsection we mentioned that the protocol externally behaves as a 1-place buffer (see Figure 3.3). We will show this by giving a explicit specification of such a buffer, and then show that this specification is observational equivalent to the protocol.

The 1-place buffer is specified as

$$\langle Buffer, \langle\rangle, Sys^{p\prime}, Sys\rangle$$

where $Sys = \langle\rangle$ and where $Sys^{p\prime} = \langle$

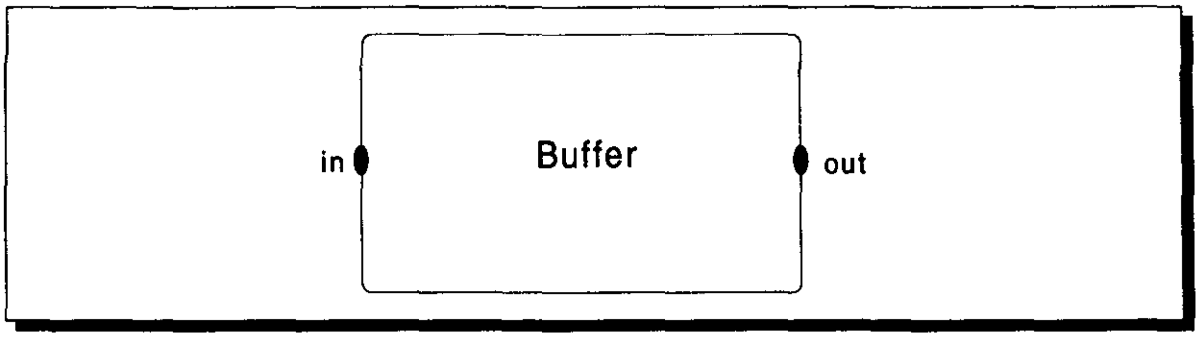

Figure 3.3: A 1-place Buffer

| | |
|---|---|
| process class | *Buffer* |
| instance variables | |
| communication channels | *in out* |
| message interface | *in?receive*$(d_B)$ *out!deliver*$(d_B)$ |
| initial method call | *start* |
| instance methods | **start** $\mid d_B \mid$ |
| | *in?receive*$(d_B)$; *out!deliver*$(d_B)$; **start** $\rangle$ |

A transition graph of the buffer is given in Figure 3.4. The (parameterized) configurations are given by

