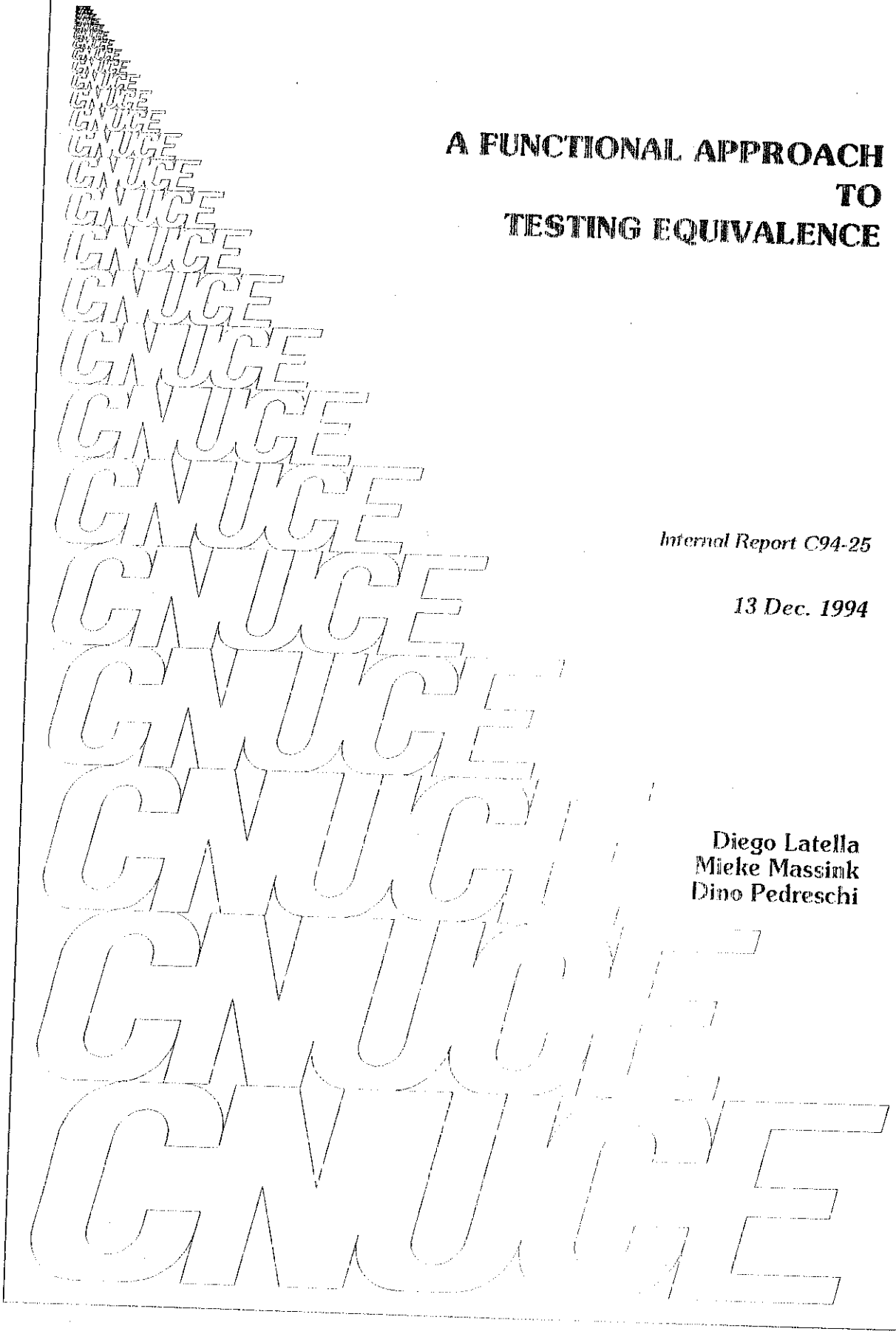

A FUNCTIONAL APPROACH TO TESTING EQUIVALENCE

Internal Report C94-25

13 Dec. 1994

Diego Latella
Mieke Massink
Dino Pedreschi

A FUNCTIONAL APPROACH TO TESTING EQUIVALENCE

Internal Report C94-25

13 Dec. 1994

Diego Latella
Mieke Massink
Dino Pedreschi

The work presented in this technical report
has been funded by:

Consiglio Nazionale delle Ricerche - C.N.R.
Progetto Coordinato *Metodologie, architetture, ambienti di progetto e valutazione per sistemi di elaborazione distribuiti*

Consiglio Nazionale delle Ricerche - C.N.R.
Progetto Bilaterale *Estensioni probabilistiche e temporali dell'algebra di processi LOTOS, basate su strutture di eventi, per la specifica e analisi quantitative di sistemi distribuiti*

Nederlandse Organisatie voor Wetenschappelijk Onderzoek - N.W.O.
Project n. 612-316-104/64

A Functional Approach to Testing Equivalence

Diego Latella * Mieke Massink † Dino Pedreschi ‡

CNUCE Internal Report C94-25 December 1993

Abstract

In this paper we propose a model for the specification of non-deterministic systems based on a functional approach. The behaviour of a system is specified by means of a predicate, *functional specification* in the sequel, which characterize *a set* of sequence processing monotonic functions. We introduce a set of combinators on sequence processing functions and on functional specifications. Such combinators together give rise to a *Functional Algebra* (FA) and then allow for systematically build specifications in a compositional way. FA is takes as the natural denotational model for a simple algebra of processes with input/output actions (PA). PA operators are *STOP*, *input/output-prefix*, which is a functional variant of action-prefix, and *choice*. We show that, under proper conditions, FA is fully abstract w.r.t Testing Equivalence when actions are interpreted as input/output pairs. Moreover, we show how full abstraction can still be guaranteed even for the *whole* PA provided the notion of *experimenter* be modified in order to take into account the conceptual separation between input events and output events peculiar of the functional approach.

*C.N.R., Ist. CNUCE, Via S. Maria 36, Pisa, Italy
†University of Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
‡University of Pisa, Corso Italia 40, 56100 Pisa, Italy

Contents

Chapter 1

Introduction

A number of formal models and languages for the specification of concurrent systems and for formally reasoning about them have been proposed in the literature [Rei85, Hen88, Hoa85, Mil89]. Many of them are based on the notion of process algebra and are provided with an operational semantics and various notions of equivalences [vG90].

In the theory of formal semantics for sequential (i.e. non-concurrent) systems, a functional approach has shown many advantages. This is mainly due to its sound and well understood mathematical basis which allows for the direct use of rigorous proof styles like transformational reasoning [DC89]. Moreover, when objects are defined as fixed-points of recursive equations, suitable induction principles [BW88, MNV73] can be used.

In [Kah92] a fixed-point semantics for a class of distributed systems has been proposed. The behaviour of a system is modeled as a function from sequences (sometimes called "streams") of input messages to sequences of output messages. Of course such an approach applies only to deterministic speccifications of system behaviour. On the other hand, using nondeterminism as a way of modeling is essential when reasoning about concurrent systems, so, recently, several approaches have been proposed for extending stream semantics to nondeterminism [Abr84, Abr89, Bro90, Bro92a, Bro92b, Dyb86, DSS[illegible] Ong93, San92].

In particular we consider [illegible] Sander's proposals very promising. In these approaches a specification is defined as a set of monotonic sequence processing functions which is itself defined as a higher order boolean function. In the approach of [illegible] number of combinators on specifications are defined which reflect the architectural aspects of the system. The main combinators in this approach are based on function composition (pipeline or cascade), parallel composition and feedback. Central in this work are notions of refinement of specifications which are compositional with respect to these

combinators. In Sander's approach the topology of the system is given by a higher order function that is defined as the minimal fixed point solution of a set of recursive equations. Central to this approach is a higher order logics for verifying that a function satisfies a given specification.

Of all the above mentioned works only [Ong93] addresses the issue of algebraic relations between specifications that take progress properties into account (deadlock etc.) like observational or testing equivalences do for process algebras. Indeed, low expressive power w.r.t. progress properties like deadlock has been an argument against stream based approaches in the concurrency theory community. In particular, it is commonly maintained that such approaches can be *at most* as much powerful as *trace semantics* in the context of process algebra. Indeed, proposals for trace semantics for dataflow networks have appeared in the literature, like [Abr89, Mis90, Jon87]. In [Ong93] a notion of Testing is introduced that is shown to be equivalent to applicative bisimulation which is a variant of observational bisimulation defined by Milner [Mil89]. This notion of Testing takes the internal structure of process expressions into account. In our appoach we work with a notion of Testing defined by Hennessy [Hen88] that is only based on externally observable behaviour of processes, so a notion of Testing that does *not* take the internal structure of processes into account.

In this thesis we show how the key idea of representing a non-deterministic specification as a set of monotonic sequence-processing functions can be used for reasoning about progress properties of systems. We show the relation between Testing equivalence as defined by Hennessy and equality of specifications.

