

The Design of Distributed Systems — An Introduction to FOCUS*

- Revised Version -

Manfred Broy
Max Fuchs

Frank Dederichs
Thomas F. Gritzner

Claus Dendorfer
Rainer Weber

Institut für Informatik
Technische Universität München
Postfach 20 24 20, 8000 München 2

* This work is supported by the Sonderforschungsbereich 342 “Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen”

Contents

| | | |
|----------|--|-----------|
| 1 | Methods for System Development | 1 |
| 1.1 | Aspects of Systems and System Models | 2 |
| 1.2 | System Models and Specification in FOCUS | 4 |
| 1.3 | Phases of Development in FOCUS | 5 |
| 1.4 | How to read this Report | 9 |
| 2 | Trace Specification | 10 |
| 2.1 | Overview | 10 |
| 2.2 | The Basic Structure: Streams | 12 |
| 2.3 | Specification of Actions | 13 |
| 2.4 | Global Specification | 14 |
| 2.4.1 | Trace Logic | 15 |
| 2.4.2 | Transition Systems | 17 |
| 2.5 | Component-oriented Specification | 19 |
| 2.5.1 | Motivation | 19 |
| 2.5.2 | Formal Treatment | 19 |
| 2.6 | Timed Trace Specification | 21 |
| 2.7 | Proof Principles | 22 |
| 3 | Functional Specification | 25 |
| 3.1 | Overview | 25 |
| 3.2 | Actions, Channels and Messages | 27 |

| | | |
|----------|---|-----------|
| 3.3 | Stream Processing Functions | 29 |
| 3.4 | Specification of Components | 30 |
| 3.5 | State-oriented Functional Specification | 33 |
| 3.6 | Specification of Networks | 36 |
| 3.6.1 | Equational Definitions | 36 |
| 3.6.2 | Compositional Forms | 37 |
| 3.7 | Refinement | 39 |
| 3.8 | Timed Component Specification | 43 |
| 3.9 | Proof Principles | 44 |
| 4 | Implementation | 47 |
| 4.1 | Overview | 47 |
| 4.2 | An Applicative Language | 49 |
| 4.3 | A Procedural Language | 54 |
| 4.4 | Transformational Synthesis of Concrete Programs | 57 |
| 5 | Conclusion | 61 |
| 6 | Glossary | 62 |

Abstract

FOCUS is a framework for the systematic formal specification and development of distributed interactive systems and their components. FOCUS provides models, formalisms and verification calculi for the stepwise specification and development, transformation and verification of such systems. FOCUS aims at the modular development and implementation of distributed interactive systems through several abstraction levels by stepwise refinement.

Chapter 1

Methods for System Development

A *(distributed) system* consists of a family of interacting, conceptually or spatially distributed *components*. A *system development method* provides a framework for organizing the stepwise construction of such systems. During the development process several descriptions are produced, that reflect different abstraction levels. Only if formal techniques are used these descriptions can be made as precise and unambiguous as necessary. Moreover, formal techniques allow to establish formal relationships between descriptions that belong to different levels. A piece of software for a distributed system is a formal description in our sense, too.

The system development method FOCUS that is outlined in the following provides:

- formalisms for the representation of distributed systems at different abstraction levels,
- advice at which level which system properties should be fixed,
- concepts for the relationship between the different levels,
- techniques that support the transition from one level to another.

In general, a development method identifies activities to be carried out as well as their objectives and their goal-directed organisation. Development activities are related to the following notions, which can be seen as development method “in the large”:

- analysis,
- specification,
- motivation and explanation,
- documentation,
- validation,
- transformation,
- justification,

- implementation,
- verification,
- integration.

In addition, a methodological framework provides techniques for the development “in the small”, including hints how to analyse, specify and verify particular system properties at particular abstraction levels.

Besides this, questions of organisation and management, of economics and teamwork, of resources and technical devices etc. have to be handled. They are not treated explicitly in the following. Moreover, the step from informal to formal descriptions is not tackled here.

Before entering into the presentation of FOCUS, let us first have a look at some major concepts underlying our approach.

1.1 Aspects of Systems and System Models

There is no principal difference between the techniques of classical software engineering, i.e. techniques for the production of sequential software, and the techniques for the production of distributed systems. However, certain notions like *safety/liveness*, *assume/commit* and *system/environment* are essential for distributed systems only. Moreover, distributed systems tend to be more complex and more difficult to develop than sequential software. In the sequential as in the distributed case, a system design is organised as a sequence of increasingly detailed descriptions of the involved system components. The final result will be a constructive description (usually a *program*) of those components that are to be implemented.

System models are formal, mathematical structures representing particular aspects of a (real or planned) system while abstracting from others. Thus every system model defines a certain abstraction level. In models for distributed systems often the following aspects are represented in more or less detail:

- *Spatial distribution*: a system can be “distributed” in space or conceptually in the sense that it is composed of subsystems called *components*. Actions may be associated with these components. States may be decomposed into substates that belong to these components.
- *Causality, interaction, synchronization*: the execution of actions may depend on the previous execution of other actions (or on the system’s state). This is called a *causal dependency*. Causal dependencies between actions that belong to different components are called *interactions*. Interaction (for instance by message passing or by accessing shared storage) is the only way by which

components can synchronize and coordinate their work while accomplishing a common task.

- *Concurrency*: typically, some actions may be carried out independently and simultaneously. Then they are called *concurrent*. When system runs are represented by (finite or infinite) action sequences, concurrency is modelled implicitly by *interleaving*. In such a sequence concurrent actions may occur in any order. This way concurrency is reduced to *nondeterminism*. Alternatively, if the system model allows to express (in a single run) that two actions may be carried out independently, it is said that the model includes *explicit* (“true”) concurrency.
- *Nondeterminism*: in most systems choices occur. A choice is called *nondeterministic* if it cannot be influenced from outside or if it depends on factors not explicitly represented in the system model.

In principle distributed systems interact by *asynchronous communication*, *synchronous communication* (for instance by shared actions), or via *shared states*. Roughly speaking two complementary types of system models can be distinguished:

- *action-oriented models* (with or without shared actions),
- *state-oriented models*.

In action-oriented models a system is described by specifying the set of actions it might exhibit and the causal relationships between these actions within system runs (histories, traces). In a state-oriented model a system is described by specifying its state space and its dynamic behaviour in terms of these states. The two views indeed are dual: an action can be viewed as a relation on states, and distinguished relations on states can be viewed as actions. Of course there are models combining both views.

The overall behaviour of a system often depends on the behaviour of its environment. Basically there are two ways to capture such dependencies. On the one hand one can explicitly include the environment into the system representation, thus considering it a particular system component. This leads to a *closed system view*, where everything relevant is included. A closed system forms an isolated complex that executes actions and assumes states without being influenced whatsoever. On the other hand one can refrain from including the environment into the system description explicitly, rather admitting the system to be influenced from outside. This leads to an *open system view*. An open system reacts on environment stimuli and these reactions in turn usually have impact on the environments behaviour. In this sense every (sub)component of a system is an open system again.

A description how an open system is influenced by its environment is called an *interface specification*. Particular actions can now be attributed to the environment, thus called *environment actions*, while others are controlled by the system, thus

called *system actions*. These two action sets are described in the syntactic part of an interface specification. Accordingly, the semantic part is given by specifying causal relationships between system and environment actions in system runs. In a state-oriented view we may similarly specify which parts of the state may be changed by the environment and which parts may be changed by the system.

There are several possibilities to derive an open system view systematically from a closed system view and vice versa. The step from a closed to an open view is essential, since it is the decisive step towards a *modular description*. Interface specifications of open systems (their semantic part to be precise) are sometimes written according to the *assume/commit* style. This style reflects the fact that open systems are usually not supposed to run in arbitrary environments but only in those, which fulfil certain assumptions. Moreover one often distinguishes between *safety* and *liveness* properties. Technically, a closed system can be considered a special case of an open system, namely an open system with an “empty” environment.

1.2 System Models and Specification in FOCUS

In the following our development method called FOCUS is outlined. FOCUS is a (mainly) action-oriented model that admits the description of open as well as the description of closed systems. Although it is geared towards systems that communicate by asynchronous message passing, in principle shared state programs can also be modelled. At the level of system runs concurrency is represented by interleaving. The basic notion of FOCUS are finite and infinite sequences (called *streams*) of elements from given carrier sets. One can distinguish between two different types of streams: streams of *actions* and streams of *messages*. Since in our framework actions and messages are closely related, we will be very liberal in our terminology and notation.

Streams of actions, which we call *traces*, are mainly used to model the behaviour of closed systems. Sets of traces are described by predicates or state transition systems. *Predicates* describe trace sets in a straightforward way. *State transition systems* are more implicit descriptions of trace sets.

Streams of messages are used to represent *communication histories* of channels. We distinguish between input and output channels. These channels are the communication links between system components.

The behaviour of a system component (as well as the behaviour of an open system that is connected to its environment by channels) is described in FOCUS mainly by logical formulas specifying *stream processing functions*. A stream processing function maps tuples of input streams on tuples of output streams. In the literature one can find many different names for the entities we have called system

components above, e.g. process, task, module. In the sequel besides component we sometimes use the term *agent*.

In the beginning of a development process one is not concerned with internal details of a system component but rather with its interface. In an interface specification all actions are considered which are relevant for the interaction between the component and its environment. In addition, the causal relationship between system and environment actions are specified (see section 1.1).

In the implementation phases algorithmic languages are used to describe system components. This way components are represented by pieces of program code. The semantics of these code fragments is again given by sets of stream processing functions. This way we obtain a coherent formal framework basically using the same notions.

All in all there are several ways to specify system components in FOCUS:

- by (sets of) traces of input and output actions,
- by state transition systems,
- by (sets of) stream processing functions,
- by executable descriptions in terms of programming languages.

All these formalisms reflect different abstraction levels and are thus used in different phases during the development process. It is part of the methodological framework to provide means to go from a trace oriented specification to a functional specification and finally to an executable descriptions in terms of an adequate algorithmic language.

1.3 Phases of Development in FOCUS

System development in FOCUS is organized into a number of phases, namely (see also figure 1.1):

- requirements specification,
- design specification,
- abstract implementation,
- concrete implementation.

In a formal *requirements specification* the properties are formalized that are relevant for the envisaged system from the customers point of view. It is decisive to formulate and validate a requirements specification carefully, since it is the first formal description of the customer's wishes and the starting point of the entire development process. Since it is derived from informal descriptions there is no way of formally verifying its correctness. Instead some evidence has to be given, for instance by a number of case demonstrations, that the formal specification

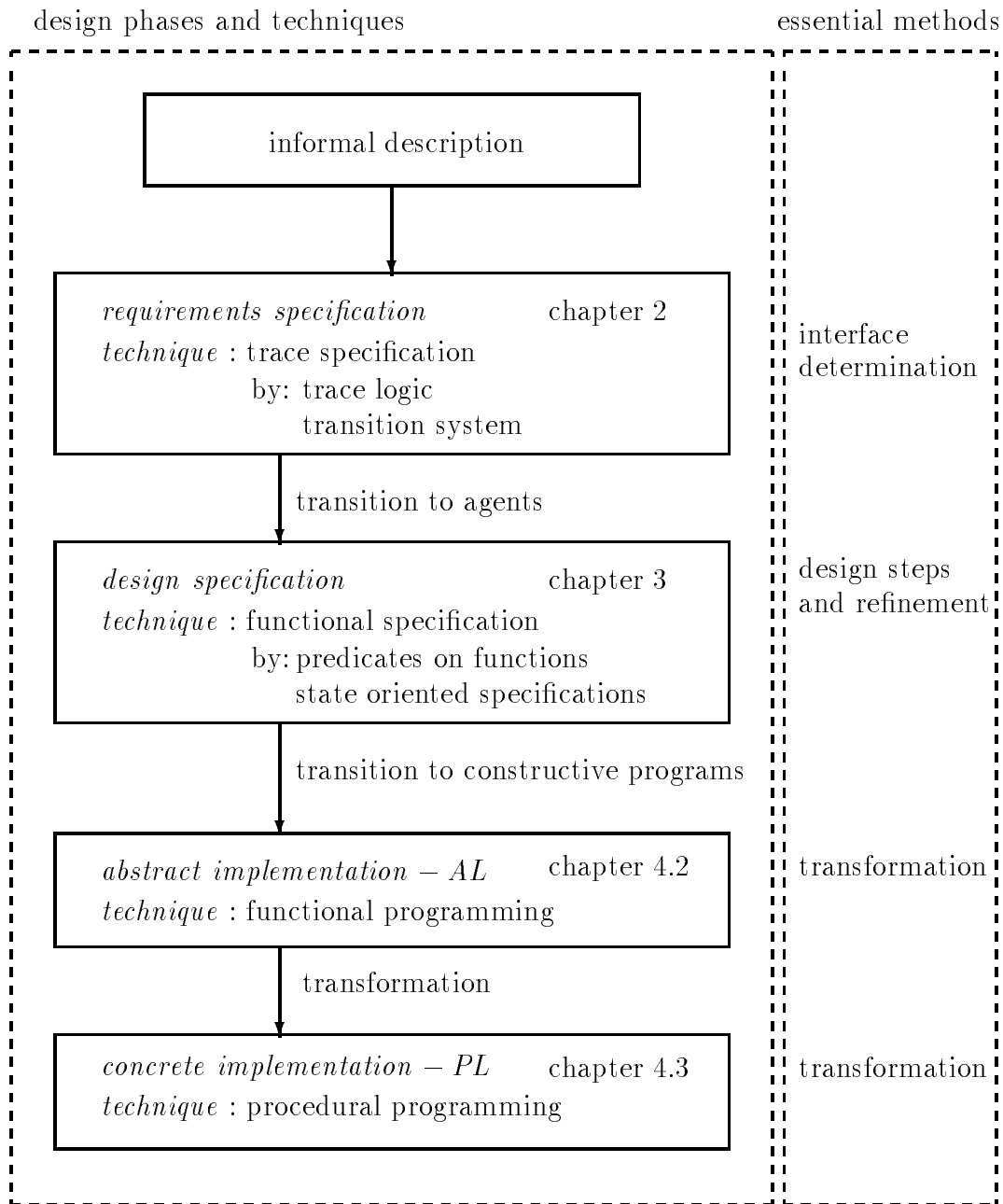


Figure 1.1: Overview of FOCUS

actually captures the informal requirements. Technically within a requirements specification first all relevant data structures and are described by algebraic specifications, (for instance, using the algebraic specification language SPECTRUM as described in [BFG⁺92]). One may aim at a closed or an open system description. A closed view might be appropriate, if the interface between the envisaged system and its environment is not clear yet. In this case, determining this interface is an important task of requirements engineering; after that, an open system view is fixed.

In FOCUS a requirements specification of a closed system is given by a *trace specification*. Such a specification formally describes a set of actions and a set of traces. If the interface is clear right from the beginning, an open system specification will be the starting point, thus skipping the closed system description. A requirements specification of an open system may be given again either by a trace specification or by a functional specification in terms of stream processing functions.

Having finished the requirements specification, in a *design specification* the step-wise refinement of those system components which are to be implemented is the point of interest. This includes the introduction of input and output channels, often a transition from non-constructive to constructive specifications, and usually a further decomposition of components into a number of subcomponents. In FOCUS a design specification is given by a network of system components (agents) connected by channels. The first step in the design specification is to determine the exact number and use of communication channels for each of these components. Then every single component is modelled by a set of stream processing functions mapping the communication histories (i.e. streams) of the input channels on the communication histories of the output channels. A set of functions is used instead of a single function to model nondeterministic components and to enable underspecification, leaving a spectrum of possibilities for the subsequent implementation. Formally, a component specification is a predicate on stream processing functions. The design specification has to be verified with respect to the preceding requirements specification. In particular, it has to be proved that the behaviour generated by the interaction of all components meets the overall requirements specified on the previous level.

Descriptions of system components that are formalized in the design phase do not have to be constructive. However, during the development process, new versions of these specifications are derived, which are closer to an executable program. For example, introducing a notion of state often leads to constructive forms. However, for arbitrary design specifications there is no way to generate the system behaviour algorithmically from these descriptions. Thus, a further development steps are necessary.

These steps to an *abstract implementation*. In FOCUS a particular applicative language called AL is used to represent abstract programs. It is tailored to fit

in the general framework of streams and stream processing functions. In general, abstract programs are not tuned towards a specific machine. They leave room for a lot of optimizations and efficiency increasing transformations. These aspects are treated when a concrete program is derived from an abstract one.

In a *concrete implementation* basically more efficient representations of the data structures are chosen and more efficient algorithms are given. Data structures and algorithms are formulated in a way tuned towards particular machines or programming languages. One way to obtain more efficient programs (at least for many existing machines) consists in replacing applicative descriptions by procedural ones. Therefore FOCUS offers a procedural language called PL for concrete programs. Like AL it is carefully embedded in the formal framework. The step from abstract to concrete descriptions can be done using program transformation techniques or by freely constructing the procedural version and then proving its correctness with respect to the applicative one.