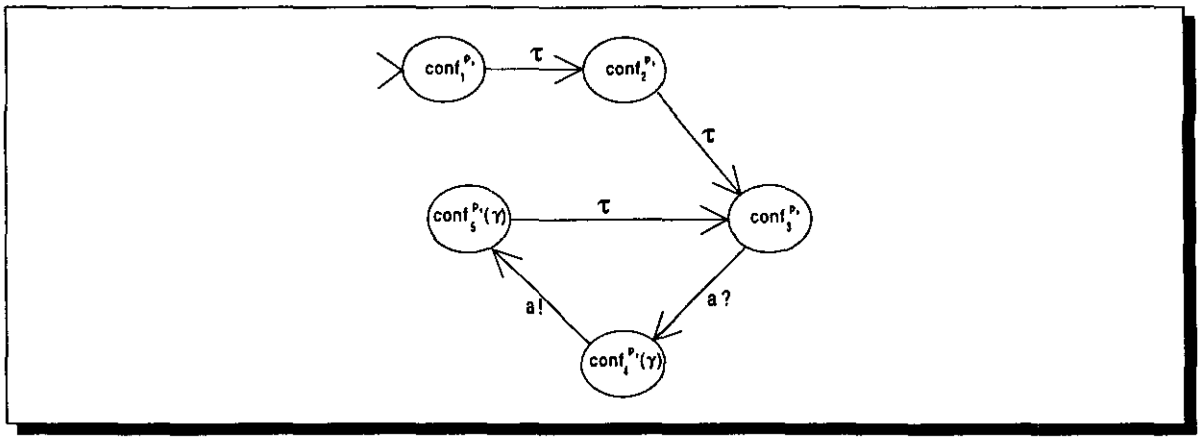

Figure 3.4: A Transition Graph of the 1-place Buffer

$$\begin{aligned}
conf_1^{p'} &= \langle Buffer, \langle\rangle, Sys^{p'}, Sys\rangle \\
conf_2^{p'} &= \langle [\mathbf{start}]_{Buffer}, \langle env_{B\varnothing}\rangle, Sys^{p'}, Sys\rangle \\
conf_3^{p'} &= \langle [\downarrow^{start} (in?receive(d_B);\ out!deliver(d_B);\ \mathbf{start})]_{Buffer}, \\
&\qquad \langle env_B(nil)\rangle, Sys^{p'}, Sys\rangle \\
conf_4^{p'}(\gamma) &= \langle [\downarrow^{start} (out!deliver(d_B);\ \mathbf{start})]_{Buffer}, \langle env_B(\gamma)\rangle, Sys^{p'}, Sys\rangle \\
conf_5^{p'}(\gamma) &= \langle [\downarrow^{start} (\mathbf{start})]_{Buffer}, \langle env_B(\gamma)\rangle, Sys^{p'}, Sys\rangle
\end{aligned}$$

where

$$\begin{aligned}
env_{B\varnothing} &= \langle \{proc \to \varnothing\}, \langle\rangle, \varnothing\rangle \\
env_B(\gamma) &= \langle \{proc \to \varnothing\}, \langle (proc, \{d_S \to \gamma\})\rangle, \varnothing\rangle
\end{aligned}$$

It is not hard to verify that relation $\mathcal{S}$ defined by

$$\mathcal{S} = (A \times B) \cup (C \times D)$$

with

$$\begin{aligned}
A &= \{conf_1^p, \cdots, conf_5^p, conf_8^p, conf_9^p, conf_{11}^p\} \cup \\
&\quad \{conf_{14}^p(\gamma), conf_{15}^p(\gamma), conf_{16}^p(\gamma), conf_{17}^p(\gamma) \mid \gamma \in PDObj\} \\
B &= \{conf_1^{p'}, conf_2^{p'}, conf_3^{p'}, conf_5^{p'}\} \\
C &= \{conf_{10}^p(\gamma), conf_{12}^p(\gamma), conf_{13}^p(\gamma) \mid \gamma \in PDObj\} \cup \{conf_{18}^p(\gamma, \lambda) \mid \gamma, \lambda \in PDObj\} \\
D &= \{conf_4^{p'}\}
\end{aligned}$$

is a weak bisimulation. Since $conf_1^p, conf^{p'}_1 \in \mathcal{S}$, we have that $\langle (Sender \parallel Receiver) \setminus \{x, y\}\langle\rangle, Sys^p, Sys\rangle \approx \langle Buffer, \langle\rangle, Sys^p, Sys\rangle$, and thus the protocol is observational equivalent to the 1-place buffer.

One would expect that a similar result could be proved if the protocol and the 1-place buffer were able to receive and deliver *arbitrary data objects* instead of only *primitive data objects*. However, it is not hard to find out that observational equivalence is too strong for this to be true. Developing equivalence relations which *do* establish the equivalence will be subject of future research.

# Chapter 4

# Reviewing the Development of POOSL

Developing a formal language is a very complicated task. Language constructs which seem to have a clear meaning at first glance, may appear to be a lot trickier than expected. During the development of POOSL we found out that difficulties or unclarities often show up only at the moment one tries to describe the meaning of constructs in very precise and detailed, that is formal, way. To solve encountered difficulties, alternative semantic interpretations or possible alternative language constructs have to be evaluated. A formal language description is then of great help. It provides a deep understanding of encountered problems and it aids in evaluating design alternatives in a systematic way.

During the development of POOSL, lots of problems were encountered and a large number of design decisions were taken. In the following sections three of the more interesting problems are studied.