Our starting point is Broy's notion of specification. *A possible behaviour* of a system is modeled by a *monotonic sequence-processing function* which associates a sequence of output values to each sequence of possible input values. A non-deterministic specification is then characterized by *the set* of *all* its possible behaviours. Such a set is specified by a *functional specification*, i.e. a boolean function over monotonic sequence-processing functions. Functions therefore are central in our theory.

In Chapter 2 we introduce a notation for defining functions and we define the basic objects in our approach like messages and sequences.

In Chapter 3 we define the set of monotonic sequence processing functions. They serve as the semantical objects for modeling possible behaviours of systems. We show how non-determinism can be used to model system behaviour and we show how non-determinism can be represented by sets of functions. We define a set of combinators on monotonic sequence-processing functions and extend these definitions to specifications. These combinators resemble those typical of algebras for finite processes and give rise to whatwe

call Functional Algebra.

In Chapter 4 both an informal and a formal introduction to testing theory is given. This is essentially the theory of Hennessy for finite processes, when actions are interpreted as input/output pairs.

In Chapter 5 we present the main results of this thesis. We show that the traditional notion of testing equivalence, developed by Hennessy, does not exactly correspond to equality of sets of functions in the Functional Algebra. We show that this is due to the fact that in the model of Hennessy no distinction is made between input and output actions and that output actions can be influenced by the environment by means of external non-determinism. We show that a slightly modified model of testing, in which the experimenter has no influence on the output of a specification other than by means of supplying input, *does* correspond to equality of specifications in the Functional Algebra. The modified model of testing is shown to be slightly weaker than the traditional notion of testing.

Chapter 2

Notation and General Definitions

In this chapter we introduce some general concepts and notation. As we said in Chapter 1 we specify systems by sets of functions. So, central in this work is a notation for defining functions. Section 2.1 introduces the functional notation Funmath [Bou93] whereas in Section 2.2 the definitions of the basic objects of our approach like messages and sequences are given.

2.1 Notation

Funmath (*Fun*ctional *Math*ematics) [Bou93] is a formalism that is based mainly on the mathematical concepts of function and sets and on predicate calculus allowing for a transformational proof style. From the history of applied mathematics we know that functions and set theory are basic concepts with which there is a lot of experience in modeling. The motivation and topic of research for Funmath is to try to find a deep embedding of many useful, but isolated, theories for describing systems into a functional framework. Moreover, in doing this, the relation to traditional mathematical notation is maintained as close as possible. An example is the embedding of temporal logic in a functional framework [vT94]. In the following we give a short informal introduction to those features of Funmath which we use in the sequel. A detailed description of the notation can be found in [vT94, Bou93].

- Identifiers

  An identifier is a string of characters or symbols denoting the name of an object. There exist two kinds of identifiers namely constants and variables. Constants can be primitive (predefined in Funmath) or new

(defined by the user). Examples of primitive constants that are used in this work are the name of a basic type $\mathbb{B}$ denoting the Boolean values $\{0, 1\}$, and operators (functions) like $\wedge$, $\vee$, $\backslash$, etc.. In the thesis we will also use the predefined type $\mathcal{F}$ denoting all functions, $\mathcal{U}$ which stands for the universe, and $\mathcal{T}$ denoting all possible types. In the sequel we will denote the boolean value 0 also by *false* and 1 by *true*. New constants can be introduced using a definition of the form:

def $a : A$ **with** P

In this definition a denotes a single identifier or a tuple of identifiers, A is an expression denoting a type, and P is a defining proposition. For example:

def $double : \mathbb{N} \to \mathbb{N}$
with $double\ n = n + n$

The existence and uniqueness of the newly defined constant must be proven by the definer. Recursive definitions are allowed. In the context of this work they are intended in the fixpoint/domain-theoretic interpretation. For variables see abstraction below.

- Function application

  Function application denotes an object as the image of an object under a function. In prefix notation $f\ x$ denotes the image of x under f. In case of higher order functions the convention is that $f\ a\ b$ stands for $(f\ a)\ b$. Partial application allows for functions returning functions as result. This is possible also for infix operators (sometimes called *sectioning*).

- Tuple denotation. A tuple denotes a function with domain $\{0, .., n-1\}$ for n in $\mathbb{N}$.

  For example: a, b, c denotes the function $(a, b, c) \in \{0, 1, 2\} \to \{a, b, c\}$ such that $(a, b, c)0 = a$, $(a, b, c)1 = b$, and $(a, b, c)2 = c$.

  Notice that in the above formula $\{0, 1, 2\}$ denotes the range of the function $0, 1, 2$. That is, curly brackets together, $\{\}$, denote the operator *range* on functions. Normal brackets "(" and ")" are used for grouping in case of ambiguity.

The tuple consisting only of the element a is denoted by τa. Considering tuples as functions is extremely useful in transformational reasoning. A small disadvantage is however that in this way we lost the traditional notation for the one-element set is lost. We will write ιa when we want to denote the one-element set containing only a.

- Abstraction

  The only kind of abstraction we use in this thesis is of the form $x : X.E$.

  In this notation x is a single identifier or a tuple of identifiers, X is an expression denoting a type. The semantics of this formula is that it denotes the function mapping x (in X) to E. For shortness we sometimes leave out the type information X in formulas if no confusion can arise.

We define a number of basic functions which are used in the sequel.

Definition 2.1.1 *Universal quantification*

def $\forall : Fcod\ \mathbb{B} \to \mathbb{B}$
with $(\forall f) \equiv 0 \notin \{f\}$

□

Here $Fcod\ \mathbb{B}$ denotes the set of functions with codomain $\mathbb{B}$. In a similar way existential quantification can be defined as a function:

Definition 2.1.2 *Existential quantification*

def $\exists : Fcod\ \mathbb{B} \to \mathbb{B}$
with $(\exists f) \equiv 1 \in \{f\}$

□

The fact that we defined quantifiers as functions has no impact on the readability of formulas. The way they are defined coincides essentially with the common mathematical notation, and also the mathematical interpretation of those formulas remains the same. For instance, the mathematical formula $\forall x \in X.Px$ in Funmath is to be written as $\forall(x : X.Px)$ where $\forall$ is applied to the abstraction $x : X.Px$. The advantage however becomes clear when manipulating formulas in a transformational proof style.

In a similar way we can define the union over a set of sets, i.e. the set extension of the union. $\bigcup(x : X \;.\; f\ x)$ is the Funmath expression for the more common

$$\bigcup_{x \in X} \{f\ x\}$$

An important operator we use is equality on functions which is defined as:

Definition 2.1.3 *Function equality*

$$\begin{array}{l}\textbf{def } \_ = \_ : \mathcal{F} \to \mathcal{F} \to \mathbb{B} \\ \textbf{with } f = g \equiv (\mathcal{D}g = \mathcal{D}f) \wedge \forall(\sigma : \mathcal{D}f . f\sigma = g\sigma)\end{array}$$

□

In this definition $\mathcal{D}$ denotes the domain of a function, the set $\mathcal{F}$ here denotes the set of all functions.

The powerset of a set X is denoted by $\mathcal{P}\ X$. It takes a set and gives the set of all possible subsets of this set.

The function *if c then a else b fi* denotes the conditional construct that gives a if condition c is true, and gives b otherwise.

$$\begin{array}{l}\textbf{def } \mathit{if} \_ \mathit{then} \_ \mathit{else} \_ \mathit{fi} : \mathbb{B} \to \mathcal{T} \to \mathcal{T} \\ \textbf{with } \mathit{if}\ c\ \mathit{then}\ a\ \mathit{else}\ b\ \mathit{fi} = (b, a)\ c\end{array}$$

Recall that tuples are functions and that $\mathbb{B}$ is the set $\{0, 1\}$. So, $(b, a)\ 0 = b$ and $(b, a)\ 1 = a$. This example shows why it is convenient, from a orthogonality point of view, to define boolean values as $\{0, 1\}$. The *if then else fi* function is mainly meant to make definitions look more familiar then the unsugared form which is used to define this conditional construct.

The function $\square$ takes a natural number n and gives the subset of natural numbers ranging from 0 to $n - 1$. So, for example $\square 4 = \{0, 1, 2, 3\}$.

The function $\rceil$ restricts the domain of a function. In its type a dependent type construction is used. With this construction we express that the type of the second argument depends on the type of its first argument. Infact, the type of the second argument is a subset of the domain of the first argument.

$$\begin{array}{l}\textbf{def } \_\rceil\_ : \mathcal{F} \cdot \quad \cdot \mathcal{P}(\mathcal{D}\ f) \to \mathcal{F} \\ \textbf{with } g\rceil_X = h \\ \qquad\qquad \textbf{where } h : X \to \mathcal{C}\ g \\ \qquad\qquad\qquad h\ x = g\ x\end{array}$$

In the type of] a dependent type construction is used. With this construction we express that the type of the second argument of] depends on the type of its first argument. Infact, the type of the second argument is a subset of the domain of the function given as first argument. For a more formal treatment of dependent types in Funmath we refer to [vT94].