Up to now we only mentioned the standard way of system development in FOCUS. However, depending on the particular application, certain deviations may be appropriate: if the overall structure of the planned system is already given and the interfaces of the components to be implemented are given in an explicit form, the design specification is the appropriate starting level. The standard exit point of our methodology is at the concrete implementation level. This is easily motivated by the need for efficient programs. Nevertheless an implementation in a certain concrete language different from PL can be required in the informal description. In this case the abstract implementation level is a suitable exit point.

From our point of view it does not make sense to exit already before an abstract implementation is gained when aiming at program development. Furthermore we do suggest not to start the specification process on levels lower than the design specification level, to keep the first formalisation abstract and comprehensible.

All development steps can in general not be done in a purely automatic way. Accordingly, the step from a higher to a lower level cannot be carried out just by a machine. However, if done in an appropriate way, all steps can be formally justified, which means that they can be verified with respect to the previous levels. Of course, it is advisable to do both development and verification hand in hand, since design and verification steps in general are closely related.

This brings us to the concepts of *refinement* and *verification*. *Refinement* is the basic notion of system development in FOCUS. In every refinement step a given description of a system or of one of its components is replaced by a refined description. One can distinguish several concepts of refinement. In its simplest form we have to show that the formulas describing the properties of the refined system representation imply the properties of the original one.

All refinement steps in FOCUS can in principle be done using classical predicate



logic. However, it is often appropriate to apply more schematic rules. The FOCUS development method gives guidance to refinement steps.

Having obtained two different system descriptions, where one is claimed to refine the other, we have to verify this claim. This means we have to show that the refined version has all the properties required by the previous version. How such a verification is carried out strongly depends on the way in which the requirements are described. For different description styles different verification techniques have to be used. FOCUS offers a number of specific proof principles.

1.4 How to read this Report

This report contains a description of the design method FOCUS. We included some small examples to illustrate key techniques. In particular, a simple message transmission system appears as a running example. Several case studies have been made with FOCUS; [BDD⁺92] contains a summary. We just mention two of them: the specification of a lift controller in [Bro88a] and the development of an implementation of the so-called Stenning-protocol in [DW92].

The structure of FOCUS is also mirrored in the organization of this report: trace specifications (used for requirements specifications) are treated in chapter 2, functional specifications (used for design specifications) are dealt with in chapter 3, chapter 4 is concerned with implementation (both abstract and concrete implementations). Chapter 5 presents conclusions.

To improve the readability of the report we have marked certain paragraphs: the label  indicates essential features of our methodology, often recipes for its use. The paragraphs comprising theoretical aspects, which are not necessary for a first understanding, are marked with .

Acknowledgement

We gratefully acknowledge helpful discussions with our colleagues from the Sonderforschungsbereich 342.

Chapter 2

Trace Specification

2.1 Overview

Trace specifications describe the behaviour of distributed systems in a very abstract way. They are well-suited for formalising requirements.



A *trace specification* describes the set of all runs of a distributed system by sequences of *actions (traces)*.

Actions (sometimes also called events) constitute the key concept of trace specifications. They represent the basic activities in a system, like “sending a message” or “pressing a button”. Actions are thought to be atomic and instantaneous. For the present, we do not care where in a system a particular action occurs, which entity generates it, and what its effect is.

A *trace* represents a record of a run (history) of the system. This is more pictorially described in [Hoa85], p. 41:

“Imagine there is an observer with a notebook who watches the process and writes down the name of each event as it occurs. We can validly ignore the possibility that two events occur simultaneously; for if they did, the observer would still have to record one of them first and then the other, and the order in which he records them would not matter.”

As unbounded and infinite behaviour is a typical phenomenon of distributed systems, both finite and infinite traces are considered. Based on a possibly infinite set of actions Act , the set of traces, i.e. *streams of actions*, is denoted by

$$Act^\omega = Act^* \cup Act^\infty,$$

where Act^* denotes the finite traces, and Act^∞ the infinite ones.

Trace specifications provide *action oriented* models of distributed systems in contrast to *state oriented* formalisms like, for instance, temporal logic. However, we

shall present means to support a state-oriented specification style as well for certain system properties.

We distinguish two different kinds of trace specifications. The first is called *global specification*. It gives a description of a system as an unstructured entity and does not take any decomposition into components into account. The second is called *component-oriented specification*. Here a system is modelled as being composed of several interacting components. In a component-oriented specification there may be two kinds of requirements: global and local requirements. A *global* requirement refers to more than one component, possibly to the whole system. A *local* requirement applies to just one component. A component is described by its input and output actions and its behaviour which is represented by a set of traces of input and output actions. In the preceding chapter we have called this an *interface specification*. Components communicate via input and output actions asynchronously with each other. Accordingly, a component may issue an output action (send a message) and continue to work without waiting until the communication partner(s) is (are) ready to receive the message.

Methodologically we first give a global specification, which corresponds to a closed system view, and then switch to a component-oriented view, which corresponds to a closed system view. This usually includes some design steps. Starting from a global specification is considered appropriate by various researchers in the fields of formal specification and requirements engineering (e.g. [CM88], [DHR90]), because customers initially often do not state the obligation of each system component, but rather state what the global objectives of the whole system are.

In the course of program development we use traces for the *requirements specification*. We proceed as follows:



The starting point are informal requirements by the customer.

1. We first formalize the informal requirements in a *global specification*.
2. We then produce a *component-oriented specification*. It is based on the global specification but contains additional structuring information.
3. *Design steps* are then carried out to gradually localize all global requirements.

The end point is a local specification of each component, which may then be further developed independently of each other.

In the following we first present the basic structure that is underlying all formal models used in Focus: streams (section 2.2). After that we consider the specification of actions (section 2.3), then we illustrate the formalisms and methods used for the global specification (section 2.4) and the component-oriented specification (section 2.5). An extension of the trace formalism for describing time sensitive systems is sketched (section 2.6). For a completely formal program development,


besides specification techniques also proof methods are necessary; we present proof methods based on traces (section 2.7).

2.2 The Basic Structure: Streams

The data type of *streams* is fundamental in FOCUS. Streams appear as streams of *actions* (shortly, *traces*) in trace specifications and streams of *messages* in functional specifications and programs. Given some set of items S (in FOCUS mainly actions or messages), the set of streams over S is denoted by S^ω ; it is the union of the set of finite and infinite sequences: $S^\omega = S^* \cup S^\infty$.

There are several basic operations and relations concerning streams (let s, t, u be streams and a, b, c be items):

- $\langle \rangle$ denotes the empty stream.
- $\langle s_1, \dots, s_n \rangle$ denotes the stream containing the elements s_1, \dots, s_n .
- $ft(s)$ yields the first element of s , if s is not empty, otherwise it yields \perp (“undefined”).
- $rt(s)$ yields the stream in which the first element of s is deleted.
- $a \& s$ denotes the stream in which a is prefixed to s . If a is defined, i.e. $a \neq \perp$, we have $ft(a \& s) = a$ and $rt(a \& s) = s$.
- sot denotes the concatenation of s and t . If s is infinite, then sot just yields s . We frequently write aos for $a \& s$ and aob for $a \& b \& \langle \rangle$, thus identifying items with streams of length 1.
- $s \sqsubseteq t$ denotes that s is a *prefix* of t , which is formally expressed by $\exists u : sou = t$. The prefix order is canonically (by pointwise application) extended to tuples of streams and to functions producing streams as results.
- $a \text{ in } s$ yields *true* exactly if a occurs in s .
- $\#s$ gives the length of s , which may also be ∞ (“infinite”).
- $a \odot s$, the filter operation, yields the substream of s that consists of a -items only, for instance, $a \odot \langle a, b, a, c \rangle = \langle a, a \rangle$. As a generalization, the first operand may also be a *set* of items, for instance, $\{a, b\} \odot \langle a, b, a, c \rangle = \langle a, b, a \rangle$.

 Our methodology is based on certain mathematical concepts which we briefly introduce in the following. For a detailed explanation of these concepts confer e.g. [LS87], from which the definitions below have been taken.


Definition 2.1 (partial order): A *partial order* is a pair (D, \sqsubseteq) with a set D and a relation $\sqsubseteq \subseteq D \times D$ such that \sqsubseteq is *reflexive* (i.e. $d \sqsubseteq d$ for all $d \in D$), *antisymmetric* (i.e. if $d_1 \sqsubseteq d_2$ and $d_2 \sqsubseteq d_1$, then $d_1 = d_2$ for all $d_1, d_2 \in D$) and *transitive* (i.e. if $d_1 \sqsubseteq d_2$ and $d_2 \sqsubseteq d_3$ then $d_1 \sqsubseteq d_3$ for all $d_1, d_2, d_3 \in D$). \square


Definition 2.2 (least upper bound): Let (D, \sqsubseteq) be a partial order and S a (possibly empty) subset of D . An element $u \in D$ is said to be an *upper bound* of S (in D), if $d \sqsubseteq u$ for all $d \in S$; u is said to be the *least upper bound (lub)* of S (in D), if u is the least element of the set of all upper bounds of S in D . The least upper bound is denoted $\bigsqcup_D S$ or, shortly, $\bigsqcup S$, provided it exists. \square

Definition 2.3 (chain): Let (D, \sqsubseteq) be a partial order. A non-empty subset S of D is called a *chain* in D if $d \sqsubseteq d'$ or $d' \sqsubseteq d$ (or both) holds for every two elements $d, d' \in S$. Said another way, S is a chain if the order relation ' \sqsubseteq ' restricted to S is total. \square

Definition 2.4 (complete partial order): A partial order (D, \sqsubseteq) is a *complete partial order (cpo)* if the following two conditions hold:

- (1) The set D has a least element. This element is denoted by \perp_D or simply by \perp (read 'bottom').
- (2) For every chain S in D the least upper bound $\bigsqcup S$ exists. \square

 Streams with the prefix relation \sqsubseteq constitute a complete partial order with least element $\langle \rangle$.

 Streams may be specified by axiomatic techniques using algebraic specification, see [Bro89]. Infinite objects (like infinite streams) are somewhat unusual in the framework of algebraic specification. In fact, the notion of a model for an algebraic specification needs some slight modification to overcome this difficulty: all sorts are interpreted as complete partial orders.

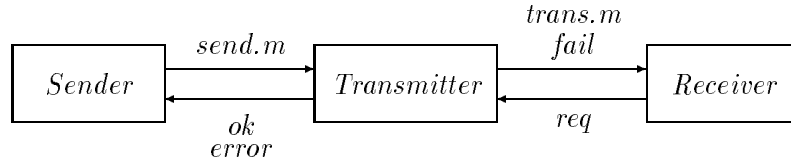
2.3 Specification of Actions



The first step in writing a trace specification is to define which *actions* occur in the system.

Example 2.1 (A simple message transmission system):

Throughout this paper we use the example of a simple *message transmission system* to illustrate our methods. (For more complex examples we recommend [Bro88b] and [DW92]; see also [BDD⁺92] for a current summary of all FOCUS case studies.) The message transmission system consists of a *sender* and a *receiver* which are connected by a buffer component called (*transmitter*). The following picture illustrates this situation:



The sender may send messages to the transmitter; each send-message is acknowledged with either *ok* or *error* by the transmitter. If the acknowledgement is *ok*, the message is stored in the transmitter, otherwise it is discarded. The receiver issues requests for messages to the transmitter, which are answered with the transmission of a message or *fail*, if no message is available.

Below we shall have a closer look at the requirements on this communication system. At the moment we only state which actions may occur in our system. Given a set M of messages, we define the actions of the system as follows:

send actions: $Send = \{send.m \mid m \in M\}$,
 transmit actions: $Trans = \{trans.m \mid m \in M\}$,
 altogether: $Act = Send \cup Trans \cup \{ok, error, fail, req\}$.



In more complex cases we suggest to use *algebraic specifications* to specify action sets and data sets, such as the set M . □

2.4 Global Specification

A *global trace specification* describes the behaviour of a distributed system, not taking any structuring into components into account yet. At this level, the system behaviour is completely defined by its set of traces.

Basically, we distinguish two different styles of trace specification. The first is called *history-oriented*. Here we impose restrictions on the complete histories (runs, traces) of the system. The second is called *transition-oriented*. Here the behaviour is specified by defining how the system evolves in a stepwise manner, starting from some initial state.



In FOCUS we may use two different means to specify trace sets:

1. *trace logic* to support a *history-oriented* specification style,
2. *transition systems* to support a *transition-oriented* style.

These means may be combined: It is possible to describe some system properties by trace logic, and others by transition systems.

2.4.1 Trace Logic

We use the term *trace logic* for a (many-sorted) predicate logic in which we have a particular sort for traces (or many trace sorts, if we want to distinguish different kinds of traces, see the next section).



Using *trace logic*, the runs of a distributed system are characterized by giving a logical formula with a free variable of sort Act^ω . This is called a *trace predicate*. Every trace that satisfies this predicate represents a run of the system.

Predicate logic offers sufficient expressiveness and is a well-understood subject. A pragmatic advantage is that it is in widespread use (at least in the computer science, engineering and natural sciences community).

There are two different kinds of system properties: Some of them refer to all finite prefixes of a trace and have an invariant-like character whereas others apply to the trace as a whole. The first are called *safety properties*; these are properties whose violation can always be detected after a finite amount of time, i.e. by considering a sufficiently large but finite prefix of a trace. Formally, a safety property is a predicate P that satisfies the following condition

$$\forall t \in Act^\omega : (P(t) \Leftrightarrow \forall s \in Act^* : s \sqsubseteq t \Rightarrow P(s)).$$

This means that a trace of the system is safe with respect to the property P if and only if all finite prefixes of the trace are safe.

The second kind of properties are called *liveness properties*; in contrast to safety properties the violation of liveness properties can only be detected after a complete, possibly infinite observation, i.e. by considering a complete possibly, infinite trace. Formally a liveness property is a predicate P that satisfies the following condition:

$$\forall s \in Act^* : \exists t \in Act^\omega : P(s \circ t).$$

Thus every finite partial trace can be extended to become a trace that is correct with respect to the liveness property P .

Example 2.2 (The message transmission system described by trace logic):

The allowed traces of our message transmission system are those that fulfil the predicates S and L defined below:

$$T = \{t \in Act^\omega \mid S(t) \wedge L(t)\}.$$

The safety requirements are verbally expressed as follows:

- There do never occur more send acknowledgements (*ok*, *error*) than send actions (*send.m*).

- There do never occur more transmit acknowledgements ($trans.m, fail$) than transmit requests (req).
- Only those messages are transmitted which were sent before and which were acknowledged with ok ; moreover, the messages are transmitted in the same order as they were sent.

Formally these requirements are captured by the following predicate S :

$$\begin{aligned}
S(t) \equiv & \forall t' \in Act^* : t' \sqsubseteq t \Rightarrow \\
& \#\{ok, error\} \odot t' \leq \#Send \odot t' \wedge \\
& \#(Trans \cup \{fail\}) \odot t' \leq \#req \odot t' \wedge \\
& msg(Trans \odot t') \sqsubseteq success(msg(Send \odot t'), \{ok, error\} \odot t'),
\end{aligned}$$

where msg and $success$ are defined as follows (let w be a stream of $send$ - and $trans$ -actions, x be a stream of messages, y be a stream of ok - and $error$ -actions):

$$\begin{aligned}
msg(\langle \rangle) &= \langle \rangle, \\
msg(send.m \& w) &= m \& msg(w), \\
msg(trans.m \& w) &= m \& msg(w).
\end{aligned}$$

msg filters the messages that were sent or transmitted in a trace.

$$\begin{aligned}
success(x, y) &= \langle \rangle, \quad \text{if } x = \langle \rangle \vee y = \langle \rangle, \\
success(m \& x, ok \& y) &= m \& success(x, y), \\
success(m \& x, error \& y) &= success(x, y).
\end{aligned}$$

Given a message stream x and a stream y of send-acknowledgements, $success(x, y)$ gives the message substream of x consisting of those messages that were acknowledged with ok . These are just those messages which are accepted by the transmitter.



It is considered one of the advantages of FOCUS that auxiliary functions and operations can be introduced whenever needed. We recommend to introduce a number of auxiliary functions and operations to keep a specification readable.

The informal liveness requirements read as follows:

- Send acknowledgements are not delayed for an infinite amount of time.
- Transmit acknowledgements are not delayed for an infinite amount of time.
- If an item is sent infinitely often and it is requested infinitely often, then it is finally transferred from the sender to the receiver.

These three verbal requirements are captured by the following predicate L :

$$\begin{aligned}
L(t) \equiv & \#Send \odot t = \infty \Rightarrow \#\{ok, error\} \odot t = \infty \wedge \\
& \#req \odot t = \infty \Rightarrow \#(Trans \cup \{fail\}) \odot t = \infty \wedge \\
& \#Send \odot t = \infty \wedge \#req \odot t = \infty \Rightarrow \#Trans \odot t = \infty.
\end{aligned}$$

Note that in the definition of S all finite approximations t' of t were considered, whereas L is only concerned with the entire trace t . \square

The example shows that a trace predicate can be composed from several more simple predicates, each representing a particular requirement. This way an appropriate structuring of the specification can be achieved. This is quite useful for requirements specifications: a trace specification can easily be extended, new requirements may simply be added, obsolete requirements may be deleted without changing the rest of the specification.