## 4.1 The Grain of Concurrency

The grain of concurrency in POOSL is the processes object. All internal activities of process objects are performed sequentially. The evaluation of data expressions, for example, is always performed *from left to right.* Consider message-send expression $E\ m(E_1, \cdots, E_n)$. To evaluate this expression, first destination expression $E$ is evaluated. Then the parameters $E_1, \cdots, E_n$ are evaluated from left to right and finally the message is sent to the destination object. The order in which the expressions are evaluated is formalized by rules *a* and *b* of the transition system of Subsection 2.4.3:

a. *Method call 1*

$$\frac{\langle E^e, \sigma, s, \tau, Sys\rangle \ \rightarrow\ \langle E^{e\prime}, \sigma', s', \tau', Sys\rangle}{\begin{array}{l}\langle E^e\ m(E_1^e, \cdots, E_n^e), \sigma, s, \tau, Sys\rangle\ \rightarrow \\ \langle E^{e\prime}\ m(E_1^e, \cdots, E_n^e), \sigma', s', \tau', Sys\rangle\end{array}}$$

b. *Method call 2*

$$\frac{\langle E^e, \sigma, s, \tau, Sys\rangle \rightarrow \langle E^{e\prime}, \sigma', s', \tau', Sys\rangle}{\begin{array}{l}\langle \underline{\beta}\ m(\underline{\beta_1}, \cdots, \underline{\beta_{i-1}}, E^e, \cdots, E^e_n), \sigma, s, \tau, Sys\rangle \rightarrow \\ \langle \underline{\beta}\ m(\underline{\beta_1}, \cdots, \underline{\beta_{i-1}}, E^{e\prime}, \cdots, E^e_n), \sigma', s', \tau', Sys\rangle\end{array}}$$

Why did we chose to evaluate the expressions in a strict *sequential* manner, in stead of in a *concurrent* one? For it seems that concurrent expression evaluation can easily be dealt with by replacing rule $b$ by $b''$:

b". *Method call 2*

$$\frac{\langle E^e_i, \sigma, s, \tau, Sys\rangle \rightarrow \langle E^{e\prime}_i, \sigma', s', \tau', Sys\rangle}{\begin{array}{l}\langle E^e\ m(E^e_1, \cdots, E^e_{i-1}, E^e_i, \cdots, E^e_n), \sigma, s, \tau, Sys\rangle \rightarrow \\ \langle E^e\ m(E^e_1, \cdots, E^e_{i-1}, E^{e\prime}_i, \cdots, E^e_n), \sigma', s', \tau', Sys\rangle\end{array}}$$

However, if we take a closer look at this rule, we see that it does not formalize concurrent expression evaluation at all! The problem is that the rule allows $n + 1$ different data objects to be executing one of their methods at the same time. Therefore $n + 1$ different local variable environments have to be active at the same time. Unfortunately, there exists only one active environment and this environment is positioned at the top of stack $s$ and belongs to the object which is currently executing one of its methods. This problem is not easily solved in our chosen type of semantics. Each possible solution will most certainly complicate the semantics in a serious way.

Next to this theoretical difficulty of concurrent expression evaluation, a more practical problem exists. Consider the following definition of a (very simple and restricted) class *Bit*:

```
data class           Bit
instance variables   bit
instance methods


  error       setToZero   setToOne   invert
   primitive  bit ← 0;    bit ← 1;    if bit = 0 then bit := bit + 1
              self        self                    else bit := bit − 1 fi;
                                      if bit < 0 or bit > 1 then self error fi;
                                      self
```

Primitive message *error* aborts the execution with an error message. Assume that a variable $b$ refers to a *Bit* which is *setToZero* and consider the concurrent evaluation of expression ($b$ *invert*) == ($b$ *invert*). It seems clear that the result of this evaluation must be *true* [1]. Indeed, *true* is one of the possible outcomes. Another possibility, however, is that the evaluation *aborts* with an error message, leaving *b* in the unexpected state where instance variable *bit* refers to 2!