In order to avoid explicit mentioning the type of frequently used general functions a notation for polymorphism is used. For example, the implicit polymorphic identity function is defined by:

$$\begin{array}{l}\textbf{poly } A : \mathcal{T} \textbf{ def } id : A \to A \\ \textbf{with } id\ x = x\end{array}$$

In the following we define a number of general operators for obtaining the the set of minimal or maximal elements of a set ordered by means of a partial order relation.

$$\begin{array}{l}\textbf{poly } A : \mathcal{U} \textbf{ def } min\_ \_ : (A \times A) \to \mathcal{P}\ A \to \mathcal{P}\ A \\ \textbf{with } min_r\ X = \{x : X \mid \forall (y : X \mid y\ r\ x \Rightarrow x = y)\}\end{array}$$

So, in this definition $r : A \times A$ denotes a partial order relation.
In a similar way we define also *max*:

$$\begin{array}{l}\textbf{poly } A : \mathcal{U} \textbf{ def } max\_ \_ : (A \times A) \to \mathcal{P}\ A \to \mathcal{P}\ A \\ \textbf{with } max_r\ X = \{x : X \mid \forall (y : X \mid x\ r\ y \Rightarrow x = y)\}\end{array}$$

2.2 Messages and Sequences

In this section we give definitions for all the basic objects we use in building our theory. The first thing we need is a set denoting messages, we call it M. We don't put any restriction on what a message looks like, but we just assume we have a countable set of them.

Given M we define sequences of messages over this set. M^* denotes all finite sequences (also called lists over M), M^∞ denotes all infinite sequences over M (also called streams over M), the set of both finite and infinite sequences ($M^\omega = M^* \cup M^\infty$). In the sequel we use M^ω, M^* and M^∞ in the traditional way. In [Bou93] it is shown that also sequences, $\_^\omega$, $\_^*$ and $\_^\infty$ can be defined as suitable functions.

On M^ω we define some useful operators. In the definitions below, # denotes the operator that gives the length of a list, and *if c then a else b fi* denotes the conditional expression, that gives a if c is *true* and b otherwise.

- Concatenation

$$\begin{array}{l}\textbf{def} \ - +\!\!+ - : M^\omega \times M^\omega \to M^\omega \\ \textbf{with} \ (x +\!\!+ y)\ k = \mathit{if}\ k < \#x\ \mathit{then}\ x\ k\ \mathit{else}\ y\ (k - \#x)\ \mathit{fi}\end{array}$$

- Prefix on sequences

$$\begin{array}{l}\textbf{def} \ - \succ - : M \times M^\omega \to M^\omega \\ \textbf{with} \ a \succ x = \tau\ a +\!\!+ x\end{array}$$

- Postfix on sequences

$$\begin{array}{l}\textbf{def} \ - \prec - : M^\omega \times M \to M^\omega \\ \textbf{with} \ x \prec a = x +\!\!+ \tau\ a\end{array}$$

Note that with the above definitions the axioms, that are normally used to define these operators, can be proven.

We need one more operation on sequences, namely prefix-ordering.

$$\begin{array}{l}\textbf{def} \ \_ \sqsubseteq \_ : M^\omega \times M^\omega \to \mathbb{B} \\ \textbf{with} \ x \sqsubseteq y = (x = \varepsilon) \vee \exists (z \in M^\omega . x +\!\!+ z = y)\end{array}$$

Chapter 3

Functional Algebra

In this chapter we define the set of functions which we shall use as semantical objects for modeling possible behaviours of systems. They are called *monotonic sequence processing functions* (*MSPF*). We show how non-determinism can be used to model the behaviour of certain systems and how non-determinism can be represented by sets of functions. A set of functions which models the behaviour of a system is called a *specification*. We denote sets of functions by predicates (i.e. boolean functions) on those functions. Each function in the set represents one possible behaviour of the system.

In the case that the behaviour of a system is modeled by only one function, the set of functions modeling the system, contains only one function. Specifications that contain only one function are called *deterministic specifications*, specifications containing more than one function are called *non-deterministic specifications*.

We see determinism and non-determinism as a properties of *specifications* rather than of systems. Non-determinism is used as a means to specify the behaviour of a system in an abstract way. This makes the often heard statement that non-determinism is a useless concept because non-deterministic systems do not exist irrelevant. Non-determinism is a useful concept that makes it possible to describe certain properties of (sub)systems in such a way that properties of these systems can be shown. For instance one can easily model the behaviour of a possibly unreliable communication medium (which for example may corrupt messages) by a non-deterministic model. Then such a model, together with a deterministic model of a communication protocol, can be used to show liveness and safety properties of the complete system (medium and protocol together).

In Section 3.1 we define the set of monotonic sequence processing functions and a graph representation of them.

In Section 3.2 a set of combinators on monotonic sequence processing

functions is introduced. This combinators are then extended to specifications in Section 3.3.

3.1 Functions modeling system behaviour

System behaviour is modeled by a set of functions. Each function is a function from sequences of input messages to sequences of output messages. In this context ε, the empty sequence, models that further communication is unspecified. We denote the set of all input and output messages by M, and we require that the functions are total on the set of all sequences over M.

The causal relation between input and output implies and is modeled by requiring *prefix monotonicity* for functions. So extension of the input sequence can only lead to extension of the output sequence. Non-monotonic functions would in fact model behaviours in which providing the system with "more input" might produce a change in the output already produced, like some output "to disappear", which goes against any notion of computability. Moreover, for similar reasons we forbid systems to produce "spontaneously" output when no input is provided, i.e. their output on ε must be ε. So in the sequel we will only speak about monotonic sequence processing functions.

For such functions we introduce the notion of derivative. The derivative of a *MSPF*-function is the extra output the function produces after having taken one more input message.

$$\begin{aligned}&\textbf{def } \partial : (M^\omega \to M^\omega) \to M^\omega \to M^\omega\\ &\textbf{with } \partial\, f\ (\sigma \prec m) = \ \sigma'\\ &\qquad\qquad\qquad\qquad \textbf{where } f\ (\sigma \prec m) = f\ \sigma +\!\!+ \sigma'\end{aligned}$$

For monotonic sequence processing functions we introduce a graph representation, which facilitates characterising a few more properties of them and which is useful in examples. We will use deterministic labeled trees. For example the following function:

$$\begin{aligned}&\textbf{def } f : M^\omega \to M^\omega\\ &\textbf{with } f\ \sigma = \ \mathit{if}\ \sigma\ 0 = a\\ &\qquad\qquad \mathit{then}\ \mathit{if}\ \sigma\ 1 = b\\ &\qquad\qquad\qquad \mathit{then}\ 1 \succ 2 \succ 3 \succ \varepsilon\\ &\qquad\qquad\qquad \mathit{else}\ \mathit{if}\ \sigma\ 1 = c\\ &\qquad\qquad\qquad\qquad \mathit{then}\ 1 \succ 3 \succ \varepsilon\\ &\qquad\qquad\qquad\qquad \mathit{else}\ 1 \succ \varepsilon\\ &\qquad\qquad\qquad\qquad \mathit{fi}\end{aligned}$$

fi
else ε
fi

can be represented by the tree in Fig. 3.1.

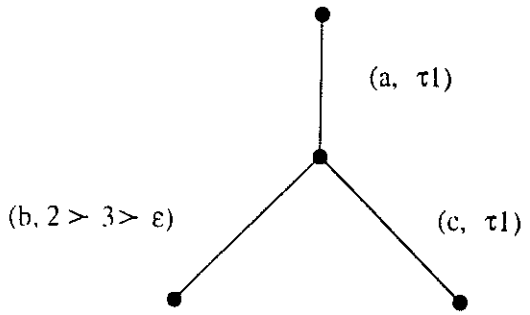

Figure 3.1: Tree of function f

A tree is deterministic if and only if all branches going out of each node are distinct. A deterministic tree is then fully characterised by its prefixed closed set of sequences of the labels at the arcs in the tree. In our case such sequences are the incremental traces of the behaviour of the function. The sequences are sequences of pairs of which the first element contains the increment in the input and the second element contains the increment in the output due to the increment in the input. So the output is the derivative of the function after incrementing the input σ with a certain message m. The tree-representation of a function can be constructed in the following way. Take all the infinite incremental traces of f. Exclude all sequences of which all outputs are ε. Shorten all other infinite sequences upto the point from which the outputs start being ε forever. As a last step include all the prefixes of the sequences that are left. Note that each monotonic sequence processing function is uniquely represented by its tree. This is due to the way this trees are constructed. This construction is formalized in the following definition of the function *tree* which gives a tree-representation of a monotonic sequence processing function.