2.4.2 Transition Systems

By *transition systems* (also called *automata* or *state machines*), a model of a distributed system is given in terms of states and state transitions which are caused by actions. A state comprises a snapshot of the system, and state transitions indicate how the system may evolve from one snapshot to the next by executing an action.

Formally, a transition system is a tuple $(Act, State, \rightarrow, Init)$ with

| | |
|---------------|--|
| Act | a set of actions, |
| $State$ | a set of states, |
| \rightarrow | $\subseteq State \times Act \times State$, the transition relation, |
| $Init$ | $\subseteq State$, the set of initial states. |

We also write $\sigma \xrightarrow{a} \sigma'$ for $(\sigma, a, \sigma') \in \rightarrow$. Given a transition system, its *executions* are sequences of states and actions of the form

$$\sigma_0 \xrightarrow{a_1} \sigma_1 \xrightarrow{a_2} \sigma_2 \cdots,$$

where σ_0 is an initial state and $\sigma_i \xrightarrow{a_{i+1}} \sigma_{i+1}$ holds for all i . The sequence may be infinite or finite, in the latter case it ends with a state. Given such an execution, its trace is just $a_1 \circ a_2 \circ \cdots$. The traces described by a transition system consist of the traces of all executions of the system. Obviously if $a_1 \circ a_2 \circ \cdots \circ a_{n-1} \circ a_n$ is a trace of a transition system, so is the trace $a_1 \circ a_2 \circ \cdots \circ a_{n-1}$. Thus, in FOCUS transition systems are used to describe safety properties.

Note that the states used here are *global* states. In a distributed system, states will be distributed; nevertheless, global states are a convenient conceptual tool in the development of a global trace specification. Aspects of logical or physical distribution are only considered in the later phases of the development process.



In FOCUS, also a transition system may be used to describe traces. It describes all traces that correspond to its executions.

Transition systems may conveniently be described by a notation in which states are seen to consist of values of state components (“variables”). Transitions are given by a logical formula. An example illustrates this technique:

Example 2.3 (The transmission system described by a state transition system):

We may specify the safety properties of our message transmission system by a transition system. (In fact, for the sake of simplicity, we give a slightly more restricted version of the safety properties: acknowledgements for send- and request-actions must be sent immediately.)

Let the state space consist of all mappings

$$\sigma : \{buf, acks\} \rightarrow Msg^* \cup Act^*,$$

such that $\sigma.buf \in Msg^* \wedge \sigma.acks \in Act^*$.

$\sigma.buf$ denotes the buffer contents in state σ , $\sigma.acks$ the list of acknowledgements that are due to be sent in state σ . The initial state is specified by the predicate

$$init.\sigma \equiv \sigma.buf = \langle \rangle \wedge \sigma.acks = \langle \rangle,$$

which expresses that the buffer is empty initially and so is the list of the acknowledgements to be sent. The transition relation is specified in the following way:

$$\begin{aligned} \sigma \xrightarrow{send, m} \sigma' &\equiv \\ \neg full(\sigma.buf) &\Rightarrow \sigma'.buf = \sigma.buf \circ m \wedge \sigma'.acks = \sigma.acks \circ ok \wedge \\ full(\sigma.buf) &\Rightarrow \sigma'.buf = \sigma.buf \wedge \sigma'.acks = \sigma.acks \circ error, \end{aligned}$$

$$\begin{aligned} \sigma \xrightarrow{req} \sigma' &\equiv \\ \neg empty(\sigma.buf) &\Rightarrow \sigma'.buf = rt(\sigma.buf) \wedge \sigma'.acks = \sigma.acks \circ trans(ft(\sigma.buf)) \wedge \\ empty(\sigma.buf) &\Rightarrow \sigma'.buf = \sigma.buf \wedge \sigma'.acks = \sigma.acks \circ fail, \end{aligned}$$

$$\begin{aligned} \sigma \xrightarrow{ft(\sigma.acks)} \sigma' &\equiv \\ \neg empty(\sigma.acks) &\Rightarrow \sigma'.buf = \sigma.buf \wedge \sigma'.acks = rt(\sigma.acks) \wedge \\ empty(\sigma.acks) &\Rightarrow false. \end{aligned}$$

This specification reveals some information about the internal structure of the transmitter, namely that it has an internal buffer for messages to be stored. The buffer may be full, which may lead to error-messages, if the sender tries to send messages in this case. This information cannot be derived from the logical specification of example 2.2. \square

2.5 Component-oriented Specification

2.5.1 Motivation

A global specification, seen as a requirements specification of a distributed system, imposes requirements on the system as a whole. However, in many cases there is already some structure information available on the requirements level. In our message transmission example, the system was seen to consist of three components. It was also obvious what actions should be considered input or output to which component.

An important point is that in a global specification often requirements on the component to be developed *and* its environment are stated. A particular goal of the component-oriented specification is to separate the requirements on the component from the assumptions about the environment. This is done by explicitly stating the responsibility of the environment in terms of a number of assumptions. A correct implementation of the component is then required to satisfy the commitments whenever the environment satisfies the assumptions. This is called the *assume/commit-style* of a component specification (see [LA90], [Pan90]).

More generally, there may not be just one component and its environment but an arbitrary number of components. The environment may also be further structured. The goal of component-oriented specification is to gradually localize global requirements.

2.5.2 Formal Treatment

Let us first look at the description of components by traces. A component is described by its input/output behaviour, which is called its *interface*. The interface is described by defining input and output actions (the set of input actions must be disjoint from the set of output actions in order to distinguish inputs from outputs in a trace) and a predicate on traces of input and output actions. These predicates are again described by trace logic and/or transition systems. Formally, a component is described by a tuple (I, O, C) with $I \cap O = \emptyset$ and

$$C : (I \cup O)^\omega \rightarrow \{true, false\}.$$

The whole system is composed of a set of components $(I_1, O_1, C_1), \dots, (I_n, O_n, C_n)$ where $O_i \cap O_j = \emptyset$ for $i \neq j$. The disjointness condition ensures that output actions can be uniquely assigned to the components. If the specification models a closed system view, (at least) one of these components represents the environment. Two components are understood to be (directly) connected, if an output action of one component is an input action of the other. The composition determines the

predicate $Sys : (I_1 \cup \dots \cup I_n \cup O_1 \cup \dots \cup O_n)^\omega \rightarrow \{true, false\}$ defined by

$$Sys(t) \Leftrightarrow \forall i \in \{1, \dots, n\} : C_i((I_i \cup O_i) \odot t).$$

Sys is a specification where all requirements are *localized*. The final component-oriented specification has to be of this form. On the other hand, the initial component-oriented specification consists of just global requirements. Between these two extremes there will be mixtures of both:



A *component-oriented specification* consists of *local requirements* on the components and of *global requirements* on the system as a whole.

Our method to derive (local) component specifications consists of the following three steps:

1. We start from a global specification. We identify a number of, say n , components and fix their inputs and outputs, $I_1, O_1, \dots, I_n, O_n$ resp., such that $I_1 \cup O_1 \cup \dots \cup I_n \cup O_n = Act$ and $O_i \cap O_j = \emptyset$ for $i \neq j$; this means that all actions of the global specification are considered and each action is uniquely assigned to one component as output action. (Actions that are not output actions of a component are seen to be output of some unspecified additional environment component. In this case the specification models an open system.)

2. We fix what (local) assumptions can be made about the environment - usually these assumptions are just some of the requirements of the global specification - and we may also assign some requirements to some system components. There may still remain some global requirements that cannot be assigned to a specific component. Accordingly a component-oriented specification has the form

$$G(t) \wedge C_1((I_1 \cup O_1) \odot t) \wedge \dots \wedge C_n((I_n \cup O_n) \odot t),$$

with a global part G and local parts C_i (some of which may be *true*, expressing that there is no local requirement for the i -th component yet).

3. We gradually localize the components' requirements. This is done in a number of *refinement steps*: new global requirements G' and local requirements C'_i are derived, such that

- $G'(t) \wedge C'_1((I_1 \cup O_1) \odot t) \wedge \dots \wedge C'_n((I_n \cup O_n) \odot t) \Rightarrow G(t) \wedge C_1((I_1 \cup O_1) \odot t) \wedge \dots \wedge C_n((I_n \cup O_n) \odot t)$
- $C'_i(t) \Rightarrow C_i(t)$, if i is the index of a component to be developed. This means some more restrictions on this component are imposed,
- $C'_i(t) \Leftrightarrow C_i(t)$, if i is the index of an environment component. This means the assumption about the environment remains the same,
- $G(t) \Rightarrow G'(t)$. The global requirements are reduced.

The requirements are completely localized when $G'(t) \Leftrightarrow true$. Then there are no more global requirements.



In order to derive interface specifications for the components:

1. Start with the global specification and identify the components plus their syntactic interface.
2. State the environment assumptions.
3. Localize the components' requirements.

The proposed requirements specification method is described in more detail in [Web92]. In particular, in this thesis the methodological use of both global and component-oriented specifications is motivated and the localization of requirements is investigated in detail. An application of the method is can be found [DW92].

Example 2.4 (Component structure of the message transmission system):

Our system is structured into the components “sender”, “transmitter” and “receiver”. The sender is responsible for *send.m* actions, the receiver is responsible for *req* and the transmitter is responsible for *trans.m* actions, and for *fail*, *ok* and *error*. This is already illustrated by the picture above.

In our simple example there are in fact no requirements on the sender and the receiver: they may issue arbitrary messages at arbitrary times. Hence, the requirements S and L both apply to the transmitter. So in fact, the requirements are already localized and the localization steps described above are not necessary here. Let

$$\begin{aligned} I &= \{send.m \mid m \in M\} \cup \{req\}, \\ O &= \{trans.m \mid m \in M\} \cup \{ok, error, fail\}. \end{aligned}$$

Then (I, O, TM) with $TM \equiv S \wedge L$ is the component-oriented trace specification of the transmitter. □

2.6 Timed Trace Specification

So far issues related to time and time sensitivity of systems have not been discussed. Of course every physical system is carried out in a particular physical time frame. For many systems, this time frame is not important for modelling their behaviour. In some application areas (for instance process control in factory automatization), however, timing is of great importance. We will now sketch how the trace formalism can be used to specify time sensitive systems.

We model time flow (discrete time) via a global clock represented by an additional action. Therefore we introduce an action \checkmark (“tick”), which occurs in a stream when no other action takes place at this point in time. We consider *timed streams* to be

elements of $(Act \cup \{\checkmark\})^\omega$ for which in addition $\#t = \infty$ holds. This requirement expresses that the ticking of the clock never stops: their either appears a “real” action or there is a tick of the clock. We denote the set of timed streams by Act_{\checkmark}^∞ , and the set of finite partial timed streams by Act_{\checkmark}^* .

To give just a flavour of this approach, we formalize the requirement “after action a occurs, at most N units of time elapse before action b occurs”:

$$P(s) \equiv \forall p \in Act_{\checkmark}^* : p \circ a \sqsubseteq s \Rightarrow \exists q \in Act_{\checkmark}^* : p \circ a \circ q \circ b \sqsubseteq s \wedge \#q \leq N.$$

The action \checkmark can be used like any other action. This simple and naive way of incorporating time nevertheless allows us to express discrete timing properties of systems conveniently in our formal framework.

2.7 Proof Principles



This section contains advanced material, which can be skipped at the first reading.

Proofs are used in system developments for two purposes. First, during a system development descriptions occur which vary in their degree of abstractness. A more abstract description may be *refined* by a more concrete one. The correctness of refinements generally is not obvious and therefore has to be proven.

Secondly, *reasoning* about a formal specification also requires proofs. As a formal specification determines a model or a model class, we are often interested whether particular properties hold for these models. These properties range from very general properties like consistency of the specification (which means that there is at least one model for the specification) to specific properties concerning the particular application. Reasoning about a trace specification is especially important in the area of a requirements specification. In particular, it is the only way to get a deeper insight into an axiomatic specification. In general, proofs may be carried out in predicate logic, but there are also more specific proof techniques.

Safety properties are usually proved by *induction*. For instance, by induction on the stream structure:

Proof Principle 2.1 (Induction on the stream structure): In order to show that a safety property S holds for a trace specification P , we show that:

$$(1) P(\langle \rangle) \Rightarrow S(\langle \rangle),$$

$$(2a) \forall s \in Act^*, a \in Act : P(a \circ s) \wedge S(s) \Rightarrow S(a \circ s) \text{ or, alternatively,}$$

$$(3b) \forall s \in Act^*, a \in Act : P(s \circ a) \wedge S(s) \Rightarrow S(s \circ a).$$

This proof technique is based on the fact that only finite traces have to be considered in the proofs of safety properties. (2a) and (2b) differ in the way a finite

trace can be built up. (2a) appends at the beginning, (2b) at the end. It depends on the particular application which rule is most convenient to use. \square

If transition systems are used in specifications, in principle, also classical predicate logic proofs may be carried out, after the transition system is translated into logical formulas. However, there are other useful proof principles, too. Safety properties expressed by transition systems may be proved by an *invariant technique*:

Proof Principle 2.2 (Proof by Invariants): In order to prove a safety property expressed by a transition system, we may give a predicate Inv (invariant) over states and finite traces such that:

- (1) $Init(s_0) \wedge Inv(s_0, \langle \rangle) \wedge (Inv(s, t) \wedge s \xrightarrow{a} s' \Rightarrow Inv(s', t \circ a))$,
- (2) $Inv(s, t) \Rightarrow S(t)$.

It is easy to see that in this case the property $S(t)$ holds for all traces t described by the transition system. \square

Proofs of *liveness properties* are in general more difficult to carry out and hard to categorize.

Example 2.5 (Proof of a liveness property of the message transmission system): We claim that the property

$$\#Send \odot t = \infty \wedge \#req \odot t = \infty \Rightarrow \#ok \odot t = \infty$$

holds for all traces t that fulfil $S(t) \wedge L(t)$. This means that, if the sender sends infinitely many messages to the transmitter and the receiver infinitely often tries to read messages from the transmitter, then infinitely many messages will be received successfully by the transmitter.

Proof: We base our proof on two lemmas. Let t be a trace such that $S(t)$ and $L(t)$ holds.

Lemma 2.1: For all $k \in \mathbb{N}$ and all $t' \in Act^*$ it holds:

$$\begin{aligned} t' \sqsubseteq t \wedge \#msg(Trans \odot t') &= k \\ \Rightarrow \#success(msg(Send \odot t'), \{ok, error\} \odot t') &\geq k. \end{aligned}$$

Lemma 2.2: For all $k \in \mathbb{N}$ and all $t' \in Act^*$ it holds:

$$\begin{aligned} t' \sqsubseteq t \wedge \#success(msg(Send \odot t'), \{ok, error\} \odot t') &= k \\ \Rightarrow \#ok \odot t' &\geq k. \end{aligned}$$

Both lemmas may easily be proven by induction on k . The first implies

$$\#msg(Trans \odot t) = \infty \Rightarrow \#success(msg(Send \odot t), \{ok, error\} \odot t) = \infty$$

and the second:

$$\#success(msg(Send \odot t), \{ok, error\} \odot t) = \infty \Rightarrow \#ok \odot t = \infty.$$

Now we can deduce:

$$\begin{aligned}
 & \#Send \circledast t = \infty \wedge \#req \circledast t = \infty \\
 \Rightarrow & \#Trans \circledast t = \infty && \text{[by } L(t)\text{]} \\
 \Rightarrow & \#msg(Trans \circledast t) = \infty && \text{[Def. of } msg\text{]} \\
 \Rightarrow & \#success(msg(Send \circledast t), \{ok, error\} \circledast t) = \infty && \text{[Lemma 2.1]} \\
 \Rightarrow & \#ok \circledast t = \infty && \text{[Lemma 2.2]}
 \end{aligned}$$

□

We do not discuss the possibility to support proofs by machine based tools, although this way a lot support can be obtained.

Chapter 3

Functional Specification

3.1 Overview

In the previous chapter the behaviour of a system was described by its set of possible action traces. This leads to a very abstract view, well-suited for requirements specification. We will now treat techniques for the design phase of the development process (see figure 1.1). For this phase we use the paradigm of communicating system *components*. Such components are connected by directed channels to form a network. Each channel links an *input port* to an *output port*.

On the trace specification level, closed and open systems were distinguished. The starting point of this chapter is the interface description of an open system (technically, a component-oriented trace specification), where the actions are already marked as input or output actions. A closed system can easily be converted into an open one by marking the system actions of interest as outputs. In order to view an open system as a component with several input and output ports, the trace specification's input and output actions have to be further partitioned into sets which correspond to the individual ports.

A functional component accepts input messages, processes them, and produces output messages. Our operational intuition is that every component has full control over the messages that appear on its output ports, but no control over the messages that arrive on its input ports. Of course, it is assumed that all messages are of the appropriate type for the channel they arrive on. A component may always send an arbitrary number of messages, which will be buffered on the connecting channels. This means that we consider *asynchronous communication*.