Problems of this kind are well-known in object-oriented languages, such as Smalltalk-80 [GR89], that support *processes* as an orthogonal language concept. These processes may act on the same collection of objects. It is even possible that they are executing the same method in the same object at the same time [AR89]. This can result in problems of synchronization and mutual exclusion as in the case of our *Bit* example.

Of course, these problems of synchronization and mutual exclusion can be solved if concurrently evaluated expressions are required to be side-effect free. However, the application of expressions with side-effects in object-oriented languages is not at all unusual. In Eiffel [Mey88], a restricted use of functions with side-effects is even recommended and exploited!

The above described problems also occur when other forms of concurrency within process objects are allowed. We have therefore determined the grain of concurrency at the level of the process object.

## 4.2 Layers of Semantics

The semantics of POOSL consists of two layers. At the layer of data objects, data expressions and data statements may take lots of small steps to be evaluated respectively executed. At the layer of process objects, these small steps are abstracted from and combined into single steps. In this section we will explain why we decided to build the semantics this way. For an alternative would have been to consider only one semantic layer and to formalize this layer by a single labeled-transition system.

Consider guarded command $[E]S^p$ whose execution is given by rule $e'$ of Subsection 3.4.3:

e'. *Guarded command*

$$\frac{\begin{array}{l}\langle [S^p]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma', ps', \tau' \rangle\rangle, Sys^p, Sys \rangle \xrightarrow{a} \\ \langle [S^{p,e}]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma'', ps'', \tau'' \rangle\rangle, Sys^p, Sys \rangle\end{array}}{\begin{array}{l}\langle [[E]S^p]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma, ps, \tau \rangle\rangle, Sys^p, Sys \rangle \xrightarrow{a} \\ \langle [S^{p,e}]_{C^p(E_1,\cdots,E_r)}, \langle\langle \sigma'', ps'', \tau'' \rangle\rangle, Sys^p, Sys \rangle\end{array}}$$

if

$$\langle \underline{\gamma}, \sigma', ps', \tau', Sys \rangle \in \mathcal{M}(\langle E, \sigma, ps, \tau, Sys \rangle) \text{ with } \gamma \in \{\mathit{true}, \mathit{bunk}\}$$

[1]Note that the primitive equality message == only determines whether two expressions refer to the *same* object or not.

In the alternative approach, the guarded command could have been described by the following axiom and rule:

*Guarded command, axiom*

$$\langle[[\gamma]S^p]_{CP(E_1,\cdots,E_r)}, \langle\langle\sigma, s, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{\tau} \langle[S^p]_{CP(E_1,\cdots,E_r)}, \langle\langle\sigma, s, \tau\rangle\rangle, Sys^p, Sys\rangle$$

if

$$\gamma = \mathit{true} \text{ or } \gamma = \mathit{bunk}$$

*Guarded command, rule*

$$\frac{\langle[E^e]_{CP(E_1,\cdots,E_r)}, \langle\langle\sigma, s, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{\tau} \langle[E^{e\prime}]_{CP(E_1,\cdots,E_r)}, \langle\langle\sigma', s', \tau'\rangle\rangle, Sys^p, Sys\rangle}{\langle[[E^e]S^p]_{CP(E_1,\cdots,E_r)}, \langle\langle\sigma, s, \tau\rangle\rangle, Sys^p, Sys\rangle \xrightarrow{\tau} \langle[[E^{e\prime}]S^p]_{CP(E_1,\cdots,E_r)}, \langle\langle\sigma', s', \tau'\rangle\rangle, Sys^p, Sys\rangle}$$

The problem is that this alternative formulation changes the intended meaning of guarded commands in combination with choice operators. Consider a process object $C()$ that is executing statement $[true]ch?m$ or $[true]ch'?m'$. If the rules of the layered semantics are applied, the process always leaves both alternatives open. It never chooses *a priori* whether it wants to receive message $m$ from channel $ch$ or message $m'$ from channel $ch'$. The actual choice depends on environment processes that are able to communicate with $C()$. In case of the alternative semantics the choice is made *a priori* by process $C()$ itself. If environment processes are only willing to send message $m$ on channel $ch$, a deadlock may occur if $C()$ chooses to receive $m'$ from $ch'$. This can never happen in case of the layered semantics.