We first define the set of infinite incremental sequences of a function. We call this set the *i/o-traces* of a monotonic function.

$$\textbf{def } iotr : (M^\omega \to M^\omega) \to \mathcal{P}\,(M \times M^\omega)^\infty$$
$$\textbf{with } iotr\ f = \{\ s : \mathcal{P}\,(M \times M^\omega)^\infty\ .$$
$$\exists(\sigma : M^\omega\ .\ \forall(i : \Box\infty\ .\ \ s\ i\ 0 = \sigma\ i \wedge$$
$$s\ i\ 1 = \partial f\ (\sigma\rceil_{\Box(i+1)})))\}$$

The following function takes the relevant (parts of) an infinite i/o-trace.

def $rel : (M \times M^\omega)^\infty \to (M \times M^\omega)^\omega$
with $rel\ \pi = \pi \rceil \sqsubset_m$
where $\iota\ m = min_{\leq}\{n : \mathbb{N}\ .\ \forall(j : \mathbb{N}\ .\ j \geq n \Rightarrow \pi\ \ j\ 1 = \varepsilon)\}$

The set of relevant i/o-traces can now be defined as:

def $reliotr : \mathcal{P}\ (M \times M^\omega)^\infty \to \mathcal{P}\ (M \times M^\omega)^\omega$
with $reliotr\ X = \{\pi\ .\ \exists(\tau : X\ .\ \pi = rel\ \tau)\}$

The prefix closure of a set of i/o-traces is obviously defined as:

def $prefclose : \mathcal{P}\ (M \times M^\omega)^\infty \to \mathcal{P}\ (M \times M^\omega)^\omega$
with $prefclose\ X = \{\pi\ .\ \exists(\tau : X\ .\ \pi \sqsubseteq \tau)\}$

The tree-representation of a monotonic function can now be defined as:

def $tree : (M^\omega \to M^\omega) \to DFT$
with $tree\ f = d$
where $L\ d = prefclose\ (reliotr\ (iotr\ f))$

In this definition $L\ d$ represents the i/o-traces of tree d. *DFT* stands for the set of all *Deterministic Functional Trees.* This is the set of all tree-representations of monotonic functions that give ε on ε. Note that monotonicity is essential for using this representation.

The formal definition of the function space *MSPF* follows:

Definition 3.1.1 *Function space (MSPF)*

def $MSPF : \mathcal{P}\ (M^\omega \to M^\omega)$
with $f \in MSPF$ $\quad f\ \varepsilon = \varepsilon$

f *is total*

$\forall(\sigma_1, \sigma_2 \in M^\omega\ .\ \sigma_1 \sqsubseteq \sigma_2 \Rightarrow f\ \sigma_1 \sqsubseteq f\ \sigma_2)$

$tree\ f$ *is finitely branching*

□

In the definition above we used the notion of finite branching. A tree $d \in DFT$ is finitely branching if and only if for all $s \in L\ d$ the set

$$\{(m, \sigma)\ .\ s \succ (m, \sigma) \in L\ d\}$$

is finite. We can now formally define what a *functional specification* is and when it is *consistent*:

Definition 3.1.2 *Specifications (SPEC)*
A specification is a boolean function over *MSPF*:

$$SPEC \equiv MSPF \rightarrow \mathbb{B}$$

□

Definition 3.1.3 *Consistency*
A specification $S : SPEC$ is consistent if and only if

$$\exists(f : MSPF\ .\ S\ \ f = true)$$

□

3.2 Function Combinators

In this section we define a constant function (terminator) and two higher order functions (combinators) which allow for building functions on the basis of their *sequential* behaviour and for composing them in a way which deals in a uniform way both with *external* and *internal non-determinism* [Hoa85]. The choice and definition of these functions has been inspired by analogous operators in process algebra.

Definition 3.2.1 *Terminator*

def $\mathcal{E} : MSPF$
with $\forall(\sigma\ .\ \mathcal{E}\ \sigma = \varepsilon)$

□

Obviously, $\mathcal{E}$ models the behaviour of not reacting to any stimulus.

Definition 3.2.2 *Input/output prefix on functions*

$$\begin{array}{l}\textbf{def}\ (—, —); — : M \to M^{\omega} \to \mathit{MSPF} \to \mathit{MSPF} \\ \textbf{with}\quad ((m,\sigma);f)\ \varepsilon = \varepsilon \\ \qquad\qquad ((m,\sigma);f)\ (k \succ \sigma') = \ \mathit{if}\ k \neq m \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{then}\ \varepsilon \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{else}\ \sigma\ +\!\!+ f\ \sigma' \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{fi}\end{array}$$

□

This is the analogous on functions of *action-prefix* in process algebras. Here an "action" consists in receiving a message m producing output σ. Thus, the first step of a (deterministic) system described by $(m,\sigma);f$ is receiving m as input and producing σ as output. After that, the system behaves like f. Notice that such a system does *not* react on sequences which do not start by m, thus giving ε on them.

The next higher order function we define on functions resembles the *choice* in process algebra. We want to define the choice in such a way that it has a number of properties. The first property is that we want it to be able to express both internal and external non-determinism. This means that if we have two functions, f and g, such that f initially can accept only input a, and g can accept only input b, the choice of this two functions must be able to apply f if it gets input a, and apply g if it gets b.

The second property follows from the fact that we want a choice that can be applied on any two *MSPF*-functions. Suppose both f and g can accept the same input, but give a different output. In this case the choice of two functions cannot be modeled by simply *one MSPF*-function. It would violate the basic property of functions, which is that for each input functions can produce at most one output. Of course we could require, as a restriction, that choice is only applied to functions in which this situation does not occur. But that is not the approach we want to take. Another solution, that is proposed in literature would be to generate a function that gives a set of possible outputs on one input. In our approach we prefer to generate a *set* of *MSPF*-functions.

The third property is inspired by our intention to define a notion of testing equivalence, in the setting of functions. In our setting the behaviour of a system is modeled by a set of *MSPF*. If a behaviour is modeled by a set containing more than one function, this means that we are in precense of internal non-determinism. This means that we assume that a certain "system run" will behave according to exactly *one* of the *MSPF*-functions in the set, but that we have no influence on which particular function it chooses. A

testing equivalence relation on specifications requires that we can compare the observable behaviour of specifications while considering them as black boxes. So the differences between the behaviour of specifications should become clear only by means of experiments that are performed on the specifications by interacting with them. These experiments are of the kind of giving one input message to the specification and observing the output of the specification. If the output is in agreement with what the experimenter was expecting, the test can be called successful. Of course the specification can be non-deterministic and this implies that the same test one time can be successful, but another time they do not. Given a certain specification, we can discriminate different kinds of experiments. Experiments that always succeed, experiments that never succeed, experiments that sometimes succeed and sometimes not when performed on the same specification. We want to model the behaviour of a system by its *maximal* set of *MSPF*. This is the set for which it is impossible to find a *MSPF*-function that can be added to the set without changing the observable behaviour the set models. So for which it is impossible to add a *MSPF*-function without changing the set of tests that must succeed, the set of those that never succeed and the set of those that both may succeed or may fail for this specification. In chapter 4 a formal introduction to the concept of testing is given. Here we restrict ourselves to an intuitive motivation for the kind of choice combinator we want.

A last obvious requirement is that the functions in the set generated by the choice on two functions should not accept (give) more than the composed functions themselves can accept (give).

In the following we show that a number of naive attempts to define the choice of functions leads to problems with one of the requirements we explained above.

From these requirements we know already the type that the choice-combinator on functions should have. It takes two *MSPF*-functions as arguments and gives a set of those functions as result. Sets of *MSPF* are specifications, and thus the type in the definition of choice is:

$$\textbf{def} \; — \; [] \; — : \mathit{MSPF} \nrightarrow \mathit{MSPF} \rightarrow \mathit{SPEC}$$

Now we describe three naive attempts to define choice. For each attempt we give an example to show which of the above requirements it doesn't meet.

First attempt:

The first atempt is to define the result of te choice of two functions as the set of this two functions.

$$\text{def} \; — \; [\!] \; — : MSPF \rightarrow MSPF \rightarrow SPEC$$
$$\text{with} \; (f_1 \; [\!] \; f_2) = \{f_1, f_2\}$$

The problem of this definition is that it cannot expresses external non-determinism.

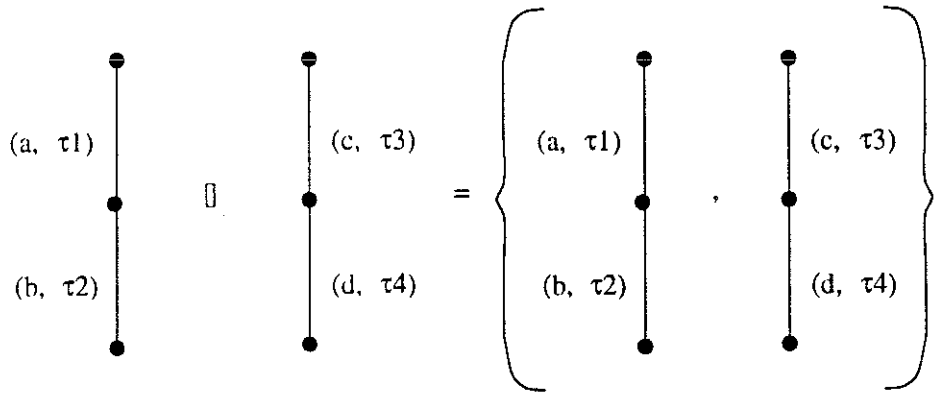

Figure 3.2: Choice of two functions

In Fig. 3.2 the choice of two functions is shown. One function can start accepting input a, the other input c. So we would expect that if we give the specification for example input a, it should always be able to accept it. However, the set of the two functions represents internal non-determinism. So the specification, as a result of the choice chooses, by itself, either to behave like one function or like the other function in the set. Suppose it chose to behave like the function that can only accept c as a first input. Then it refuses to accept input a. And thus the specification refuses to accept input a in this case. This example shows that the specification does not deal properly with external non-determinism.