Communicating components still provide quite an abstract view of a system's behavior. Agents can be related to a variety of concrete computational units, such as recursive definitions in a functional programming language, procedures in an imperative programming language, processors in a multiprocessor machine, and

even digital circuits in hardware design. In FOCUS, components are modelled by sets of *stream processing functions*. These functions operate on (tuples of) input and output streams, which correspond to the respective communication channels. Stream processing functions may also be parameterized by (auxiliary) non-stream arguments. Remember that traces and streams are technically the same thing, namely finite or infinite sequences. However, in the context of our method, we use a trace to model a run of the whole system or a system component, and a stream to model the communication history of just a single channel.

If the components described at the trace specification level are not too complex, then it may be acceptable to replace every component-oriented trace specification by one monolithic functional specification. Sometimes, however, it is more convenient to specify a component not as a single processing agent, but as a network composed of sub-component. The specification and development techniques which are presented in this chapter can be applied repeatedly to refine components that way. The sub-components in a network are connected by *internal channels*, and communicate via *internal messages*, which are hidden from the environment. At this level of the development process, the structure of an network should only depend on the physical and/or logical structure of the system to be specified. Implementation considerations should not be taken into account yet.



The design steps described in this chapter can be summarized as follows:

1. Start with a component-oriented trace specification of a single component.
2. Partition the sets of input and output actions given by the trace specification into subsets corresponding to the component's input and output channels.
3. Specify the component in terms of (a network of) stream processing functions.
4. Prove that this specification refines the initial trace specification.
5. If the specification is too abstract to be implemented, then give a more constructive specification and prove that it refines the abstract one. Repeat this step if necessary.

The first two steps deal with the classification of actions and their association with certain channels. These points will be described in section 3.2. Steps 3 to 5 refer to functional component specifications. In section 3.3 we will have a closer look at stream processing functions, which model the behaviour of components. Then, in section 3.4, techniques for the functional specification of components will be explained. The important special case of state-oriented specifications will be handled in section 3.5. In 3.6 we will present techniques for the description of component networks. In section 3.7 we address refinements, and finally timing aspects (section 3.8), and the advanced topic of proof principles (section 3.9).

3.2 Actions, Channels and Messages

In this section we give some additional information on the first two design steps described above. Assume we are given a component-oriented trace specification (see section 2.5), which consists of a triple (I, O, C) , where I and O are disjoint sets of input and output actions, respectively, and C is a predicate characterizing a set of traces over these actions. The set I may be empty, but O must not. In the following, we write A for $I \cup O$.

A component is connected to its environment by a fixed number of input and output ports (corresponding to the channels). There must be at least one output port, and, if the action set I is not empty, then there must also be at least one input port. The input actions in I are associated with the input ports and the output actions in O are associated with the output ports. Every action must be associated with exactly one port. Formally, this means that the set I is partitioned into $p \geq 0$ sets I_1, \dots, I_p , and the set O is partitioned into $q > 0$ sets O_1, \dots, O_q . The component is represented by functions of the functionality

$$I_1^\omega \times \dots \times I_p^\omega \rightarrow O_1^\omega \times \dots \times O_q^\omega.$$

This functionality fixes the *syntactic interface* of a system component, stating the number of input ports, the number of output ports and which sorts of messages can be sent over the individual ports.

While the sets of input and output actions are already fixed on the trace specification level, it is an additional design decision to determine the “appropriate” number of input and output ports and the actions that are associated with the individual ports. This decision is based on the designer’s application dependent understanding of the overall system structure. The designer must also take into account that putting two actions onto different channels hides the information about the relative ordering of these two actions (at least if no timing information is available, see the discussion of the non-strict fair merge in section 3.4).

The association of actions to ports already carries some information, which can be used to characterise the actions uniquely in an abbreviated notation. We call this the transition from *actions* to *messages*. For example, consider the message transmission system introduced in the previous chapter. There, we had actions of the form $send.m$ and $trans.m$ for every $m \in M$. If we decide that send and transmit actions should be associated with different channels, then it is obviously sufficient just to refer to the message m appearing on one or the other channel. Knowing on which channel a message m appeared is sufficient to determine whether it stands for the action $send.m$ or $trans.m$.

Formally, this means that we replace an action set I_k by some message set I'_k , provided that there exists a total bijection between I_k and I'_k . The same can be done for each set of output actions O_k . The elements of the sets I'_k and O'_k are

called *messages*. Since this way actions and messages can be converted into each other, we will be very liberal in our terminology and notation. We always use the concept which is most convenient. The reader can imagine that messages are just a notational shorthand for actions.

For the rest of this chapter, we write *Instreams* for $I_1^\omega \times \dots \times I_p^\omega$, and *Outstreams* for $O_1^\omega \times \dots \times O_q^\omega$.

The next step in the development, which will be described in detail in the following sections, is to specify a component as (a network of) stream processing functions of the overall functionality $comp : Instreams \rightarrow Outstreams$. For the internal channels of such a network, actions and messages from A and newly defined internal actions and messages may be used.

Example 3.1 (Messages of the message transmission system):

The actions in our message transmission system example were defined by (see example 2.1):

$$Act = \{send.m \mid m \in M\} \cup \{trans.m \mid m \in M\} \cup \{ok, error, fail, req\}.$$

From example 2.4 we already know that $I = \{send.m \mid m \in M\} \cup \{req\}$ are the transmitters input actions and $O = \{trans.m \mid m \in M\} \cup \{ok, error, fail\}$ are the transmitters output actions.

Furthermore, from the picture in example 2.1 we see that the output actions *ok* and *error* should go to the sender, and *trans.m* and *fail* should go to the receiver; so it is appropriate to model the transmitter as a component with two output channels. The input is not split into two channels because the relative order of the incoming messages is important. The component specified here expects that the messages coming from the sender and receiver are merged by the environment. The reason for this will be explained in section 3.4. We also perform a transition from actions to messages as described above and obtain the following sets of messages:

$$\begin{aligned} In &= M \cup \{req\}, \\ Out &= M \cup \{fail\}, \\ Ack &= \{ok, error\}. \end{aligned}$$

Hence the transmitter to be specified has the functionality:

$$trans : In^\omega \rightarrow Ack^\omega \times Out^\omega$$

Performing several design steps we are going to develop a procedural program that realizes the transmitter. \square

3.3 Stream Processing Functions



A *stream processing function* is a (prefix continuous) function that operates on (tuples of) streams and produces (tuples of) streams as results.

A deterministic component can be represented to a single stream processing function. For the moment, we restrict our attention to stream processing functions. The way components (including nondeterministic ones) are specified is discussed in the next section.

Our operational understanding that stream processing functions model interacting components leads to some basic requirements for them. In particular, an interactive component is not capable to undo (take back) an output message that has already been emitted. This observation is mirrored by the requirement of *monotonicity*. It also reflects our notion of causality: if some sequence of input messages already causes some output messages to occur, then a longer input must be causal for at least the same, or more, output.



Formally, this requirement is expressed as follows:

Definition 3.1 (Monotonicity): Let (X, \sqsubseteq) and (Y, \sqsubseteq) be cpos (complete partial orders). A function $f : X \rightarrow Y$ is monotonic, if for all $x, x' \in X$ we have that $x \sqsubseteq x' \Rightarrow f(x) \sqsubseteq f(x')$. \square



Besides the operational justification of monotonicity, there is also a theoretical one. From theorems by Knaster and Tarski it is well known that monotonic functions over cpos have a least fixed point. Recall that the set of streams M^ω together with the (partial) prefix order forms a cpo (see section 2.2). Since (the streams generated in) feedback loops will be modelled by least fixed points, the use of monotonic stream processing functions is a necessity if we want to be able to give a semantics for functional networks with feedback loops within the framework of domain theory.

The second basic requirement for stream processing functions is that of *continuity*: a function's behaviour should be fully described by its behaviour for finite inputs. Given some (possibly infinite) input, the function's output for all finite prefixes of the input must successively approximate the function's total output. For example, this means that it is not possible to specify an agent that produces some output only as a reaction to infinite input.



Formally, the requirement of continuity is expressed as follows, where $\bigsqcup X$ stands for the least upper bound of a chain (totally ordered set) X :

Definition 3.2 (Continuity): A monotonic function $f : X \rightarrow Y$ is continuous, if for all chains $X' \subseteq X$ the following equation holds $\bigsqcup\{f(x) \mid x \in X'\} = f(\bigsqcup X')$. \square



Note that continuity implies monotonicity, but not vice versa. As it was the case with monotonicity, continuity also has important theoretical aspects:

- Consider an input stream $x = \sqcup\{x_i \mid i \in \mathbb{N}\}$, where the x_i 's form a chain. Then the output of a continuous function can be determined either by $f(x)$ or equivalently by $\sqcup\{f(x_i) \mid i \in \mathbb{N}\}$. This demonstrates that the behaviour of a continuous function is uniquely defined by its behaviour for finite inputs.
- It has been shown by Kleene that $fix.f = \sqcup_{n \in \mathbb{N}} f^n(\perp)$ for continuous functions f , where $f^0(x) = x$ and $f^{n+1}(x) = f(f^n(x))$ and $fix.f$ denotes the *least fixpoint* of f , i.e. the least element that fulfils the equation $f(x) = x$. Building this chain of repeated function applications models the stepwise computation process that takes place when feedback loops are considered (see [Kah74]).



Continuous functions are “well-behaved” functions in the sense that they properly model a stepwise computation process. In FOCUS, only continuous functions will be considered as descriptions of components.

In spite of the continuity requirement for individual stream processing functions, fairness and even more general liveness properties can be expressed in FOCUS. Fairness is not modelled as a property of a single function, but as a property of a component, which corresponds to a set of functions.

While the behaviour of a component is represented by continuous stream processing functions, non-continuous auxiliary functions may also be used to support the specification. An example is the concatenation function \circ , which is non-monotonic (and therefore not continuous) in its second argument.

3.4 Specification of Components



A component is modelled by a (non-empty) set of continuous stream processing functions. Such a set is represented by a predicate on functions. Each function from this set corresponds to one particular (deterministic) behaviour.

Hence, a functional component specification is given by a predicate:

$$P : (Instreams \rightarrow Outstreams) \rightarrow \mathbb{B},$$

that describes the following set of (continuous) stream processing functions:

$$\{ f : Instreams \rightarrow Outstreams \mid P(f) \wedge f \text{ is continuous} \}.$$

The functionality $Instreams \rightarrow Outstreams$ corresponds to the component's input and output channels. Every function describes a potential input/output behaviour of the component stating how it reacts to all possible inputs. Obviously, if the above set contains only one element, it stands for a deterministic component, since in this case for any input the output is uniquely prescribed. In general, however, there will be more than one continuous function fulfilling a predicate P . In this

sense, a component specification is a description of the *properties* the component is required to have, expressed by a formula of predicate logic.

Example 3.2 (A summation component):

We specify a component operating on streams of natural numbers which, for every input, gives an output which is not less than the sum of the inputs received so far. Let x_i denote the i -th element of a stream x for $1 \leq i \leq \#x$. The predicate Sum , which is a straightforward translation of the informal requirements, gives a specification for such a component. Note that it is sufficient to describe the component's behaviour for finite input streams because of the implicit requirement of continuity.

$$\begin{aligned} Sum_1 &: (\mathbb{N}^\omega \rightarrow \mathbb{N}^\omega) \rightarrow \mathbb{B}, \\ Sum_1(f) &\equiv \forall x \in \mathbb{N}^\omega, n \in \mathbb{N} : \#x = \#f(x) \wedge \\ &\quad (1 \leq n \leq \#x < \infty \Rightarrow f(x)_n \geq \sum_{1 \leq i \leq n} x_i). \end{aligned}$$

Alternatively, the component Sum_1 can also be specified by stating that the input and output streams have the same length and, for a finite input stream, only the *last* output is greater than or equal to the sum of the input elements received so far. Together with the implicit requirements of monotonicity and continuity this alternative specification is equivalent to the specification given above. Generally, it is a question of style to decide whether a more explicit specification or an implicit one is preferable. \square

Example 3.3 (A fair merge component):

An example of an often-needed component is one that merges two input streams into a single output stream. As an additional constraint, we require the component to be *fair* in the sense that it does not neglect

one input infinitely long. Suppose that I_1 and I_2 are disjoint. The specification of the component *Fair-Merge* reads as follows:

$$\begin{aligned} Fair-Merge &: (I_1^\omega \times I_2^\omega \rightarrow (I_1 \cup I_2)^\omega) \rightarrow \mathbb{B}, \\ Fair-Merge(f) &\equiv \forall x \in I_1^\omega, y \in I_2^\omega : \\ &\quad \#x = \#y = \infty \Rightarrow x = I_1 \odot f(x, y) \wedge y = I_2 \odot f(x, y). \end{aligned}$$

This is an example of a very implicit specification, which explicitly restricts the component's behaviour for infinite input streams only. For finite inputs, certain restrictions for its behaviour follow from monotonicity and continuity. For example, it is easy to show that no output can ever appear which has not already been received as input. The assumption that this could be the case leads to a contradiction if the inputs are extended to infinite streams. \square

Example 3.4 (Functional specification of the message transmission system):

We now specify the transmitter component of the message transmission system by a predicate $Trans_1$. In this predicate, the safety requirements exactly correspond to those of the original trace description (see section 2.4.1, example 2.2). We could do

the same with the liveness requirements, thus creating a certainly correct but not very constructive specification. However, as a first step towards an implementable component, we replace the liveness requirements, stating that acknowledgements may be delayed an arbitrary but finite amount of time, by certain fixed, finite bounds. Moreover, we now explicitly introduce a finite bound on the transmitters capacity to store messages. The first bound (mds) represents the maximum delay between the sending of a message by the sender and the reply (ok or $error$) of the transmitter. The second bound (mdr) represents the maximum delay between a request (req) issued by the receiver and the reply (a message m or $fail$) of the transmitter and the third bound ($mstor$) represents the transmitter's capacity to store messages. For example, a value of $mstor = 1$ means that the transmitter can store at most one message at any given time.

$$\begin{aligned}
Trans_1 &: (In^\omega \rightarrow Ack^\omega \times Out^\omega) \rightarrow \mathbb{B} \\
Trans_1(f) &\equiv \exists mds, mdr \in \mathbb{N}, mstor \in \mathbb{N} \setminus \{0\} : \\
&\quad \forall x \in In^* : \exists y \in Ack^\omega, z \in Out^\omega : f(x) = (y, z) \wedge \\
&\quad \#y \leq \#M \odot x \leq mds + \#y \wedge \\
&\quad \#z \leq \#req \odot x \leq mdr + \#z \wedge \\
&\quad M \odot z \sqsubseteq success(M \odot x, y) \wedge \\
&\quad \#success(M \odot x, y) - \#M \odot z > 0 \Rightarrow lt(z) \neq fail \wedge \\
&\quad \#success(M \odot x, y) - \#M \odot z < mstor \Rightarrow lt(y) \neq error.
\end{aligned}$$

Here lt is the function dual to ft ; it gives the last element of a finite, non-empty stream. Remember: $In = M \cup \{req\}$, $Out = M \cup \{fail\}$ and $Ack = \{ok, error\}$. The transmitter's behaviour is fully described by an explicit specification of its behaviour for finite input streams and by our implicit requirement of continuity. This implies that, because of the design decision to introduce finite bounds, we have managed to transform liveness into safety properties. We only need the additional liveness assumption concerning the length of the output streams. \square

We have already seen that a component specification need not describe exactly one stream processing function. In fact, all specifying predicates of the examples above are fulfilled by many functions. This is called *underspecification*. It can be used to represent *nondeterminism*. Given a nondeterministic component, there are no means for its environment of controlling which of the allowed behaviours will occur in an actual run of its implementation. In fact, we cannot even control *when* the nondeterministic choices are made: they can be made during the design process, since a specification may always be strengthened, for instance, to add error handling capabilities or to implement the component in a deterministic programming language. The choice can also be made during the run of the implementation, where different choices are possible for each individual run, and for multiple instances of the same agent.



It should be noted that the seemingly straightforward relational approach to nondeterminism, where a nondeterministic agent corresponds to a *relation* on streams, is

not sufficient, since it is not compositional. A relation in general does not convey enough information about an component's behaviour. This has, for instance, been demonstrated by Brock and Ackermann in [BA81].

The requirement of continuity is implicit in all component specifications, even if it is not explicitly stated each time. A component specification P is called *consistent* if it does not describe the empty set, i.e. if it exists at least one *continuous* function f such that $P(f)$ holds. Usually we are only interested in consistent specifications. Sometimes, however, it is hard to tell, whether a given specification is consistent.