The problem is caused by the fact that guarded commands and choice statements are orthogonal language constructs. A possible (partial) solution is to replace guarded commands and choice statements by so-called *select statements*. An example of such a select statement is

sel
  $[E_1]ch!m(E_2, E_3)$ then $S_1^p$
or
  $[E_4]ch'?m'(p_1)$ then $S_2^p$
or
  $[E_5]ch''!m''(E_6)$ then $S_3^p$
les

It is executed by first evaluating expressions $E_1, \cdots, E_6$ in the order of appearance. Each branch which guard evaluates to *false* is discarded. For a branch which guard evaluates to *bunk*, a non-deterministic choice is taken whether it is discarded. For each non-discarded branch, an attempt is made to execute the corresponding communication statement. If the attempt succeeds, the communication actually takes place and the rest statement (after the *then*) is executed. [2]

In fact, in imitation of POOL [PAR85], earlier versions of POOSL indeed incorporated select statements. The semantics of these statements was quite complex though. This complexity became unmanageable when message-receive statements $ch?m(p_1\ ,\ \cdots, p_m)$ where replaced by *selective* message-receive statements $ch?m(p_1, \cdots, p_m \mid E)$. The formalization of these statements in terms of the one-layered semantics appeared to be extremely cumbersome. For this reason it was decided to split the semantics in two layers. As a result, the select statements could be replaced by guarded commands and choice statements. Through this decision the semantics was considerably simplified. Furthermore, the introduction of guarded commands and choice statements increased the expressive power of POOSL.

## 4.3 Tail Recursion

Reactive behaviour of complex (real-time) hardware/software systems is often most naturally described in terms of finite state machine-like descriptions. Now each finite state machine is more or less expressible in terms of *if* and *do* constructs. However, the required conversions can be quite complicated and the results are often unreadable. For this reason, POOSL has incorporated a construct that allows state machine behaviour to be expressed directly and naturally. Consider the state diagram of a 1-place buffer *Buf* given in Figure 4.1. In process calculi such as CCS [Mil89], this behaviour is naturally specified as

$$\begin{aligned} Empty &\overset{def}{=} ch?in(data) \cdot Full(data) \\ Full(data) &\overset{def}{=} ch!out(data) \cdot Empty \end{aligned}$$

This specification can directly be translated into POOSL by representing states *Empty* and *Full(data)* by *methods*:

| | | |
|---|---|---|
| process class | *Buf* | |
| instance variables | | |
| communication channels | *ch* | |
| message interface | *ch?in(data) ch!out(data)* | |
| initial method call | *Empty* | |
| instance methods | **Empty** $\mid data \mid$ | **Full**(*data*) |
| | $ch?in(data); Full(data)$ | $ch!out(data); Empty$ |

[2]Remark that, although this execution resembles the execution of statement $[E_1]ch!m(E_2, E_3)\ S_1^p$ or $[E_4]ch'?m'(p_1)\ S_2^p$ or $[E_5]ch''!m''(E_6)\ S_3^p$ in the layered semantics, it is not the same. If one of the expressions has side-effects, different observable behaviour can be obtained.