Second attempt:

In the first attempt we would have liked to get the set shown in Fig. 3.3 as a result of the choice of the functions in Fig. 3.2. We can get this function by taking f_1 and extend it with the branches of f_2 which initially accept an input which is not equal to those f_1 accepts initially. And the same for f_2 which in this case leads to the same function.

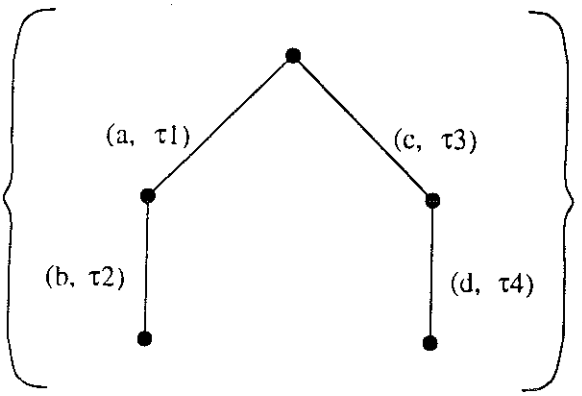

Figure 3.3: Result of choice in Fig. 3.2

In order to obtain this set we change the definition of choice of the first attempt into:

$$\textbf{def} \; — \; [\!] \; — : \mathit{MSPF} \rightarrow \mathit{MSPF} \rightarrow \mathit{SPEC}$$
$$\textbf{with} \; (f_1 \; [\!] \; f_2) = \{f \; . \; f = f_1 \; \mathit{extended\ to}\ f_2 \; \mathit{for\ different\ input} \vee$$
$$f = f_2 \; \mathit{extended\ to}\ f_1 \; \mathit{for\ different\ input}\}$$

However, with this definition we can show an example in which the requirement that the set of functions should be maximal is not met. Consider the choice of functions in Fig. 3.4. The specification as a result of the choice deals properly with external and internal non-determinism. But we could add function g shown in Fig. 3.5 to the set without being able to find a test that can discriminate the set $S_1 = \{f_1, f_2\}$ and the set $S_2 = \{f_1, f_2, g\}$. To see this we take a look at function g. The only thing this function can do more then f_1 and f_2 is that it can accept both b and d after $(a, \tau\ 1)$. However, we cannot find a test to find this difference if this function is together with f_1 and f_2. This is because when testing we can supply only one input at a time and see what happends. The reaction the specification S_2 gives on any test can always be explained as if there were only the two functions from S_1. For example if after $(a, \tau\ 1)$ action b would be accepted as input, it could have been because function g had been chosen, but also because of function f_1 had been chosen. Since we cannot find a test that shows any difference in behaviour specified by S_1, we can only conclude that the set with the two functions was not maximal.

Third attempt:

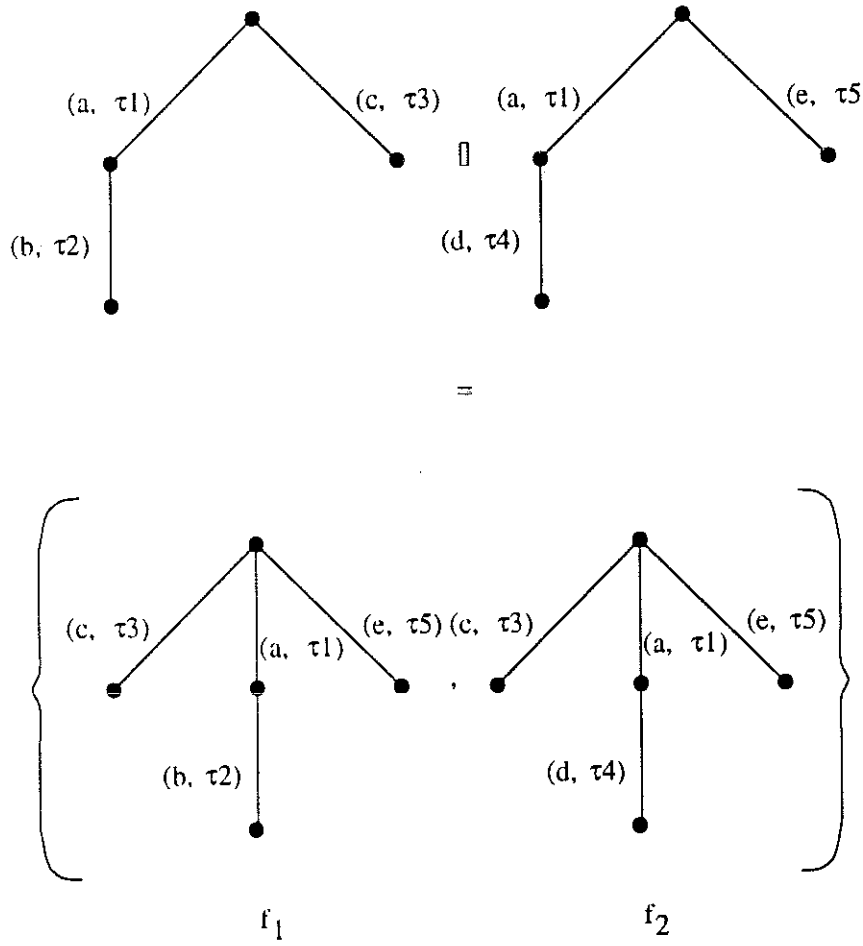

Figure 3.4: A non-maximal set of functions

In the third attempt we try to get directly the maximal set. We note that extending the functions with the branches of the other function which start with a different input was a good idea. So we keep that part. The problem was in the branches that start with an input both functions could accept. If we assume that output sequences on non-empty input are never ε these branches correspond exactly to the input sequences on which both composed functions can give an output which is not the empty sequence. On those input sequences we define f either as f_1 or as f_2 in order to get all possible combinations.

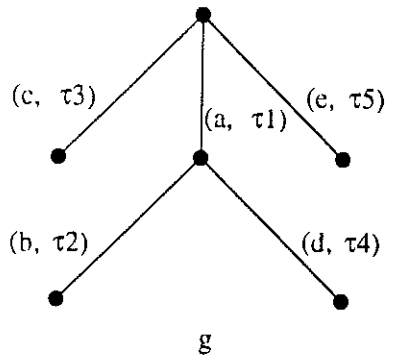

Figure 3.5: The extra function to be added to the ones in Fig. 3.4

$$\textbf{def} \; — \; [\!] \; — : \mathit{MSPF} \to \mathit{MSPF} \to \mathit{SPEC}$$
$$\textbf{with} \; (f_1 \; [\!] \; f_2) = \{f \; . \; \forall(\sigma \; . \; f_1 \; \sigma \neq \varepsilon \wedge f_2 \; \sigma = \varepsilon \Rightarrow f \; \sigma = f_1 \; \sigma \wedge$$
$$f_1 \; \sigma = \varepsilon \wedge f_2 \; \sigma \neq \varepsilon \Rightarrow f \; \sigma = f_2 \; \sigma \wedge$$
$$f_1 \; \sigma \neq \varepsilon \wedge f_2 \; \sigma \neq \varepsilon \Rightarrow f \; \sigma = f_1 \; \sigma \vee f_2 \; \sigma)\}$$

It is easy to see that function g satisfies this new definition. However, also this definition does not give the required result. Functions might get included that are not prefix-monotonic or that can do observably less then the original composed functions. Moreover we have to require that for every input message some non-empty output is produced. Fig. 3.6 shows two functions that, when composed by the choice defined above, give a set in which a non-monotonic function gets included. This function is defined on the sequences a and $a \succ b \succ \varepsilon$ as follows:

$$f \; (a \succ \varepsilon) = f_1 \; (a \succ \varepsilon) = \ldots 1$$
$$f \; (a \succ b \succ \varepsilon) = f_2 \; (a \ldots = 3 \succ 4 \succ \varepsilon$$

If we brutely require that ... functions, as a result of a choice, only contains monotonic functions ... functions that we don't want. For example the choice of the two ... in Fig. 3.4 results in a set containing a function, shown in Fig. 3.7. ... input a and output 1 cannot accept any input anymore. This is less ... composed functions can do, because those, after input a, either ... accept b or d.

From these naive attempts we can learn that we have to look for a better way to insert the missing functions in the incomplete set. For that we need a better understanding of what kind of functions exactly should be included.