Example 3.5 (A non-strict fair merge component):

The agent *Fair-Merge* as defined above does not guarantee that, given two finite input streams, every element of the inputs re-appears in the output stream. It is easy to write down a specification which seems to solve this problem:

$$\begin{aligned} \text{Non-Strict-Fair-Merge}(f) &\equiv \text{Fair-Merge}(f) \wedge \\ &\quad \forall x \in I_1^\omega, y \in I_2^\omega : \#f(x, y) = \#x + \#y. \end{aligned}$$

Unfortunately, this specification is not fulfilled by any monotonic function: suppose we had a monotonic function f which performed a non-strict fair merge. Without loss of generality we assume that $f(\langle a \rangle, \langle b \rangle) = \langle a, b \rangle$. Because of monotonicity, $f(\langle \rangle, \langle b \rangle)$ cannot be $\langle b \rangle$, but any other result would violate the predicate given above. Thus *Non-Strict-Fair-Merge* is an inconsistent specification in our sense. \square

It is indeed not possible to specify such a non-strict fair merge in a straightforward manner by predicates on stream processing functions. This is a general problem with stream-oriented functional methods. The more elaborate formalism of *input choice specifications*, which is presented in [Bro90], is able to handle this case. Another, less sophisticated, solution (the inclusion of dummy messages similar to time ticks indicating the lack of input is presented in chapter 3.8.

3.5 State-oriented Functional Specification

In the section on trace specifications we distinguished between the *history-oriented* and the *transition-oriented* specification style. In a similar way, we may use and combine these specification styles also in functional specifications. We have already presented the general framework of stream-processing functions and methods for specifying components using predicates on functions. In some of the examples given in section 3.4 a component was specified by relating *complete* input and output streams (for instance the summation component in 3.2 and the fair-merge component in example 3.3); this corresponds to the history-oriented specification style. On the other hand, a *transition-* or *state-oriented* functional specification style is also possible. We shall now have a closer look on the methodological use of such a specification style.

There are two possible cases when a state-oriented specification is useful. Firstly, it may simply be difficult to specify a component by a history-oriented specification. Secondly, a state-oriented specification can already look very much like an abstract program. Therefore the transformation of a history-oriented specification into a state-oriented one is often an important design step. Of course, it must then be shown that the state-oriented specification implies and thus refines the history-oriented one.

A stream-processing function can — because of its monotonicity and continuity — be considered to be an abstract model of an agent which receives some input and, depending on its internal state, produces some output and changes its state. The functional view is an abstraction of this operational model. For certain applications, however, it is convenient to introduce an explicit notion of state and use it for a transition-oriented specification. This specification method will be explained in the following.

Let us first consider deterministic components only, i.e. components that may be described by a single function. For the moment let us assume for simplicity that specify a function $f : I^\omega \rightarrow O^\omega$ with a single input stream and a single output stream (this may easily be generalized to functions with more than one input or output stream). To specify f , we introduce a set *State* and describe f using an auxiliary function

$$h : State \rightarrow (I^\omega \rightarrow O^\omega),$$

with a state parameter such that

$$f = h(\mathit{init}),$$

where *init* is some particular initial state. Then, we specify h by giving formulas of the form:

- (1) $P(s) \Rightarrow h(s)(\langle \rangle) = o$,
- (2) $Q(i, s) \Rightarrow h(s)(i \& x) = o' \circ h(s')(x)$,

where $i \in I$ is a single input, $o \in O^\omega$ is a (possibly infinite) output, $o' \in O^*$ is a finite output, $x \in I^\omega$ is an input stream, $s \in State$ is the “old” state, $s' \in State$ is the “new” state, and P and Q are predicates (which can of course be omitted if they are identical to *true*). The stream o (usually) depends on s , and o' and s' (usually) depend on i , x and s .

(1) is the “termination case”: if the input stream is empty and s fulfils a certain condition P , then o is output — and nothing more happens.

(2) expresses the transition behaviour: if an input i and a state s fulfil a certain condition Q , then h reads i in state s , outputs o' and enters state s' . Obviously this “rule” may only be applied if the input stream is not empty.



In fact, a stream-processing function can be represented by a (not necessarily finite)

automaton with input and output. This formal relationship has been investigated in [LS89].

By h a deterministic component is specified: we have described exactly one function. The state-oriented specification of nondeterministic components is treated below.

Example 3.6 (State-oriented specification of the message transmission system):

In order to achieve a simpler specification, we now make the design decision to set the maximum delay between send and request messages and the corresponding acknowledgements introduced in example 3.4 to zero: $mds, mdr = 0$. This implies that our new agent must give a reply as soon as new input arrives. However, we still have the freedom to choose an arbitrary storage capacity, $mstor$, for a subsequent implementation.

Let M^* be the state space: any sequence $buf \in M^*$ represents the messages the transmitter has currently stored. Initially no message is stored, thus $\langle \rangle$ is the initial state. Now we give a state-oriented specification of the transmitter component as follows:

$$\begin{aligned}
Trans_2(f) &\equiv \exists h : M^* \rightarrow (In^\omega \rightarrow Ack^\omega \times Out^\omega) : \\
&\quad \forall x \in In^\omega : f(x) = h(\langle \rangle, x) \wedge H(h), \\
H(h) &\equiv \exists mstor \in \mathbb{N} \setminus \{0\} : \forall x \in In^\omega, m \in M : \\
&\quad h(buf)(\langle \rangle) = \langle \rangle \wedge \\
&\quad \#buf < mstor \Rightarrow h(buf)(m \& x) = (ok, \langle \rangle) \circ h(buf \circ m)(x) \wedge \\
&\quad \#buf = mstor \Rightarrow h(buf)(m \& x) = (error, \langle \rangle) \circ h(buf)(x) \wedge \\
&\quad \#buf > 0 \Rightarrow h(buf)(req \& x) = (\langle \rangle, ft(buf)) \circ h(rt(buf))(x) \wedge \\
&\quad \#buf = 0 \Rightarrow h(buf)(req \& x) = (\langle \rangle, fail) \circ h(buf)(x).
\end{aligned}$$

Here we use a generalization of the concatenation operation \circ to concatenate pairs of streams. \square

The state-oriented specification of a nondeterministic component is a slight generalization of the procedure described above: we specify a *set* of initial states by giving a predicate $Init(s)$ on states, moreover, we specify the transition relation nondeterministically in the following way:

$$\begin{aligned}
(1') \quad P(s) &\Rightarrow \exists o : R(o, s) \wedge h(s)(\langle \rangle) = o, \\
(2') \quad Q(i, s) &\Rightarrow \exists s', o' : S(o', s', i, s) \wedge h(s)(i \& x) = o' \circ h(s')(x).
\end{aligned}$$

Here P, Q, i, o, o', x, s, s' are as above and R and S are predicates. $R(o, s)$ expresses the range of possible outputs o in a state s , $S(o', s', i, s)$ expresses the range of possible outputs o' and next states s' for a given input i and a state s .

Such a specification can as before be seen as the description of a *predicate* H on functions: $H.h \equiv S_1 \wedge \dots \wedge S_n$, where S_1, \dots, S_n are formulas as in (1') and (2') above. The predicate H describes all functions with the specified behaviour.

Example 3.7 (A state-oriented version of the summation component):

The component Sum_1 introduced in example 3.2 will now be specified in a state-oriented way. The current state is just the sum of input elements received so far.

$$\begin{aligned} Sum_2(f) &\equiv \exists h \in \mathbb{N} \rightarrow (\mathbb{N}^\omega \rightarrow \mathbb{N}^\omega) : \forall x \in \mathbb{N}^\omega : f(x) = h(0)(x) \wedge H(h), \\ H(h) &\equiv \forall n, m \in \mathbb{N}, x \in \mathbb{N}^\omega : h(n)(\langle \rangle) = \langle \rangle \wedge \\ &\quad \exists k \in \mathbb{N} : h(n)(m \& x) = (n + m + k) \circ h(n + m)(x). \quad \square \end{aligned}$$



The complete procedure is as follows:

1. Embed the function to be specified in an auxiliary function with an additional state parameter.
2. Choose an appropriate state space for the auxiliary function.
3. Specify the initial states.
4. Specify the transition behaviour with several rules of the form (1) and (2) or their generalizations (1') and (2').

There are alternative ways of writing state-oriented specifications. For example, one can use recursively defined predicates. See [Den91] for an example where a relatively large system has been specified using this approach. There exist special notational conventions to denote state transitions, such as for instance the notation presented in [Lam83]. This makes use of a state oriented specification concept possible for complex systems.

3.6 Specification of Networks



The definition of networks is the main structuring tool on the functional specification level. There is no (semantical) difference in principle between a single component and a network of components. A network can be defined either by recursive equations or by special composition operators.

In FOCUS, networks of components can be represented by directed graphs, where the nodes represent components and the edges represent point-to-point, directed communication channels. It is a fundamental fact known as the *Kahn principle* (see [Kah74], [KM77]) that such networks of components can (semantically) be seen as components again. Hence, we are allowed to build an component from a collection of simpler components.

3.6.1 Equational Definitions

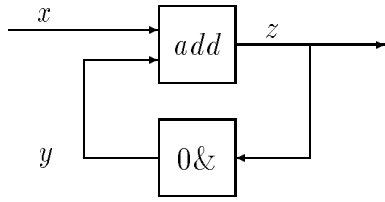


In an equational definition, a network is defined by a set of mutually recursive stream equations.

Recall the summation component Sum_2 from example 3.7. For the moment, we restrict ourselves to a deterministic version which always outputs the *exact* sum of the inputs obtained so far (choose $k = 0$ in the specification of Sum_2). Now, suppose that we do not want to build it from scratch, but instead employ an already implemented component add that adds two input streams element by element. Then, the specification can be rewritten as follows:

$$Sum_3(f) \equiv \forall x \in \mathbb{N}^\omega : \exists y, z \in \mathbb{N}^\omega : f(x) = z \wedge z = add(x, y) \wedge y = 0 \& z.$$


Thus, a function f that satisfies the predicate Sum_3 corresponds to the following network:



$$f(x) = z, \text{ where } \begin{aligned} z &= add(x, y), \\ y &= 0 \& z. \end{aligned}$$

or equivalently:

$$f(x) = z, \text{ where } z = add(x, 0 \& z).$$

 The semantics of an equationally defined network is the *least* function (with respect to the prefix order) that fulfils the defining equations for all input values. This interpretation is consistent with an operational view of component networks connected by asynchronous communication channels.

3.6.2 Compositional Forms

The introduction of channel names (which is necessary for equational specifications) is often helpful, but in some cases it may clutter a specification. Depending on the regularity of the network structure, the use of channel names may or may not be advisable. Moreover, different proof methods are used for the parallel and sequential composition of agents (here, equational reasoning is sufficient), and for feedback loops (where induction arguments are required). It may therefore be appropriate to build a network of functional components using specific composition operators. Below we call a stream processing function with n input ports and m output ports an (n, m) -ary function.

Definition 3.3 (Sequential (functional) composition): Let f be an (n, m) -ary function, and let g be an (m, o) -ary function. Then $f \circ g$ is the (n, o) -ary function defined by:

$$(f \circ g)(x) = g(f(x)). \quad \square$$

Definition 3.4 (Parallel composition): Let f be an (n, m) -ary function, and let g be an (o, p) -ary function. Then $f \parallel g$ is the $(n + o, m + p)$ -ary function defined by:

$$(f \parallel g)(x_1, \dots, x_{n+o}) = (f(x_1, \dots, x_n), g(x_{n+1}, \dots, x_{n+o})) \quad \square$$

Definition 3.5 (Feedback): Let f be an (n, m) -ary function, where $n > 0$. Then μf is the $(n-1, m)$ -ary function such that the value of $(\mu f)(x_1, \dots, x_{n-1})$ is the least solution (with respect to the prefix order \sqsubseteq) of the equation:

$$(y_1, \dots, y_m) = f(x_1, \dots, x_{n-1}, y_m). \quad \square$$

The μ -operator defined above always feeds back the m th output channel of a (n, m) -ary component. A generalized version of this operator is also possible.

Alternatively, the μ -operator is often defined as follows, where $fix.g$ denotes the least fixed point of a function g (see section 3.3):

$$(\mu f)(x_1, \dots, x_{n-1}) = fix.g, \quad \text{where } g(y_1, \dots, y_m) = f(x_1, \dots, x_{n-1}, y_m) \quad \square$$

The example specification Sum_3 can now be redefined as follows, where id denotes the identity function:

$$Sum_4(f) \equiv f = \mu((id \parallel 0\&) \circ add).$$

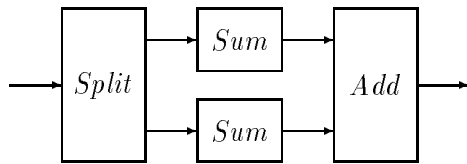
It corresponds to same net as depicted above. Every network given in terms of the introduced compositional forms can easily be transformed into an equational form.

The compositional forms defined for stream processing functions can also be lifted to predicates on such functions in a straightforward way. This sometimes shortens the notation since one need not write quantifiers and sorts etc..

Definition 3.6 (Compositional forms for specifications): Let P and Q be predicates describing functional components. Then we define:

$$\begin{aligned} (P \circ Q)(f) &\equiv \exists g, h : P(g) \wedge Q.h \wedge f = g \circ h, \\ (P \parallel Q)(f) &\equiv \exists g, h : P(g) \wedge Q.h \wedge f = g \parallel h, \\ (\mu P)(f) &\equiv \exists g : P(g) \wedge f = \mu g. \end{aligned} \quad \square$$

As an example, consider the following network $Double$, whose output for every input element is not less than twice the sum of the inputs received so far:



$$\begin{aligned} Double &= Split \circ (Sum \parallel Sum) \circ Add, \\ \text{where } Split(f) &\equiv f(x) = (x, x), \\ Add(f) &\equiv f = add. \end{aligned}$$

Functional calculus provides a basis for reasoning about networks.

3.7 Refinement

As stated in section 1.3 *refinement* is the basic notion of development in FOCUS. (see also [Bro92b] and [Bro92a])



In general, a refinement relates two different system representations that (possibly) belong to different levels of abstraction.

Refinement steps are taken when going from one development phase to the next, for instance from the requirements specification phase to the design phase, but also within these phases.

In chapter 2 we have already shown how global trace specifications are gradually refined into component-oriented trace specifications. In design phase components are no longer represented by traces but by (sets of) stream processing functions. Hence, we technically have to refine a component oriented trace specification (I, O, C) into a functional component specification

$$P : (I_1^\omega \times \dots \times I_p^\omega \rightarrow O_1^\omega \times \dots \times O_q^\omega) \rightarrow \mathbb{B},$$

where $I = \bigcup I_j$ and $O = \bigcup O_j$ (see section 3.2 for an explanation why I and O are partitioned into subsets).

In order to prove the correctness of such a refinement step, it is necessary to relate both kinds of specifications formally. This is done by defining the set of traces that is generated by a stream processing function. The following definition states that all outputs of a stream processing functions eventually appear in the trace (first conjunct of the definition), but that they may be delayed arbitrarily long (second conjunct). This corresponds to the behaviour of an asynchronous communication channel.

Definition 3.7 (Traces corresponding to a function): Given a stream processing function $f : \text{Instreams} \rightarrow \text{Outstreams}$. The *traces* of f are defined by:

$$\begin{aligned} \text{Traces}.f = \{ & t \in (I \cup O)^\omega \mid (O_1 \odot t, \dots, O_q \odot t) = f(I_1 \odot t, \dots, I_p \odot t) \wedge \\ & \forall t' \sqsubseteq t : (O_1 \odot t', \dots, O_q \odot t') \sqsubseteq f(I_1 \odot t', \dots, I_p \odot t') \}, \end{aligned}$$

where, as usual, *Instreams* stands for $I_1^\omega \times \dots \times I_p^\omega$ and *Outstreams* stands for $O_1^\omega \times \dots \times O_q^\omega$. □

Now the trace set corresponding to a functional component specification can be defined straightforwardly:

Definition 3.8 (Traces corresponding to a component specification): Let P be as above. The traces corresponding to P are defined by

$$\text{Traces}.P = \bigcup \{ \text{Traces}.f \mid P(f) \wedge f \text{ is continuous} \}.$$

Thus, $Traces.P$ is the union of all trace sets that correspond to the continuous functions which fulfill the predicate P . \square

A component specification is said to *refine* a trace specification (I, O, C) if all traces admitted by P are also admitted by C . Formally: P refines (I, O, C) iff

$$Traces.P \subseteq \{t \in (I \cup O)^\omega \mid C(t)\}.$$

Remember that we only considered total traces, which correspond to complete computations. Hence all liveness properties are taken into account. In general P only generates a subset of the traces described by the trace predicate C .

In fact, if P is *inconsistent*, i.e. if there is no continuous function fulfilling P , then $Traces.P$ is the empty set, which refines any trace specification. Of course we are only interested in consistent specifications.

The obligation to prove that P refines the initial trace specification is the connecting link between the trace and functional specification stages. See section 3.9 for some advice on how such a proof can be done. For some cases, there exist syntactic transformation between trace specifications and corresponding function predicates. Examples for such an easy transition between these two formalisms are given in [DW92].

Once a component is represented functionally there are several ways how it can be refined. As explained in section 3.4 a functional component specification describes the *properties* the component is required to have. It can be refined by adding further requirements. Technically this means that the specifying predicate is strengthened. In our logical framework refinement of component specifications is therefore expressed by an implication. This is called *behaviour refinement*. Let

$$P, Q : (Instreams \rightarrow Outstreams) \rightarrow \mathbb{B}$$

be two component specifications. Then Q is a *behaviour refinement* of P iff

$$Q \Rightarrow P$$

holds. Any behaviour admitted by Q is also admitted by P but Q may be more restrictive. It is important to notice that this refinement notion is consistent with the one introduced to connect the trace and the functional layer, since apparently $Q \Rightarrow P$ implies $Traces.Q \subseteq Traces.P$. Thus, if P refines a given trace specification so does Q .