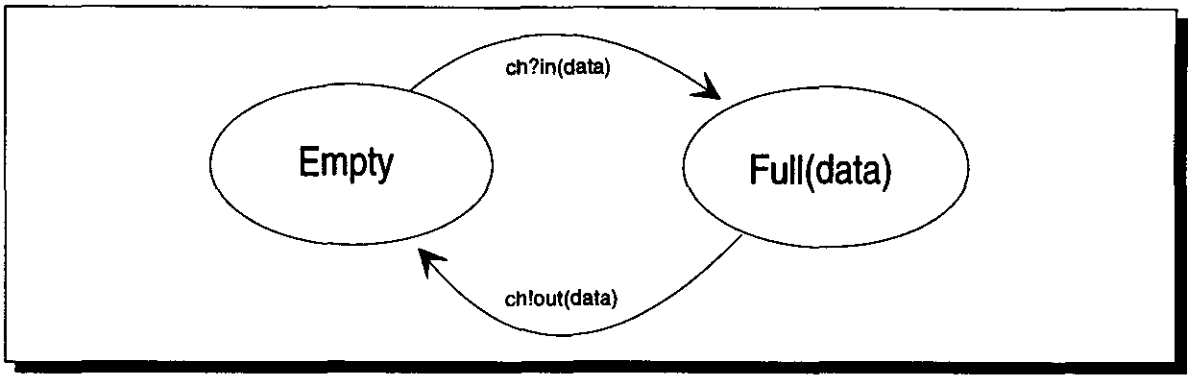

Figure 4.1: State Diagram of a 1-place Buffer

To describe *infinite*, *non-terminating* behaviour, methods *Empty* and *Full* are defined mutual recursively. But can methods really be defined this way without causing any problems? Unfortunately, the answer to this question is negative. Each time one method calls on another one, the depth of the process stack is increased by one. Since no method ever terminates, the stack will eventually grow beyond any bound. This problem has earlier been detected in [Blo93] in which context-free grammars are applied to specify the (infinite) behaviour of data communication protocols.

There exists a solution to this problem though. Suppose the buffer is currently executing method *Empty* and is ready to execute method call *Full*(*data*). Since this method call is not followed by any other statement (the call is tail-recursive) and since method *Full* as well as method *Empty* do not have any output parameters, the current local variable environment is never needed anymore. Upon invocation of method *Full* it can therefore be popped off the stack. Similarly, if the buffer calls on method *Empty*, the top of the stack can first be removed. In this way, the depth of the stack will never exceed 1.

The general solution to the unbounded stack problem is formalized in terms of rule g' of the labeled-transition system of Subsection 3.4.3. It states that if a method with no output parameters is called from another method with no output parameters in a tail-recursive way, the top of the process stack can first be removed.

# Chapter 5

# Conclusions

In this report we have developed a Plotkin-style *structural operational semantics* for the Parallel Object-Oriented Specification Language POOSL. The language is developed as a part of an object-oriented methodology for the analysis and design of data processing systems which contain a mix of software and hardware components.

We have explained that a formal semantics, and in particular a structural operational semantics, is of great importance during the process of language design and during the development of software tools such as simulators and compilers. Further, a formal semantics provides the means to reason about specifications, thereby offering possibilities of formal verification and correctness-preserving transformation.

The semantics consists of two parts. The *data* part, which is concerned with *data* objects or *traveling* objects, is a *computational* semantics. It emphasizes the *individual* steps needed to evaluate or execute data expressions respectively data statements. The semantics is specified by means of a *transition system*. We have clarified the formal description by an example in which we calculate the semantics of a complex-number expression.

The *process* part, concerning *process* objects, *clusters* and *system specifications*, is a computational *interleaving* semantics based on the communication model of CCS. The execution of a system of parallel objects is modeled as the *interleaving* of all atomic actions, i.e., as a sequential execution of these actions. The semantics is defined in terms of a *labeled* transition system.

On top of the operational semantics we have defined *observation equivalence*, a well-known and useful equivalence relation. We have given an example in which we prove that a simple handshake protocol and a 1-place buffer are observational equivalent. This ability to reason about specification equivalences is of vital importance in the context of *correctness-preserving transformations* which form an important part of the object-oriented design methodology.

In comparison to semantics found in literature, our semantics may seem rather complex. In literature, however, semantics are often based on toy languages or on simple and clean parts of realistic languages. In our case we have given a *full* semantics of a *complex* and *realistic* language. We think that this justifies the additional complexity.