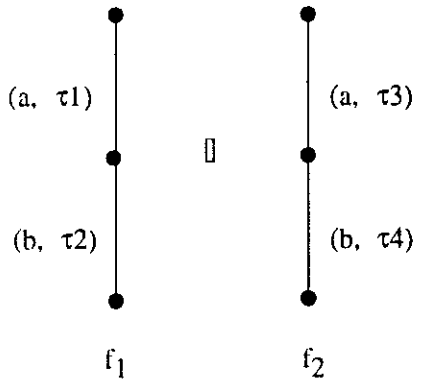

Figure 3.6: Inclusion of a non-monotonic function

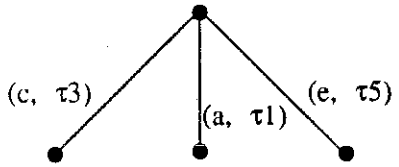

Figure 3.7: Function that stops after $(a, \tau\ 1)$

Let's first study which input/output-pairs the functions in the set resulting from a choice initially should be able to cope with. Suppose the set of initial input/output-pairs of a function f_1 is $\{(a, \tau\ 1), (b, \tau\ 2)\}$ and the initial set of f_2 is $\{(c, \tau\ 3)\}$. In order to model external non-determinism, all functions in the result of the choice of f_1 and f_2 should start with the set which is the union of the two sets. This has also been shown in the example of the second attempt above.

Now consider a different case. Suppose f_1 starts with pairs from the set $\{(a, \tau\ 1), (b, \tau\ 2)\}$ and f_2 starts with the set $\{(c, \tau\ 3), (b, \tau\ 5)\}$. If we take the union we get the set $X = \{(a, \tau\ 1), (b, \tau\ 2), (c, \tau\ 3), (b, \tau\ 5)\}$. This set contains two pairs that have input 5 but give different output. So this set cannot be the initial set a function can start with. We solve this problem by creating two functions instead of one. One function starts with the set $\{(a, \tau\ 1), (b, \tau\ 2), (c, \tau\ 3)\}$ the other with $\{(a, \tau\ 1), (b, \tau\ 5), (c, \tau\ 3)\}$. The output the specification will give on input b now depends only on the non-deterministic choice of the specification to behave either like one function or like the other. The two subsets we obtained from the set X are called the *maximal functional subsets* (*mfs*) of the set. A functional set is a set of

i/o-pairs in which each pair has a different input part. How to obtain such sets is formalized in Definition 3.2.4 on page 27.

In order to study the sets of input-output pairs at other points in the functions we take a look at the example in Fig. 3.4. Here the problem was that the two functions composed by choice can perform the same trace $(a, \tau\ 1)$ and after that behave different. It turned out that the function that was missing in the result was the one in which both $(b, \tau\ 2)$ and $(d, \tau\ 4)$ could be performed after $(a, \tau\ 1)$. So we should have created also the function with the union of the set of pairs of the individual functions after $(a, \tau\ 1)$. More complicated examples show that simply taking the union is not enough. For example, suppose the first function, after $(a, \tau\ 1)$ could perform the pairs in the set $\{(b, \tau\ 2), (c, \tau\ 3)\}$ instead of only $\{(b, \tau\ 2)\}$. Then we should have included the function that after $(a, \tau\ 1)$ can perform all pairs in the set $\{(b, \tau\ 2), (c, \tau\ 3), (d, \tau\ 4)\}$. But now, it is easy to see that the overall behaviour modeled by the specification would not change if we would add also functions which after $(a, \tau\ 1)$ can perform all pairs in $\{(b, \tau\ 2), (d, \tau\ 4)\}$ or in $\{(c, \tau\ 3), (d, \tau\ 4)\}$ or in $\{(b, \tau\ 2), (c, \tau\ 3)\}$. So, also all these functions should be included. So, in general, besides the sets characterizing the functions we started with, we should include also (the maximal functional susbsets of) their union and all intermediate sets.

The operator we are looking for, and which gives us exactly this kind of sets, is the *closure* operator which has been defined in the context of testing theory by Hennessy [Hen88]. There is only one problem left. This closure operator gives us not always exactly the sets we are looking for. We have already seen in the initial case, that taking the union of sets might result in obtaining sets that are not "functional", i.e. contain two or more pairs that have the same input but different output. The solution is to replace such sets by their maximal functional subsets, like we did in the initial case.

After this analysis we can really construct the set of functions we want to obtain from the choice of two functions. We use an idea that has been inspired by mathematical representations of behaviour developed for testing theory in process algebra. The key idea is to use the tree-representation we have defined to represent *MSPF*-functions.

Consider the choice of the functions in Fig. 3.8. These functions we can also represent by trees which are a little bit more sophisticated in the sense that we label the nodes of the trees by the sets containing the set of successors at that node. The set of successors is the set of i/o-pairs the function can perform at that node. We get the trees shown in Fig. 3.9. On this kind of trees we define a sum-combinator, which creates a *deterministic* tree with sets of successor sets at its nodes. The idea is to create this tree in such a way

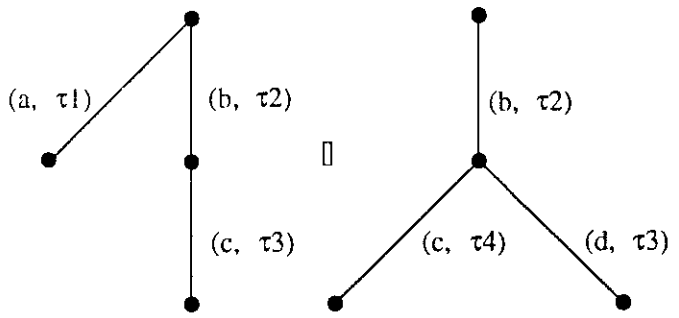

Figure 3.8: Two functions

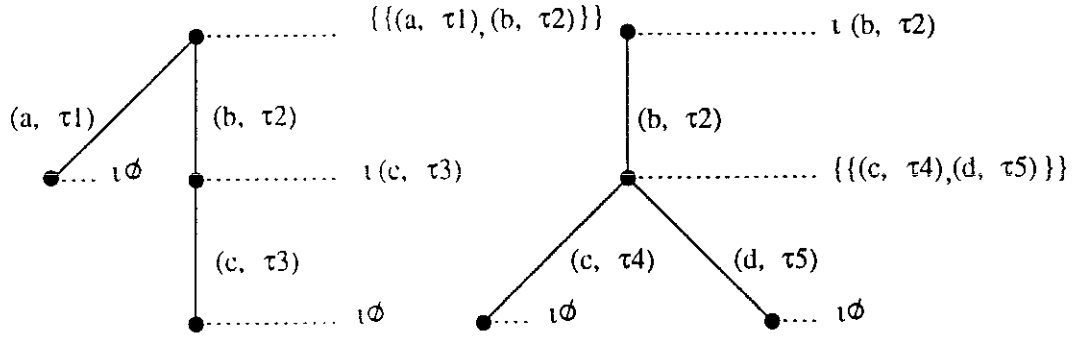

Figure 3.9: Enhanced trees of functions in Fig. 3.8

that it can be decomposed into a set of trees representing functions. This set contains exactly the set of functions we want to have. The sum-operator is a variant of the $+_{rd}$ operator of Hennessy [Hen88]. The set of paths in the intermediate tree is the union of the paths of the trees of the functions. The sets of successor sets at the nodes of the intermediate tree are obtained by an operator that is defined on successor sets that are found in the trees of the functions. This operator is a composition of the closure operator and an operator for obtaining maximal functional subsets. The closure operator is defined as follows

Definition 3.2.3 *Closure*

poly $X : \mathcal{U}$ **def** $c : \mathcal{P}\ \mathcal{P}\ X \to \mathcal{P}\ \mathcal{P}\ X$
with $c\ \Psi = Y$
where $Y = min_{\subseteq} K$
$K = \{Y\ .\ A \in Y \equiv\ A \in \Psi \vee$
$\exists((X_1, X_2) : Y\ .\ X_1 \cup X_2 = A) \vee$
$\exists((X_1, X_2) : Y\ .\ X_1 \subseteq A \subseteq X_2)\}$

□

The requirement $\exists((X_1, X_2) : Y\ .\ X_1 \cup X_2 = A)$ is also called $\cup$-closure and the requirement $\exists((X_1, X_2) : Y\ .\ X_1 \subseteq A \subseteq X_2)$ is called convex-closure. The operator *min* has been defined in Chapter 2.

For example the closure of the set of sets

$$\{\{(m, s_1)\}, \{(m, s_2), (n, s_1)\}\}$$

gives the set

$$\{\{(m, s_1)\}, \{(m, s_2), (n, s_1)\},$$
$$\{(m, s_1), (m, s_2)\}, \{(m, s_1), (n, s_1)\},$$
$$\{(m, s_1), (m, s_2), (n, s_1)\}\}$$

The second function is the function *mfs* which takes a completed set of successor sets and replaces each non-functional set by its maximal functional subsets.