Example 3.8 (Refinement of the message transmission system):

In the previous sections several versions of the transmitter component of the message transmission system have been developed. We gave these examples in order to study different specification styles but also to demonstrate how a systematic development is performed in FOCUS. This is formally reflected by the fact that the initial trace specification of the transmitter (I, O, TM) (see example 2.4) is

refined by the first functional component specification $Trans_1$ (see example 3.4). It holds:

$$Traces.Trans_1 \subseteq \{t \in (I \cup O)^\omega \mid TM(t)\}.$$

Moreover the first transmitter specification is (behaviourally) refined by the second (see example 3.6), since the following formula is valid:

$$Trans_2 \Rightarrow Trans_1.$$

These claims can be proved using the proof principles of section 3.9 below. \square

Behaviour refinement is the most basic refinement notion for component specifications. It only considers a component's input-output behaviour not taking into account changes of the internal structure and the way the specification is represented. For instance, it might be a considerable step towards an implementation if a specification P is replaced by a state-based specification Q , although P and Q are equivalent with respect to their external behaviour, i.e. $P \Leftrightarrow Q$. This demonstrates that a refinement step may bring a specification in a more concrete form although the specified external behaviour is not changed. We call this *refinement by reformulation*. Semantically refinement by reformulation is just a special case of behaviour refinement (and so are all other refinement notions introduced further on).

Example 3.9 (Refinement by reformulation of the summation component):

The summation component Sum_1 specified in example 3.2 is refined with respect to its representation by Sum_2 specified in example 3.7. The external behaviour is not changed:

$$Sum_1 \Leftrightarrow Sum_2.$$

In Sum_2 a state is introduced. This specification is already executable. \square

A way of refinement typical for distributed systems consists of a change of their *conceptual* or *spatial distribution*. Often a first specification describes a system as monolithic black box, only considering its external interface, as we do it in the global trace specification (see section 2.4). During the development process it is then split up into a network of interconnected components. We regard it as one of the main features of FOCUS that one can do such a decomposition at all development phases. In particular, in the design phase one can refine a given component specification P by a specification Q that describes a network of (more basic) components. This is called *distribution refinement*. Of course, these more basic components can be further refined, for instance, by reformulation or again by distribution refinement.

Example 3.10 (Distribution refinement of the summation component):

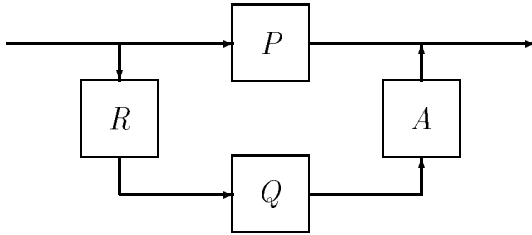
The specification Sum_3 introduced in section 3.6.1 represents the summation com-

ponent by a (cyclic) network. It is a distribution refinement of Sum_2 and also a (real) behaviour refinement, since $Sum_3 \Rightarrow Sum_2$. \square

All the refinement notions introduced above have in common that they restrict the behaviour or change the structure of a component but not its *syntactic interface*. In the rest of this section we now study a form of refinement that also allows to change the syntactic interface consisting of the number and type of the input and output channels. Hence, this form of refinement is called *interface refinement*. Technically, the interface of component corresponds to the functionality of the specifying predicate: a predicate

$$P : (I_1^\omega \times \dots \times I_p^\omega \rightarrow O_1^\omega \times \dots \times O_q^\omega) \rightarrow \mathbb{B}$$

specifies a component with p input and q output channels. When the interface of Q is refined, one may change the numbers of input and output channels as well as the type of these channels, i.e. the granularity of the communicated message. Consider an agent P that performs some manipulation on an input stream. If we want to design an agent Q that fulfills the same task in an environment, that issues messages of a more “concrete” datatype, we need to specify a transition A from the concrete to the abstract level and a transition R from the abstract to the concrete level. The agent Q is then an *interface refinement* of P if $R \circ Q \circ A$ behaviourally refines P . Here \circ is one of the composition operators for specifications introduced in section 3.6.2. The following figure shows the situation:



Definition 3.9 (Abstraction and representation): Consider a tuple of streams of type M on the abstract level and a tuple of streams of type M' on the concrete level. Two component specifications A and R are called *abstraction* and *representation*, respectively, iff the following conditions hold:

- A has the functionality $(M' \rightarrow M) \rightarrow \mathbb{B}$
- R has the functionality $(M \rightarrow M') \rightarrow \mathbb{B}$
- $R \circ A = Id$, where Id is the predicate that is only fulfilled by the identity function \square

Based these notions we formally call a component Q an *interface refinement* of a component P if there is an abstraction A and a representation R of appropriate functionality such that:

$$(R \circ Q \circ A) \Rightarrow P$$

holds.

Example 3.11 (Interface refinement of the summation component):

Suppose we want to add integers by a hardware device. At a higher level of abstraction, this is achieved by the summation components Sum_i treated above. In order to implement the adder in hardware, the integers of our abstract specification are to be replaced by finite sequences of bits, which are to be transferred sequentially. According to our definition we need a specification $R : (\mathbb{N}^\omega \rightarrow (\mathbb{B}^*)^\omega) \rightarrow \mathbb{B}$ of the representation and a specification $A : ((\mathbb{B}^*)^\omega \rightarrow \mathbb{N}^\omega) \rightarrow \mathbb{B}$ of the abstraction. Here we give just the informal description that R converts a stream of integers to a stream of equivalent binary representations and A performs the converse operation. Obviously, the equation $R \circ A = Id$ holds. Suppose that we also have defined an agent Sum_4 . To prove that Sum_4 is a correct implementation of, for instance, Sum_2 it has to be shown that the following implication holds:

$$(R \circ Sum_4 \circ A) \Rightarrow Sum_2 \quad \square$$

In this example we have replaced a single action (transmission of an integer) by several actions (transmission of individual bits) but have not changed the number of channels. One can also refine a specification in a way that a single channel is replaced by a group of new channels. For example, we could implement a hardware summation component that is connected to its environment by 8 input and 8 output channels.

3.8 Timed Component Specification

So far no explicit timing information is given in functional specifications. However, as we already pointed out in chapter 2, there are situations where the inclusion of time aspects is necessary. One possible reason is that a time critical component is to be specified, where timing aspects are part of the component's functionality. A second reason is that the inclusion of time may lead to simpler specifications. This will be explained below.

There are many possible models of time. Here, we consider a rather simple model: we assume a *global discrete time*. This means that in every time interval at most one message can be sent or received. Each element of a timed stream represents the (single) communication event on a channel during one time interval. The situation that *no* proper message has been sent during one time interval is modelled by a special element \checkmark (pronounced 'tick'). Formally, let A be a set of actions, which does not contain \checkmark . Then a *timed stream* is an element of $(A \cup \{\checkmark\})^\omega$.

The model reflects a global notion of time. In a timed environment, we have to make sure that every component conforms to such a view. In particular, we require that, as soon as the input stream is known for some time interval, the output stream is fixed for at least the same interval. This requirement is called the *time progress property*:

Definition 3.10 (Time progress property): Let f be a stream processing function of the functionality $f : \text{Instreams} \rightarrow \text{Outstreams}$. For any tuple of streams, $\text{minlength}(x_1, \dots, x_p)$ is defined to be $\min\{\#x_1, \dots, \#x_p\}$. Then f has the *time progress property* if:

$$\forall x \in \text{Instreams} : \text{minlength}(f(x)) \geq \text{minlength}(x) \quad \square$$

Of course, for an agent specification P we must prove that every f with $P(f)$ has the time progress property. See section 4 of [BD92] for further information on this class of timed functions.

Remember that we cannot define a non-strict fair merge within FOCUS. With timed streams, this is no longer a problem, since total timed streams (these are streams that correspond to a whole run of the system) are always infinite. Therefore, the *Fair-Merge* agent specified in section 3.4 is sufficient.

Another interesting aspect is that the introduction of timing information usually leads to an introduction of nondeterminism. This is because nondeterminism is often just introduced by the abstraction from time.

3.9 Proof Principles



This section contains advanced material, which can be skipped at the first reading.

An important method to prove properties of *stream processing functions* in general is the principle of *Noetherian induction*.

Proof Principle 3.1 (Noetherian induction): Let $f : \text{Instreams} \rightarrow \text{Outstreams}$ be an arbitrary stream processing function, and let P be a total predicate $(\text{Instreams} \times \text{Outstreams}) \rightarrow \mathbb{B}$. Select an arbitrary Noetherian strict order \prec on Instreams (for example, take the strict prefix order). To prove that $P(x, f(x))$ holds for all $x \in \text{Instreams}$, show that:

$$\forall x \in \text{Instreams} : ((\forall x' \prec x : P(x', f(x')))) \Rightarrow P(x, f(x))) \quad \square$$

In some cases, this method can be modified to make it more usable: consider a predicate that holds for an infinite stream whenever it holds for all finite prefixes of this stream. Such a predicate is called *admissible*. In particular, continuous predicates (and safety predicates) are always admissible. For other classes of admissible predicates see, for example, [Man74]. Given an admissible predicate P and a continuous function f , it must only be shown that $P(x, f(x))$ holds for all finite inputs x . This means that induction on the length of x can be used, which is often easier to handle than full Noetherian induction. All in all, we get the following modified proof principle:

Proof Principle 3.2 (Modified Noetherian induction): To prove that $P(x, f(x))$ holds for all $x \in \text{Instreams}$, show the following:

- (1) P is an admissible predicate.
- (2) f is a continuous stream processing function.
- (3) $P(x, f(x))$ holds for all *finite* $x \in \text{Instreams}$. This is often proved by induction on the structure of the input streams (see proof principle 2.1). \square

Of course, besides Noetherian induction there are more specific induction principles related to recursive declarations. We shall come back to these later on.

In the previous section on refinement we saw that we had to prove that a function f satisfies a trace specification in order to establish the connection between the trace specification and the functional specification. This proof can often be structured into two parts (provided that the function f never outputs infinitely many elements in response to a single input). Since our techniques are geared towards asynchronous message passing, we can imagine that every channel is allowed to introduce arbitrary (but finite) delays. This is reflected by the definition of the operator *Traces*. Therefore, a trace specification that is not closed with respect to finite output delays can certainly not be refined by a functional component. On the other hand, once it is shown that the trace specification *does* allow such delays, it suffices to show that those traces of f are allowed in which no output delays occur. We will first define the set of these traces:

Definition 3.11 (Normal form traces): The *normal form traces* of a stream processing function $f : \text{Instreams} \rightarrow \text{Outstreams}$ are defined by:

$$\text{NF-Traces}.f = \{ t \in \text{Traces}.f \mid \forall t' \in (I \cup O)^*, i \in I : t' \circ i \sqsubseteq t \Rightarrow f(I_1 \odot t', \dots, I_p \odot t') = (O_1 \odot t', \dots, O_q \odot t') \}$$

An alternative, and more operational, characterization of normal form traces is that immediately after every input all the outputs generated by this input appear. \square

The proof principle given above will now be summarized. See [Ded90] for a comprehensive example.

Proof Principle 3.3 (Refinement of a trace specification): Let (I, O, C) be a component-oriented trace specification, and f be a stream processing function. In order to prove that f refines (I, O, C) , show that (let A be $(I \cup O)$):

- (1) C is closed with respect to finite output delays:

$$\begin{aligned} \forall t \in A^\omega, t' \in A^*, o \in O, i \in I : \\ (t' \circ o \circ i \sqsubseteq t \wedge C(t)) \Rightarrow (\exists t'' \in A^\omega : t' \circ i \circ o \sqsubseteq t'' \wedge C(t'')). \end{aligned}$$

- (2) C is true for all normal form traces of f : $\text{NF-Traces}.f \subseteq \{t \in A \mid C(t)\}$. \square

Properties of equationally defined networks can be proved by exhibiting values for each of the network's internal channels as well as output channels, such that the defining equations are fulfilled, in other words by exhibiting a fixed point of the network. Such a set of values is generally not the *least* fixed point. If desired, this must be established independently. Note that, for example, safety predicates are downward closed, such that it is sufficient to show that some safety property holds for *any* fixed point (or, indeed, for any other value that is greater than the least fixed point).

We now show some proof methods for networks defined by the composition operators. For the parallel and sequential composition of functions, equational reasoning is the standard tool. Fixpoint induction can be used to prove properties of feedback loops.

Proof Principle 3.4 (Fixpoint induction for the feedback operator): Given an (n, m) -ary function of the form μf and an n -tuple of input streams (x_1, \dots, x_n) . Let g be a function such that $g(y_1, \dots, y_m) = f(x_1, \dots, x_n, y_m)$. Then for every m -tuple of output streams (y_1, \dots, y_m) , if $g(y_1, \dots, y_m) \sqsubseteq (y_1, \dots, y_m)$, then $(\mu f)(x_1, \dots, x_n) \sqsubseteq (y_1, \dots, y_m)$. \square

Other commonly used methods are folding and unfolding operations, and the following application of Kleene's theorem, which is often helpful to establish safety properties.

Proof Principle 3.5 (Computational induction): Given an (n, m) -ary function of the form μf , an admissible predicate P of appropriate type, and an n -tuple of input streams (x_1, \dots, x_n) . Define g by $g(y_1, \dots, y_m) = f(x_1, \dots, x_n, y_m)$. To prove $P((\mu f)(x_1, \dots, x_n))$, it suffices to show that, for all $n \in \mathbb{N}$:

$$(\forall i < n : P(g^i(\langle \rangle, \dots, \langle \rangle))) \Rightarrow P(g^n(\langle \rangle, \dots, \langle \rangle)).$$

Here the superscript i on g stands for function iteration. \square

As for trace specifications the proof principles for functional component specifications are an immediate consequence of well-known general principles from domain-theory and fixpoint theory.

Chapter 4

Implementation

4.1 Overview

The implementation phase covers the two final stages of a system development in FOCUS, namely:

- the derivation of an abstract program, and
- its transformation into a concrete program.



A major characteristic is the use of *programming* languages with fully fledged syntax and formal semantics.

During the preceding development steps we described distributed systems by logical and functional means, in the language of mathematics so to speak. This resulted in considerable expressive power, notational flexibility and allowed for the direct use of a variety of mathematical methods and techniques. However, since the ultimate aim of the methodology is program construction, one eventually has to switch over to representations that can be interpreted by a machine. This requires unambiguity and therefore a precisely defined syntax, and executability and therefore the absence of non-algorithmic descriptive constructs. In this chapter two algorithmic (programming) languages are introduced, which fulfill both requirements.

The first one — AL — is an *applicative language*. AL is close to the functional style used in the design phase (see section 1.3). In particular streams, stream processing components and networks may be expressed. System descriptions in AL are still quite abstract but already executable. They are therefore called *abstract programs*.

The second one — PL — is a *procedural language*. It comprises the classical imperative features like variables, assignments, loops, etc. and additional means for parallel programming and communication. Component networks can be repre-

sented in PL, too. Once a system is described in PL no further change of the formalism takes place. Therefore we call PL-programs *concrete*.

The applicative language is deliberately chosen to ease the step from functional descriptions (see chapter 3) to abstract programs. A functional system description consists of a number of communicating components interconnected by streams. Technically these components are denoted by predicates that determine sets of (continuous) stream processing functions. Streams are defined by (recursive) stream equations (see section 3.6.1). On the abstract program layer components are described in AL-syntax (here we use the keyword *agent*).



The *refinement relation* between functional component specifications and AL component declarations is provided by the denotational semantics of AL. It assigns a set of (continuous) stream processing functions to every component declaration. Thus the notion of *behaviour refinement* introduced in section 3.7 can straightforwardly applied to this situation.

Example 4.1 (Refinement of a functional specification by an AL declaration):

To get an idea how a functional component specification is refined by an AL declaration let us look at the following specification:

$$\begin{aligned}
 P &: (\mathbb{N}^\omega \rightarrow \mathbb{N}^\omega) \rightarrow \mathbb{B} \\
 P(f) &\equiv \exists g : P(g) \wedge \forall x \in \mathbb{N}^\omega, n \in \mathbb{N} : \\
 &\quad f(n \& x) = (2 * n) \& g(x) \vee f(n \& x) = (3 * n) \& g(x).
 \end{aligned}$$

It corresponds to the following AL fragment:

```

agent P  $\equiv$  chan nat i  $\rightarrow$  chan nat o :
    o  $\equiv$  (2 * ft(i)  $\square$  3 * ft(i)) & P(rt(i))
end.