# 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, 1990.
Report Series: Computing Science Note, nr. 90/12.

[ABK85] America, P. and J.W. de Bakker, J.N. Kok, J. Rutten.
*Operational semantics of a parallel object-oriented language.*
Amsterdam : Centre for Mathematics and Computer Science (CWI), 1985.
CWI Report CS-R8515.

[Apt83] Apt, K.R.
*Formal justification of a proof system for communicating sequential processes.*
Journal of the Association for Computing Machinary, Vol. 30(1983), no. 1, p. 197–216.

[Apt81] Apt, K.R.
*Recursive assertions and parallel programs.*
Acta Informatica, Vol. 15(1981), p. 219–232.

[AR89] America, P.H.M. and J.J.M.M. Rutten.
*A parallel object-oriented language: Design and semantic foundations.*
Amsterdam : Centre for Mathematics and Computer Science (CWI), 1989.
CWI Report CS-R8953.

[Blo93] Bloks, R.H.J.
*A grammar based approach towards the automatic implementation of data communication protocols in hardware.*
Ph.D. thesis, Eindhoven University of Technology, 1993.

[Eij89] Eijk, P.H.J. van.
*The design of a simulator tool.*
In: The formal description technique LOTOS. Ed. by P. van Eijk, C. Vissers and M. Diaz. Amsterdam : North-Holland, 1989. P. 351–390.

[EVD89] Eijk, P.H.J. van and C.A. 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.

[Hen90] Hennessy, M.
*The semantics of programming languages: An elementary introduction using structural operational semantics.*
Chichester : Wiley, 1990.

[HP79] Hennessy, M. and G.D. Plotkin.
*Full abstraction for a simple parallel programming language.*
In: Proceedings of the 8th conference on Mathematical Foundations of Computer Science (MFCS'79), Olomouc, Czechoslovakia, September 3-7, 1979. Ed. by J. Becvar. Berlin : Springer, 1979 (Lecture Notes in Computer Science, Vol. 74). P. 108-120.

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

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

[Mil83] Milner, R.
*Calculi for synchrony and asynchrony.*
Journal of theoretical computer science, Vol. 25(1983), p. 267-310.

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

[Par81] Park, D.M.R.
*Concurrency and automata on infinite sequences.*
In: Proceedings of the 5th German Informatics Society Conference, Karlsruhe, Germany, March 23-25, 1981. Ed. by P. Deussen. Berlin : Springer, 1981 (Lecture Notes in Computer Science, Vol. 104). P. 167-183.

[Plo81] Plotkin, G.D.
*A structural approach to operational semantics.*
Aarhus : University of Aarhus, Department of Computer Science, 1981.
Technical Report DAIMI FN-19.

[Plo83] Plotkin, G.D.
*An operational semantics for CSP.*
In: Formal description of programming concepts II. Ed. by D. Bjørner. Amsterdam : North-Holland, 1983. P. 199–223.

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

[Ten91] Tennent, R.D.
*Semantics of programming languages.*
Worcester, UK : Prentice-Hall, 1991.

[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.

[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.

[Voe95] Voeten, J.P.M.
*POOSL: An object-oriented specification language for the analysis and design of hardware/software systems.*
Eindhoven: Eindhoven University of Technology, Faculty of Electrical Engineering, 1995.
EUT Report 95-E-290.

ISSN 0167-9708
Coden: TEUEDE

(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

(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

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

Faculty of Electrical Engineering

(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

(291) Smeets, B.H.T. and M.H.J. Bollen
STOCHASTIC MODELLING OF PROTECTION SYSTEMS: Comparison of four mathematical techniques.
EUT Report 95-E-291. 1995. ISBN 90-6144-291-5

(292) Voeten, J.P.M. and A. van Rangelrooij
CCS AND TIME: A practical and comprehensible approach to a performance evaluation of finite state CCS descriptions.
EUT Report 95-E-292. 1995. ISBN 90-6144-292-3

(293) Voeten, J.P.M.
SEMANTICS OF POOSL: An object-oriented specification language for the analysis and design of hardware/software systems.
EUT Report 95-E-293. 1995. ISBN 90-6144-293-1