Definition 3.2.4

def $mfs : \mathcal{P}\ \mathcal{P}\ (M \times M^*) \to \mathcal{P}\ \mathcal{P}\ (M \times M^*)$
with $mfs\ \Psi = \bigcup(X \in \Psi\ .\ smfs\ X)$

def $smfs : \mathcal{P}\ (M \times M^*) \to \mathcal{P}\ \mathcal{P}\ (M \times M^*)$
with $smfs\ X = max_{\subseteq}\{Y\ .\ Y \subseteq X \wedge func\ Y\}$

def $func : \mathcal{P}\ (M \times M^*) \to \mathbb{B}$
with $func\ X = \forall((x, y) : X^2\ .\ x \neq y \Rightarrow x\ 0 \neq y\ 0)$

□

The function *max* used above has been defined in Chapter 2.

The sum-combinator on the enhanced trees of the functions is defined by means of c and mfs. The enhanced trees, on which this sum-combinator is defined will be formally defined in Chapter 4 and we will call them *functional finite acceptance trees* ($ffAT$). The sum-combinator on this trees is therefore denoted by $+_{ffAT}$. In the definition of the sum-combinator below L_{ffAT} denotes the function that given a deterministic tree gives its set of paths. Function $\mathcal{A}_{ffAT}$ takes a deterministic tree t and a path s in the tree as arguments and gives the set of successor sets at the node in t reached by following s. $\mathcal{A}_{ffAT}\ t\ s$ is called the "acceptance set" of the node in tree t identified by s. Note that such a node is uniquely identified by s since t is deterministic. Function u denotes the pairwise union of two sets of sets. Its definition is:

Definition 3.2.5 *pairwise union*

$$\begin{aligned}&\textbf{poly } X : \mathcal{U} \textbf{ def } u : \mathcal{P}\,\mathcal{P}\,X \to \mathcal{P}\,\mathcal{P}\,X \to \mathcal{P}\,\mathcal{P}\,X\\ &\textbf{with } \Psi\ u\ \Phi = \{\, Y \mid \exists((A,B) \in \Psi \times \Phi \mid Y = A \cup B)\}\end{aligned}$$

□

The sum-combinator is defined as:

$$\begin{aligned}&\textbf{def } \text{—} +_{ffAT} \text{—} : ffAT \to ffAT \to ffAT\\ &\textbf{with } t_1 +_{ffAT} t_2 = t\\ &\textbf{where}\quad L_{ffAT}\ t = L_{ffAT}\ t_1 \cup L_{ffAT}\ t_2 \wedge\\ &\qquad\qquad \mathcal{A}_{ffAT}\ t\ \varepsilon = mfs\ (\mathcal{A}_{ffAT}\ t_1\ \varepsilon\ u\ \mathcal{A}_{ffAT}\ t_2\ \varepsilon)\\ &\qquad\qquad \mathcal{A}_{ffAT}\ t\ s = \textit{if } s \in L_{ffAT}\ t\\ &\qquad\qquad\qquad\qquad \textit{then } mfs\ (c\ (\mathcal{A}_{ffAT}\ t_1\ s \cup \mathcal{A}_{ffAT}\ t_2\ s))\\ &\qquad\qquad\qquad\qquad \textit{else } \emptyset\\ &\qquad\qquad\qquad\qquad \textit{fi}\end{aligned}$$

As an example the tree resulting from the sum-operation applied on the trees in Fig. 3.9 is shown in Fig. 3.10. The resulting tree has acceptance sets at its nodes. All elements in each set are functional. We can decompose such a tree into a set of trees which directly represent functions. The set of trees resulting from decomposing the tree in Fig. 3.10 is shown in Fig. 3.11. Each tree in the set is obtained by choosing one set in each acceptance set at the nodes of the tree in Fig. 3.10. From each collection we get in that way we can reconstruct a tree. So, for example, tree a) in Fig. 3.11 is the result of choosing set $\{(a, \tau\ 1), (b, \tau\ 2)\}$ in the set at the root of the tree in Fig. 3.10

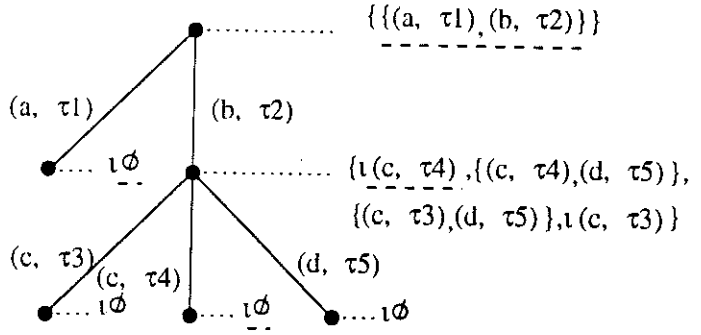

Figure 3.10: Resulting tree

and set $\{(c, \tau\ 4)\}$ at the node after the arc labeled with the element $(b, \tau\ 2)$ in the set we chose for the root. Then, after the arc labeled with $(c, \tau\ 4)$ we find in the tree in Fig. 3.10 the set containing only the empty set. So we choose that one.

The collection of sets that characterise tree a) in Fig. 3.11 is marked by underlining the relevant sets in Fig. 3.10. The decomposition-operator on trees is defined below. In this definition $S_{ffAT}\ d\ s$ gives the successor set of a deterministic tree d at the node which is uniquely identified by path s in d.

$$\textbf{def}\ dec : ffAT \to \mathcal{P}ffAT$$
$$\textbf{with}\ dec\ t = \{d\ .\ \forall(s :\ L_{ffAT}\ d\ .\ \ S_{ffAT}\ d\ s \in\ \mathcal{A}_{ffAT}\ t\ s)\}$$

The choice-combinator on *MSPF*-functions is now defined as:

Definition 3.2.6 *Choice on MSPF-functions* ($[\!]$)

$$\textbf{def}\ \text{—}\ [\!]\ \text{—} : MSPF \to MSPF \to SPEC$$
$$\textbf{with}\ (f_1\ [\!]\ f_2)\ f = tree\ f \doteq dec\ (tree\ f_1 +_{ffAT}\ tree\ f_2)$$

□

The following lemma guarantees consistence of choice:

Lemma 3.2.7 *Consistency of choice*

$$\forall(f_1, f_2 : MSPF\ .\ \exists(f : MSPF\ .\ (f_1\ [\!]\ f_2)\ f))$$

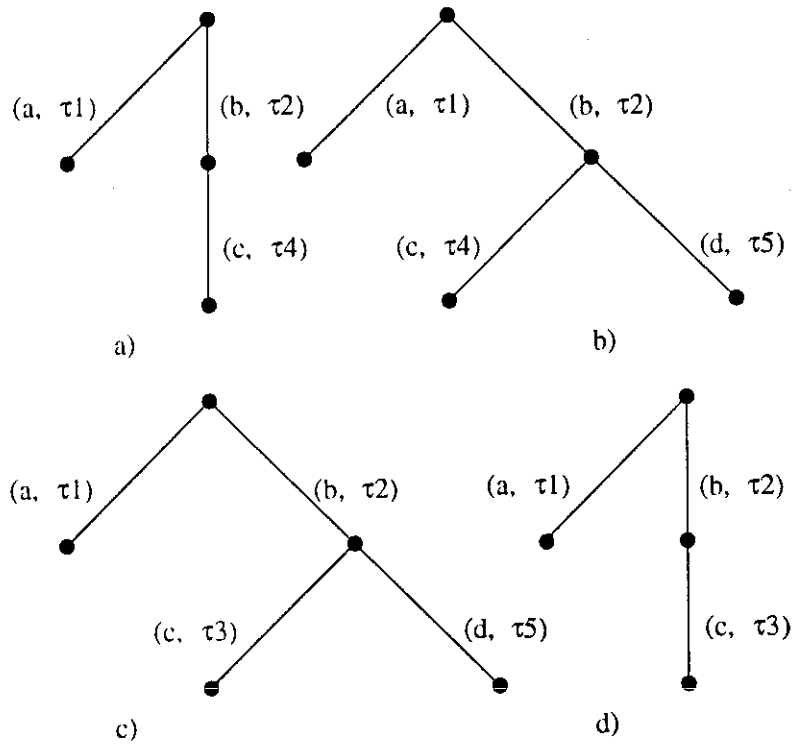

Figure 3.11: Set of trees resulting from decomposition of tree in Fig. 3.10

Proof
The assert trivially follows from the definitions of *tree*, $+_{gAT}$ and *dec*.

□

The following lemmata show that *MSPF* is closed under i/o-prefix and choice.

Lemma 3.2.8 *MSPF is closed under i/o-prefix*

$$\forall (f : MSPF, k \; M \; M^{\cdot} \; (k, \delta); f \in MSPF)$$

□

Lemma 3.2.9 *MSPF is closed under choice*

$$\forall (f_1, f_2, g : MSPF \; {}_{1}] \; f_2) \; g \Rightarrow g \in MSPF)$$

□

The proofs of these lemmas are omitted becasue they are trivial.

3.3 Combinators on Functional Specifications

So far we have defined a basic *MSPF*-function, and two combinators on *MSPF*. Our aim is, however, to define specifications, thus sets of *MSPF*, in a structured way. This means that we have to "lift" our combinators on functions to combinators on specifications. In [Bro92a, Bro92b] Broy proposes a general method for defining combinators on sets of functions based on combinators on functions. We will apply the same method to our combinators. The extension of combinators to specifications is straightforward. We call the resulting set of combinators our *Functional Algebra* (*FA*).