```

Here *i* is the (local) name of the *input stream* of *P* and *o* is the (local) name of the *output stream*. The operator of (*erratic*) *nondeterminism* \square admits straightforward context free choice between two alternatives. So every element of *i* is multiplied either by 2 or 3. \square

We have already mentioned that abstract programs can be executed. Often, however, they are not efficient enough. During the previous development steps efficiency aspects only played a minor role. We mainly tried to achieve a clearly structured problem oriented solution. In the last phase, however, implementation details become increasingly important. One way to obtain more efficient solutions is to transform applicative programs into procedural ones. In our context this means to go from an applicative AL to a procedural PL program.



A distributed PL program constitutes the end point of a complete development process. The step from AL to PL programs may be carried out by transformations.

Transformations are correctness preserving rules that relate pieces of AL code to (semantically equivalent) pieces of PL code. In section 4.4 some transformation rules are given. For the moment we do not know whether the rules developed so far (see [Ded92]) are complete in the sense that every AL program can be translated into a PL program. It is one of the future research topics to look for a general strategy to guide the transformation process. If such a strategy exists then the AL \rightarrow PL transition could be automated, thereby shifting the methodological exit of a development process upward by one abstraction layer. At present neither AL nor PL are implemented¹. In the long run we plan to implement both of them. We believe that AL and PL are in particular suitable to be implemented on parallel distributed memory machines, e.g. hypercube architectures, since both languages are based on asynchronous message passing communication. The channels of PL can be mapped to the communication lines between the processors of the parallel machines. Either directly (and statically) or by means of appropriate system calls (e.g. by mailbox commands of the MMK, the Munich hypercube program library, see [BL90]). Dynamic networks could be tackled by load-balancing.

In the next two subsections the languages for abstract and concrete programs are described, mostly in terms of examples. Then the transformational style of program development is explained (see [CIP85] and [CIP87]). Some sample transformation rules are given. A comprehensive treatment of these issues can be found in [Ded92]. [Bro88a] contains a case study, where abstract and concrete programs are developed for a lift control module; in [DW92] components for a protocol are developed.

4.2 An Applicative Language



An abstract program consists of a number of *component declarations* and a *system of equations* describing their interconnection.

The paradigm underlying this representation is the same as on the previous layer (see section 3): a distributed system is modelled by concurrently working components that asynchronously exchange messages over unbounded, directed channels. The syntactic framework for abstract programs is provided by the applicative language AL. The language is derived from AMPL (“applicative multiprogramming language”) developed in [Bro86]. Conceptually it can be compared to functional languages like HASKELL [HJW⁺91] or dataflow languages like LUCID [WA85]. AL contains means for the definition of stream processing functions, and moreover admits the definition of mutually recursive stream equations. Here is a simple numerical AL-program:

¹In fact, there exists an implementation of AMPL, a predecessor of AL, on the SUN SPARC-station (see [Nue88]). Moreover some experiments concerning the implementation of AL on a INTEL hypercube using the Munich program library MMK are under way (see [Gor92]).

Example 4.2 (A simple AL-program):

```

program factorial  $\equiv \rightarrow$  chan nat o :
  funct fac  $\equiv$  nat n  $\rightarrow$  nat : if  $n = 0$  then 1 else  $n * fac(n - 1)$  fi,
  agent streamfac  $\equiv$  chan nat i  $\rightarrow$  chan nat o :
     $o \equiv fac(ft.i) \& streamfac(rt.i)$ 
  end,
  agent streamadd  $\equiv$  chan nat i1, i2  $\rightarrow$  chan nat o :
     $o \equiv (ft.i_1 + ft.i_2) \& streamadd(rt.i_1, rt.i_2)$ 
  end,
   $o \equiv streamfac(s)$ ,
   $s \equiv streamadd(0 \& s, t)$ ,
   $t \equiv 1 \& t$ 
end factorial.

```

First the program name and its input and output streams are defined. The example program is called *factorial* and has only one output stream of type **nat**. It generates the stream 1! 2! 3! ... of all factorials in increasing order. \square

In the headers of programs and components (for which the keyword **agent** is used) streams are declared by means of the keyword **chan**. This keyword (instead of **str** for instance) is motivated by the fact that streams represent communication channels and because we want to have a common syntactic interface for both the applicative and the procedural language (see section 4.3).

AL is a typed language. Every data object or stream belongs to some type, which semantically means that it is an element of a particular cpo. In this sense **chan nat** stands for \mathbb{N}^ω , the prefix ordered cpo of streams over \mathbb{N} .

There is a syntactic distinction between *functions* mapping data objects to data objects and *components* mapping data objects and/or streams to streams. Functions like *fac* are defined by expressions. They may have zero or more named input parameters and yield exactly one output. With *if-then-else-fi* and (mutually) recursive function calls the standard concepts of functional languages can be used to define them. However, AL is not a higher order language so that functions and components can not be passed as arguments or results.



Components are modelled by (sets of) stream processing functions.

They have zero or more named input parameters and one or more output parameters. Therefore in component headers output parameters are named, too. Any combination of data objects and/or stream parameters are allowed as inputs but only streams are allowed as outputs. This restriction is imposed since AL components are intended to model system components that communicate via channels. On the other hand, admitting non-stream inputs makes programming more flexi-

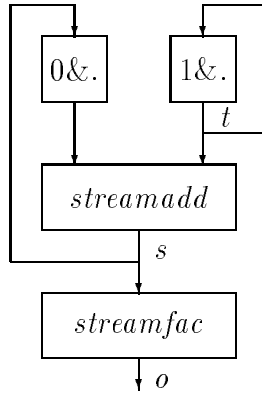
ble. The body of a component is built by equations just like the equational part of complete programs:

```

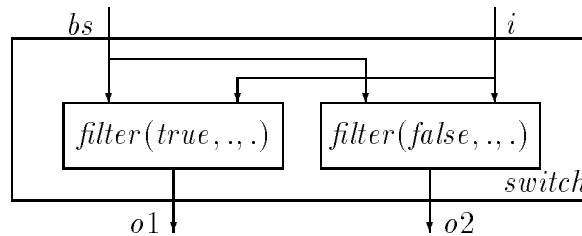
agent filter  $\equiv$  bool b, chan bool bs, chan nat i  $\rightarrow$  chan nat o :
    o  $\equiv$  if ft.bs = b then ft.i & filter(b, rt.bs, rt.i)
        else filter(b, rt.bs, rt.i) fi
end,
agent switch  $\equiv$  chan bool bs, chan nat i  $\rightarrow$  chan nat o1, o2 :
    o1  $\equiv$  filter(true, bs, i),
    o2  $\equiv$  filter(false, bs, i)
end.

```

Every equation has a number of stream identifiers on the left hand side and an expression of adequate arity and type on the right hand side. Every stream occurs at most once on a left hand side: output streams occur exactly once, while input streams never occur. Streams that are neither input nor output are called *internal*. Since output streams and internal streams may appear on both the left and the right hand sides of equations, one can express sequential composition of components as well as feedback loops. This becomes obvious if the graphical representation of equational systems is considered. The *factorial* program, for instance, corresponds to the following net:



The component *switch* is graphically represented as follows:



The relationship between equations and nets can be made strictly formal (see [Bro86], [Bro88b]).



Because the body of a component can again be seen as a network, AL supports hierarchical structuring of programs. Certain components that appear as black boxes on one layer may be refined as networks of more elementary components on a lower layer. This is just *distribution refinement* as defined in section 3.7.

Note that components are restricted from being defined within the body of other components. However, they can be called there. This way already finished components can be used for the definition of new ones. Two different types of recursion may be used:

- Stream recursion: the body of a component may contain (mutually) recursive stream definitions. As in the *factorial*-example this leads to networks with feedback loops.
- Functional recursion: a component may be called in its own body (see *streamfac* or *streamadd* above). Since this call can be unfolded arbitrary often (during run time), this kind of recursion leads to infinite networks.

Conceptually stream recursion can be viewed as recursion in time (a particular component is used more than once for items that are fed back), while functional recursion corresponds to recursion in place (functional recursive components can be unfolded and thereby lead to a number of different instances of the same component working in parallel). Often it is a particular development goal to transform a functional recursive component into a stream recursive one, thus replacing a potentially infinite network by a finite one.

Since AL comprises the finite choice operator \square , one can define nondeterministic expressions, (AL-)functions, components and programs. The denotational semantics of AL relates any of these syntactic categories to sets of continuous functions.



Consider, for instance, the following component declaration:


```
agent  $f \equiv \mathbf{chan\ nat\ } i \rightarrow \mathbf{chan\ nat\ } o :$ 
       $ES$ 
end.
```

Here, ES is a system of equations that may contain nondeterministic right hand sides. The semantic mapping \mathbf{F} gives a set of stream processing functions

$$\mathbf{F}[\mathbf{agent\ } f \equiv \dots : ES \mathbf{end}] \subseteq \mathbb{N}^\omega \rightarrow \mathbb{N}^\omega,$$

which is taken to be the meaning of f .

This approach is consistent with functional component specifications. It avoids the well-known anomalies (see [Kel78], [BA81]) that appear when a relational semantics is used. (Such a semantics assigns relations or set valued functions to nondeterministic declarations.)

 Based on the semantic mapping \mathbf{F} it is easy to define when an AL component declaration *refines* a functional component specification: let

$$P : (\text{Instreams} \rightarrow \text{Outstreams}) \rightarrow \mathbb{B}$$

be a component specification. Then an AL component declaration p refines P iff:

$$\mathbf{F}[\mathbf{agent} \ p \equiv \text{Instreams} \rightarrow \text{Outstreams} : \text{ES} \ \mathbf{end}] \subseteq \{f \mid P(f)\}.$$

Obviously, when going from P to p the specification formalism has changed, but we might also constrain the behaviour.

Example 4.3 (The message transmission system described in AL):

The functional version of the transmitter presented in example 3.6 can be immediately represented as an AL-component. Let

$$\text{queue} \equiv M^*$$

be the datatype of finite streams (sequences) over M . The component *transmitter* looks as follows:

```
agent transmitter  $\equiv$  nat mstor, chan in i
                     $\rightarrow$  chan ack a, chan out o :
    (a, o)  $\equiv$  h(mstor,  $\langle \rangle$ , i)
end.
```

mstor is the parameter that specifies the maximum number of messages the transmitter is able to store. The auxiliary function h is defined below:

```
agent h  $\equiv$  nat k, queue q, chan in i
           $\rightarrow$  chan ack a, chan out o :
    (a, o)  $\equiv$  if ft.i = req
              then if #q > 0 then (y, ft.q&z) else (y, fail&z) fi
              else if #q < k then (ok&y, z) else (error&y, z) fi
              fi,
    (y, z)  $\equiv$  if ft.i = req
              then if #q > 0 then h(k, rt.q, rt.i) else h(k, q, rt.i) fi
              else if #q < k then h(k, q o ft.i, rt.i) else h(k, q, rt.i) fi
              fi
end.
```

Note that h has an additional state parameter: q represents the queue of currently stored messages. In each recursive call it is updated appropriately. Initially q is set to $\langle \rangle$, which represents the empty queue. This example also demonstrates the close relationship between state-oriented functional specifications and AL-components (see example 3.6). In fact, *transmitter* refines (in the sense defined above) the functional component specification Trans_2 . \square

4.3 A Procedural Language



According to our terminology, a program is called “concrete” if there is no further refinement necessary (in particular no rewriting into another formalism). Methodologically a concrete program constitutes the final result of the development process. It depends on the designer (and his customer) whether a program is considered concrete. In this section as an example a procedural language is proposed for the representation of concrete programs.

While on the applicative layer distributed systems were represented by stream processing components and recursive stream equations, on the procedural layer we have *procedures* and *channels*. The procedural language PL used here comprises the usual imperative constructs:

- variables,
- assignments,
- while loops,
- procedural components (procedures).

Moreover it is equipped with means for communication and concurrency:

- components can be executed in parallel,
- channels are used to establish directed point to point communication,
- they are accessed by *send* and *receive* commands.

Let c be a channel, x be a variable and E be an expression (c , x and E of compatible type). Then $c?x$ is the command by which the first element of the input channel c is removed and assigned to x . If c is empty then the execution of $c?x$ is delayed until an item becomes available. This implies that the execution may be delayed forever, if that item never appears. The command $c!E$ first evaluates E and then sends the result to the output channel c . Provided that the evaluation of E terminates $c!E$ is never delayed. This models asynchronous communication. (Thus the meaning of $?$ and $!$ in PL should not be confused with the meaning of these operators in CSP (see [Hoa85]) where they stand for synchronous communication).



In PL two types of components can be distinguished. The first type is called *sequential*.

The body of a sequential component consists of a (sequential) statement sometimes preceded by variable declarations. The procedural version of the *filter*-component

from section 4.2 is a sequential one.

```

agent filter  $\equiv$  bool b, chan bool bs, chan nat i  $\rightarrow$  chan nat o :
  var bool x; var nat y;
  loop bs?x; i?y;
    if x = b then o!y else skip fi
  pool
end.

```

Here **loop** *stat* **pool** denotes the infinite execution of the statement *stat*, i.e. a nonterminating loop.



Components belonging to the second type are called *hierarchical* or *parallel*. The body of an hierarchical component consists of a number of parallel component calls.

Syntactically these calls are represented by equations. Thus an hierarchical PL component looks very much like an AL component:

```

agent switch  $\equiv$  chan bool bs, chan nat i  $\rightarrow$  chan nat o1, o2 :
  i1, i2  $\equiv$  split(i),
  bs1, bs2  $\equiv$  split(bs),
  o1  $\equiv$  filter(true, bs1, i1),
  o2  $\equiv$  filter(false, bs2, i2)
end.

```


Here *split* is an component that copies every input message to both of its output channels. Every equation in the body of *switch* stands for an component call generating a new instance of the called entity. Channel identifiers on the right hand sides denote input channels, those on the left hand sides denote output channels. Parallelism is expressed by simple juxtaposition of equations. Since recursion is permitted for PL programs, new channels can be introduced and dynamically changing networks can be modelled. Here a channel identifier occurs at most once on the left hand side of the equations and at most once on the right hand side. This distinguishes PL from AL where (stream) identifiers may occur twice or more on the right hand sides. At this point the difference between channels and streams becomes obvious.



Conceptually the (denotational) semantics of AL and PL are quite similar. In both cases the meaning of components is described by sets of (continuous) stream processing functions.



A special treatment is needed only for sequential components that are defined by statements. Semantically a statement corresponds to a set of state transformations that change the contents of its input and output channels. (see [BL91]). By suitable abstraction (see [Ded92]) every state transformation τ can be related to a stream processing function f_τ . So we obtain a coherent functional framework.

 The *refinement relation* between AL and PL is obvious: a PL declaration q refines an AL declaration p , iff

$$\begin{aligned} & \mathbf{F}[\mathbf{agent} \ q \equiv \mathit{Instreams} \rightarrow \mathit{Outstreams} : \mathit{STAT} \ \mathbf{end}] \\ & \subseteq \\ & \mathbf{F}[\mathbf{agent} \ p \equiv \mathit{Instreams} \rightarrow \mathit{Outstreams} : \mathit{ES} \ \mathbf{end}] \end{aligned}$$

The same definition applies if p and q are both from AL or both from PL.

On both the applicative and the procedural layer communication is asynchronous. Channels can be viewed as unbounded buffers accessed by (non-blocking) send and (blocking) receive commands. The decision to use just these two commands facilitates the semantic definitions. On the other hand it restricts the expressive power of the language. For instance a fair (non-strict) merge can not be expressed in PL (nor in AL; see [Bro86]). Obviously one can think of various enhancements overcoming this constraint. One possibility would be to introduce a polling statement that tests whether a channel is currently empty, yielding *true* in this case and *false* otherwise. Another option would be to introduce a *disjunctive wait* command allowing an agent to wait for two channels at the same time taking the first item that arrives. In fact both constructs can be seen as timing features; they allow for the description of agents that show time dependent behavior. This heavily influences the semantic description. When having to choose between simplicity and expressiveness we vote for the first and thus decide not to include such constructs.

Example 4.4 (The message transmission system described in PL):

The procedural version of the transmitter, described as AL-component in the previous section, looks as follows:

```

agent transmitter  $\equiv$  nat mstor, chan In i
                     $\rightarrow$  chan ack a, chan out o :
    var in x;
    var queue q := empty;
    loop i?x;
      if x = req
      then if #q > 0 then o!ft.q; q := rt.q else o!fail fi
      else if #q < mstor then o!ok; q := q o x else o!error fi
      fi
    pool
  end.

```

Note that we do not need an auxiliary component h here, because we can store the incoming messages in a variable of the type **queue**. The transmitter is described as a sequential component. \square

4.4 Transformational Synthesis of Concrete Programs

Program transformation is a highly formalized method of software development. By applying only semantics preserving transformation rules (which are rules that preserve an implementation relation or even semantical equivalence) to a given specification a program is derived that is correct by construction.

Since the early seventies various transformation calculi have been developed for sequential programs (see, for instance, [BD77] or [CIP85], an overview can be found in [Fea87]), but only recently has the transformational approach been applied to concurrent programs (see [Bar85], [Bar88], [KB⁺90a], [Old91]).

In FOCUS transformation rules are mainly applied in the implementation phase in order to deduce a concrete program from an abstract program. This restricted use is due to the following reasons:

In the early development phases the planned system is described at a quite abstract level. Many design decisions are necessary to derive more concrete representations. Since these decisions are rather specific and problem dependent, it will be difficult to (re-)use standard transformation rules. Furthermore, the application of transformation rules requires a precisely defined syntactical frame. In FOCUS such a frame is only fixed for the implementation phase. On the previous layers we deliberately choose a more liberal style of notation (which nevertheless is strictly formal). At the beginning of a system development we try to achieve an adequate mixture of top-down steps (development steps) and bottom-up steps (proof steps), while on the later stages the emphasis is shifted towards the top-down-steps conducted in the form of program transformation.



In the sequel some basic rules for the transformation of applicative (abstract) programs into procedural (concrete) ones are given. Actually not all transformations really relate applicative components/programs to procedural components/programs. Some of them are applied to applicative components and yield applicative components again. Methodologically one applies the later group of rules to a given component definition until it is in a suitable form to admit the use of the first group.

Every rule has the form:

$$\frac{I}{O} \ C$$

where I and O are program fragments and C is an application condition. I is called the *input template*, O is called the *output template* (see [Par90]). A rule is correct, when O is a *refinement* of I , whenever C holds.

As described in the previous section the refinement relation is formalized by set inclusion. Thus a rule is correct if

$$C \Rightarrow \mathbf{F}[O] \subseteq \mathbf{F}[I].$$

Every transformation rule given in the sequel is correct in this sense and therefore represents a refinement. Correctness proofs can be found in [Ded92]. Transformation rules can be applied locally, i.e. to small fragments of complete programs. Here we can again take advantage from the similarity between AL and PL.



Although both languages are treated distinctly in the previous sections they can be integrated into a wide spectrum language (see [CIP85]). So mixed forms become possible, which nevertheless are syntactically correct and semantically sound.

We now start with the most fundamental transformation rule relating (functional) recursive components to iterative ones:

- rule: recursion-to-iteration I

$$\begin{array}{l}
 \mathbf{agent } f \equiv \mathbf{chan } \mathbf{v } i \rightarrow \mathbf{chan } \mathbf{w } o : \\
 \quad o \equiv F[ft.i] \& f(rt.i) \\
 \mathbf{end} \\
 \hline
 \mathbf{agent } f \equiv \mathbf{chan } \mathbf{v } i \rightarrow \mathbf{chan } \mathbf{w } o : \\
 \quad \mathbf{var } \mathbf{v } x; \\
 \quad \mathbf{loop } i?x; o!F[x] \mathbf{pool} \\
 \mathbf{end}.
 \end{array}$$

The form of recursion displayed by f is called *tail-recursive-modulo-cons*, where *cons* stands for the stream constructor $\&$. In the realm of stream processing this kind of recursion plays the same role that ordinary tail-recursion plays for sequential programs (see [BW81]). The above rule basically uses the fact that $\&$ is non-strict in its second argument, i.e. $E\&S$ can partially be evaluated without evaluating the stream expression S on the left. Operationally this means that the first input item can be processed before the second one arrives, thus admitting a loop construct on the procedural layer. A lot of similar agents can be transformed by rules like this, for instance:

- rule: recursion-to-iteration II

```

agent  $f \equiv \mathbf{u} \ p, \mathbf{chan} \ v \ i \rightarrow \mathbf{chan} \ w \ o :$ 
   $o \equiv \mathbf{if} \ B[p, ft.i] \ \mathbf{then} \ F[p, ft.i] \ \& \langle \rangle$ 
   $\mathbf{else} \ G[p, ft.i] \ \& \ f(T[p, ft.i], rt.i) \ \mathbf{fi}$ 
end

```

```

agent  $f \equiv \mathbf{u} \ p, \mathbf{chan} \ v \ i \rightarrow \mathbf{chan} \ w \ o :$ 
  var  $u \ x := p; \mathbf{var} \ v \ y;$ 
   $i?y; \mathbf{while} \ \neg B[x, y] \ \mathbf{do}$ 
     $o!G[x, y]; x := T[x, y]; i?y$ 
  od;
   $o!F[x, y];$ 
  close.o
end.

```

Here the role of the object parameter p becomes apparent. It is used as local variable that is updated appropriately in each cycle. The ft - rt combination of the applicative layer corresponds to the $i?y$ command on the procedural layer since this assigns the first item of i to y and removes it.

Equationally defined components,

```

agent  $f \equiv \mathbf{chan} \ u \ i \rightarrow \mathbf{chan} \ u \ o :$ 
   $s_1 \equiv f_1(t_{11}, \dots, t_{1m_1}),$ 
   $\dots,$ 
   $s_n \equiv f_n(t_{n1}, \dots, t_{nm_n})$ 
end,

```

where $o \in \{s_1, \dots, s_n\}, \{t_{11}, \dots, t_{nm_n}\} \subseteq \{i, s_1, \dots, s_n\}$ and every t_{ij} occurs exactly once on a right hand side, can immediately be seen as PL components.

Thus one can bring applicative agents into procedural form by simply transforming the equations in their body. General stream expressions on the right hand side of AL-equations must be substituted by component calls of the above type. Various rules are necessary for this purpose (see [Ded92]). They can be obtained as particular combinations from the following basic *network transformation rules*:

- folding/unfolding of equations
- folding/unfolding of agent definitions,
- introduction of auxiliary streams etc.

Due to lack of space only one rule is presented here:

- rule: unfolding of stream equations

```

agent  $f = \mathbf{chan\ v\ } i \rightarrow \mathbf{chan\ w\ } o$ 
       $s_1 \equiv S_1, \dots, s_k \equiv S_k, \dots, s_l \equiv S_l, \dots, s_n \equiv S_n$ 
end

```

```

agent  $f = \mathbf{chan\ v\ } i \rightarrow \mathbf{chan\ w\ } o$ 
       $s_1 \equiv S_1, \dots, s_k \equiv S_k[S_l/s_l], \dots, s_l \equiv S_l, \dots, s_n \equiv S_n$ 
end

```

Using network transformation rules only leads to quite schematic concrete programs. In particular the recursive structure of the applicative program is entirely transferred to the procedural version. Often one wishes to avoid this, for instance, to admit a static mapping of processes to processors. Then rules must be used that realize a dynamic network on the applicative layer by a static network on the procedural layer. This can be done by replacing functional recursion by stream recursion. For the moment it is not clear whether this is always possible. For special cases rules can be found in [Ded92].

Once the step from AL to PL has been made further rules can be applied in order to optimize the procedural program obtained by then. Here one can make use of many standard transformations for procedural programs known from the sequential case. But also more specific rules can be applied, for instance, certain feedback loops may be replaced by local variables.

In general, the application of a transformation rule requires three separated activities. First a rule is chosen and (syntactically) matched with the agent/program under consideration. Then the application conditions is checked and finally the rule is applied by substituting the (matched) input template by the output template. All these activities can be supported by appropriate tools (see [CIP87], [KB⁺90b]). Nevertheless transformational program development is not automatic programming. The designer still plays an important role: he has to check the validity of application conditions. Furthermore, although it is desirable that there is a considerable number of predefined transformation rules available, from time to time he may find it necessary to develop new transformations appropriate for his current problem. A complete automatic translation would be possible if we can find a transformation strategy that is universally applicable. Here many questions remain to be solved.

Chapter 5

Conclusion

FOCUS is not a syntactically fixed formalism but rather a collection of mathematical models, logical concepts and rules centered around the idea of descriptive, functional system modelling based on the notion of streams. Much more work has been done in this area than has been presented in the previous chapters. They are intended to give a rather informal basic introduction to FOCUS. Much more work needs to be done to explore the potentials and limitations especially from the practical point of view.

It was the goal of this presentation to demonstrate a formal framework for the systematic development of information processing systems.

Chapter 6

Glossary

Action: Indivisible unit of activity (at the considered abstraction level).

Agent: Component of a system modelled by a set of stream processing functions.

AL: Applicative language for the representation of abstract programs allowing the definition of streams and stream processing functions.

Algebraic specification: Property oriented specification of a data structure and related operations using axioms (mostly conditional equations).

Bottom-up step: Development step by which a more concrete system description is related to more abstract one (for instance by verification step).

Communication: Exchange of messages.

Component: Subsystem of a system: in FOCUS a component communicates via its *interface* with its environment; the interface is given in terms of the input/output-behaviour of the component. In the design phase a component is modelled by a set of stream processing functions.

Component-oriented specification: (Trace) specification of a system structured into components.

Denotational semantics: Approach to program semantics that assigns mathematical objects (e.g. functions) to syntactical entities in order to describe their meaning.

Distribution: Spatial or conceptual decomposition of a system.

Interaction: Causality between actions (especially send and receive actions) of distributed system components.

Global specification: (Trace) specification of the whole closed system without explicitly referring to particular system components.

Implementation relation: Relation between two system descriptions one being more abstract the other one being more concrete.

Liveness property: Property of a system of a form that misbehaviour with respect to this property cannot be finitely observed (cannot be observed just by looking at finite prefixes of a trace).

Nondeterminism: Freedom of choice between several behaviours of a system in an instantiation without possibility to influence this choice by the environment.

Network: Collection of agents connected by communication channels.

PL: Procedural language for the representation of concrete channels based on imperative constructs and asynchronous channels.

Port: Name for an input or output channel of an agent.

Process: Instantiation (run) of a system or system component.

Refinement: Replacing a system description by one containing possibly more details.

Safety property: Property of a system of a form that misbehaviour of a system with respect to this property can be finitely observed (by looking at finite prefixes of traces).

State: Representation of the relevant aspects of a finite history of a system by some element from a mathematical set (called the state space).

Stream: Finite or infinite sequence of elements; used for communication channels, histories of actions by traces, or histories of states of state machines.

System: Conceptual or technical distinguished structure with a dynamic behaviour.

Top-down step: Development step by which a more concrete system description is derived from a more abstract one.

Trace: Stream of actions modelling a process.

Transformation Technique: Formal method of software development based on semantics (correctness) preserving transformation rules.

Verification: Showing that a system description fulfils its specification by giving a formal proof.

Bibliography

- [BA81] J.D. Brock and W.B. Ackermann. Scenarios: A model of non-determinate computation. In J. Díaz and I. Ramos, editors, *Formalization of Programming Concepts*, volume 107 of *LNCS*, pages 252–259. Springer, 1981.
- [Bar85] D. Barstow. Automatic programming for streams. In *Proc. 9th International Joint Conference on Artificial Intelligence*, pages 232–237. 1985.
- [Bar88] D. Barstow. Automatic programming for streams II. In *Proc. 10th International Conference on Software Engineering*, pages 439–447. 1988.
- [BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.
- [BD92] M. Broy and C. Dendorfer. Modelling of operating system structures by timed stream processing functions. *Journal of Functional Programming*, 2(1):1–21, 1992.
- [BDD⁺92] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. Summary of case studies in FOCUS — a design method for distributed systems. SFB-Report 342/3/92 A, Technische Universität München, January 1992.
- [BFG⁺92] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, and K. Stolen. The Requirement and Design Specification Language SPECTRUM, An Informal Introduction, Version 0.3. Technical Report TUM-I9140, Technische Universität München, May 1992.
- [BL90] T. Bemmerl and T. Ludwig. MMK — a distributed operating system kernel with integrated dynamic loadbalancing. In H. Burkhard, editor, *CONPAR '90 - VAPP IV*, volume 457 of *LNCS*, pages 744–755. Springer, 1990.
- [BL91] M. Broy and C. Lengauer. On denotational versus predicative semantics. *Journal of Computer and System Sciences*, 42(1):1–29, 1991.
- [Bro86] M. Broy. A theory for nondeterminism, parallelism, communication, and concurrency. *Theoretical Computer Science*, 45:1–68, 1986.

- [Bro88a] M. Broy. An example for the design of a distributed system in a formal setting — the lift problem. Technical Report MIP 8802, Universität Passau, February 1988.
- [Bro88b] M. Broy. Nondeterministic dataflow programs: how to avoid the merge anomaly. *Science of Computer Programming*, 10:65–85, 1988.
- [Bro89] M. Broy. Towards a design methodology for distributed systems. In M. Broy, editor, *Constructive Methods in Computing Science*, volume 55 of *NATO ASI Series F: Computer and System Sciences*, pages 311–364. Springer, 1989.
- [Bro90] M. Broy. Functional specification of time sensitive communicating systems. In G. Rozenberg J.W. de Bakker, W.-P. de Roever, editor, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *LNCS*. Springer, 1990.
- [Bro92a] M. Broy. Compositional refinement of interactive systems. Working material, International Summer School on Program Design Calculi, 1992.
- [Bro92b] M. Broy. (Inter-)action refinement: The easy way. Working material, International Summer School on Program Design Calculi, 1992.
- [BW81] F.L. Bauer and H. Wössner. *Algorithmische Sprache und Programmentwicklung*. Springer, 1981.
- [CIP85] The CIP Language Group. *The Munich Project CIP Vol. I: The Wide Spectrum Language CIP-L*, volume 183 of *LNCS*. Springer, 1985.
- [CIP87] The CIP System Group. *The Munich Projekt CIP Vol. II: The Program Transformations System CIP-S*, volume 292 of *LNCS*. Springer, 1987.
- [CM88] K.M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, 1988.
- [Ded90] F. Dederichs. System and environment: The philosophers revisited. Technical Report TUM-I9040, Technische Universität München, October 1990.
- [Ded92] F. Dederichs. Transformation verteilter Systeme: Von applikativen zu prozeduralen Darstellungen. SFB-Report 342/17/92 A, Technische Universität München, Technische Universität München, August 1992.
- [Den91] C. Dendorfer. Funktionale Modellierung eines Postsystems. SFB-Report 342/28/91 A, Technische Universität München, November 1991.
- [DHR90] E. Dubois, J. Hagelstein, and A. Rifaut. ERAE: A formal language for expressing and structuring real-time requirements. Draft Version, June 1990.

- [DW92] C. Dendorfer and R. Weber. From service specification to protocol entity implementation — an exercise in FOCUS. SFB-Report 342/4/92 A, Technische Universität München, January 1992. Also to appear in: Proc. 12th symposium on protocol specification, testing, and verification 1992.
- [Fea87] M.S. Feather. A survey and classification of some transformation approaches and techniques. In L.G.L.T. Meertens, editor, *Program Specification and Transformation*, pages 165–195. Elsevier, 1987.
- [Gor92] S. Gorlatch. Parallel program development for a recursive numerical algorithm: a case study. SFB-Report 342/7/92 A, Technische Universität München, March 1992.
- [HJW⁺91] P. Hudak, S. Peyton Jones, P. Wadler, et al. Report on the programming language Haskell, a non-strict purely functional language (version 1.1). Technical Report YALEU/DCS/RR777, Yale University, Department of Computer Science, August 1991.
- [Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [Kah74] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing 74*, pages 471–475. North-Holland, 1974.
- [KB⁺90a] B. Krieg-Brückner et al. PROgram development by SPECification and TRAnSformation. vol. I (Methodology). PROSPECTRA Report M.1.1.S3-R-55.2, Universität Bremen, 1990.
- [KB⁺90b] B. Krieg-Brückner et al. PROgram development by SPECification and TRAnSformation. vol. III (System). PROSPECTRA Report M.1.1.S3-R-57.2, Universität Bremen, 1990.
- [Kel78] R.M. Keller. Denotational models for parallel programs with indeterminate operators. In E.J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 337–366. North Holland, 1978.
- [KM77] G. Kahn and D.B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing 77*, pages 993–998. North-Holland, 1977.
- [LA90] L. Lamport and M. Abadi. Composing specifications. In G. Rozenberg J.W. de Bakker, W.-P. de Roever, editor, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of LNCS, pages 1–41. Springer, 1990.
- [Lam83] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, pages 190–222, 1983.

- [LS87] J. Loeckx and K. Sieber. *The Foundations of Program Verification*. Wiley-Teubner, 2nd edition, 1987.
- [LS89] N. Lynch and E. Stark. A proof of the Kahn principle for input/output automata. *Information and Computation*, 82:81–92, 1989.
- [Man74] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.
- [Nue88] H. Nueckel. Eine Zeigerimplementierung von Graphreduktion für eine Datenflußsprache. Universität Passau, 1988. Diploma Thesis.
- [Old91] E.-R. Olderog. Towards a design calculus for communicating programs. In J.C.M. Baeten and J.F. Groote, editors, *CONCUR '91*, volume 527 of *LNCS*, pages 61–77. Springer, 1991.
- [Pan90] P. Pandya. Some comments on the assumption commitment framework for compositional verification of distributed programs. In G. Rozenberg J.W. de Bakker, W.-P. de Roever, editor, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *LNCS*, pages 622–640. Springer, 1990.
- [Par90] H.A. Partsch. *Specification and Transformation of Programs*. Texts and Monographs in Computer Science. Springer, 1990.
- [WA85] W. Wadge and E. Ashcroft. *Lucid, the dataflow programming language*. Academic Press, 1985.
- [Web92] R. Weber. Eine Methodik für die formale Anforderungsspezifikation verteilter Systeme. SFB-Report 342/14/92 A, Technische Universität München, March 1992.