The basic function $\mathcal{E}$ can be used to define the basic specification **STOP**.

Definition 3.3.1 *Stop*

def STOP : *SPEC*
with STOP$f = (f = \mathcal{E})$

□

The input/output-prefix combinator on specifications is defined as:

Definition 3.3.2 *Input/output-prefix*

def $?\,\text{—}!\text{—};\text{—} : M \to (M^\omega) \to SPEC \to SPEC$
with $(?m!\sigma\,;S)\,f = \exists(g\,.\,S\;g \wedge f = (m,\sigma);g)$

□

Finally the choice-combinator on specifications as:

Definition 3.3.3 *Choice*

def $\text{—} \,[\!]\, \text{—} : SPEC \to SPEC \to SPEC$
with $(S_1 \,[\!]\, S_2)\,f = \exists(f_1, f_2 : MSPF\,.\,S_1 f_1 \wedge S_2 f_2 \wedge (f_1 \,[\!]\, f_2)\,f)$

□

Lemma 3.3.4 *Consistency of Specification Combinators*
All specifications built using only *STOP*, i/o-prefix on specifications and choice on specifications are consistent.

Proof
Trivial.

□

Note that the Functional Algebra constitutes the natural denotational model for the language defined by the following grammar:

$$S := \mathbf{STOP} \mid ?m!\sigma; S \mid S \,[]\, S$$

This will turn out to be useful in Chapter 5 where other operational and denotational semantics are defined for this language.

Chapter 4

Introduction to Testing Equivalence for Functional Specifications

In this chapter we introduce the notion of testing of processes as it is used in process algebra. It has been described in detail by Hennessy [Hen88] and in this introduction we will follow mainly his approach. The notion of testing is based on a model of the behaviour of systems. It considers systems to perform so-called *actions*, which can be observed by the environment in which the system is active. These actions, and in particular their relative ordering in time, represent the *behaviour* of such a system. The description of the actions and their order in time is called a *behaviour expression*. The model of the system, only representing its behaviour, is often called a *process*.

The framework Hennessy used to define a notion of testing equivalence for processes is mainly based on modeling behaviour by means of transition system oriented mathematical structures. In this chapter we want to investigate, however, if the same concept of testing equivalence can also be modeled in a framework based on monotonic stream processing functions. We like to do so because monotonic stream processing functions can serve as a functional variant for specification of system behaviour. The properties of functions and their composition are well-studied and known as a basis for powerful proof techniques that are suitable to be supported by automatized tools like theorem checkers and theorem provers. Testing equivalence has turned out to be an important equivalence in the field of process algebras because it takes progress properties of processes into account, and it is based on a widely accepted notion of "experimenting." It reflects the equivalence of processes which external behaviour cannot be distinguised by means of experiments. There exist other equivalences that take progress proper-

ties into account, like bisimulation [Mil89] (and applicative bisimulation in a functional framework [Ong93]), but they do not compare processes *only* on the basis of their external behaviour. For this reason these last equivalences quite often result to be "too strong."

To our knowledge however, testing equivalence is not yet defined in a functional framework in which specifications are sets of functions. In this field it is more common to work with the trace equivalence relation [Abr89] , which is known to be much weaker then testing equivalence, and which is not so much suited to deal with deadlock properties of processes.

When using functions as a model to describe the behaviour of systems, there is a natural inclination to divide the class of actions a system can perform into input actions and output actions. Such a strict division is however not necessary.

We will show how to extend the concept of action in such a way that an action deals both with input and output.

In Section 4.1 an informal and formal introduction to the theory of testing is given which resembles very much the introduction given by Hennessy [Hen88] except that we use actions that deal both with input and output. The notion of testing equivalence of processes is defined.

In Section 4.2 labeled transition systems are introduced and a relation on them is defined that corresponds to testing equivalence.

In Section 4.3 finite Acceptance Trees are introduced. It is explained that equality of finite Acceptance Trees correspond to testing equivalence. Moreover we show that under certain conditions finite Acceptance Trees can be used to represent sets of monotonic stream processing functions. We also show that under the same conditions equality of sets of monotonic stream processing functions, built from the functional algebra operators **STOP**, i/o-prefix and choice, corresponds to testing equivalence in which actions are represented as input/output pairs.

4.1 Concept of Testing

Once we are able to describe the behaviour of systems it follows almost naturally that we want to order these descriptions in certain classes. The criteria for classification depend on what aspects of systems we are interested in. One important aspect of the behaviour of systems is the behaviour that can be observed from outside the system, that is without taking into account the internal structure of the system. Two processes which are denoted by syntactically different behaviour expressions still may perform the same observable behaviour. For example if we have two expressions, say A and B,

which we combine in the description $A \parallel B$ meaning that the system behaves either as A or as B we could as well have written $B \parallel A$ to describe the same behaviour. So, syntactical different expressions may describe the same behaviour. What do we exactly mean by "the same behaviour ?" Summarizing Hennessy freely we could say : "Two descriptions are describing the same behaviour if no hypothetical user can ever distinguish between the behaviour of the two processes denoted by the two descriptions". We will formalize exactly this notion, sticking to our model of systems that accept input and produce output. This last assumption does not create any fundamental difference between the way testing equivalence is formalized by Hennessy and the way it is done in this chapter.

In the following we introduce a series of notions, mainly due to Hennessy, which will be used for the definition of testing equivalence. Each notion will be first introduced informally and then formalized.

Consider two processes, p_1 and p_2, and let us assume that they can produce non-deterministic behaviour. This means that if we introduce an experimenter it may find say p_1 reacting different everytime it tests p_1's behaviour with the same test.

One can imagine that this non-determinism of processes makes it quite complicated to test if there are differences between the behaviour of processes. But when we assume that we can test as often and as long as we want (at least for processes with finite behaviour) we can find three different "classes" of experiments, which can discriminate between non-deterministic behaviour of processes. These three classes are:

1. Experiments that *always* succeed
2. Experiments that *may succeed* and *may fail*
3. Experiments that *never* succeed

This leads to the following notions about the behaviour of a process p. We can say that p *may* satisfy an experiment t if t falls in class 1 or 2 above. We say that p *must* satisfy an experiment t if t falls in class 1. The "decision" if an test is successful or not is in this theory up to the experimenter.

In this framework we now can say that two processes are testing equivalent if and only if they *may* satisfy the same experiments and they *must* satisfy is the same experiments. So in short, we cannot find an experiment that can demonstrate a difference in their behaviour.

Example 4.1.1
Let us study the classical example of two testing equivalent processes.

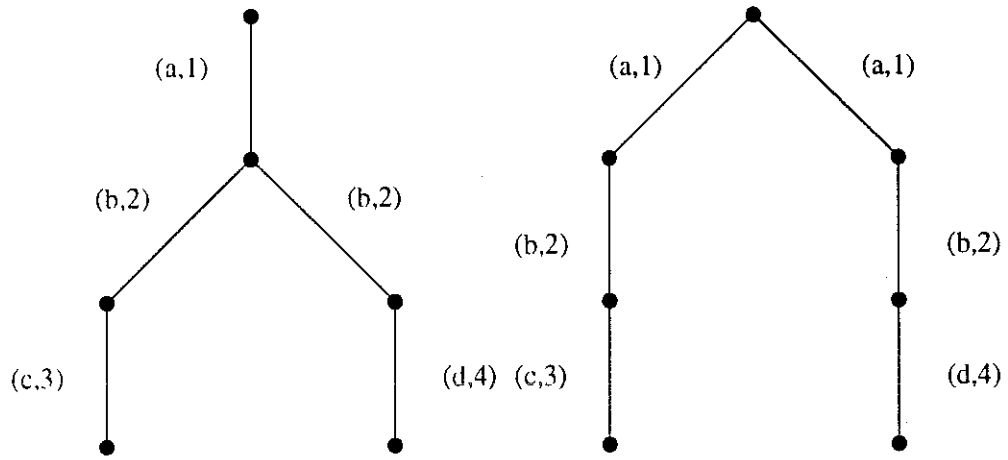

Figure 4.1: Two testing equivalent processes

The two pictures in Fig. 4.1 denote two processes with non-deterministic behaviour. The leftmost picture denotes a process that can take input a give output 1, and then non-deterministically decide to behave according to the leftmost or to the rightmost branch of the tree.

The rightmost picture in Fig. 4.1 denotes a process that already at the beginning decides either to behave like its leftmost or its rightmost branch. Suppose we put these processes into two different boxes, so that we cannot see the differences in the structure, but we can observe their behaviour. Then we will soon discover that our notions of *may* and *must* satisfy cannot show us any difference in behaviour between these processes. We can observe that these processes always must accept action a followed by b, producing as result 1 followed by 2. Also we can observe that the sequence $(a,1)(b,2)(c,3)$ sometimes can be observed, but sometimes not. For example, if p_2 chooses to behave like its right branch, it cannot accept a c and producing a 3 after input a followed by b. So for example, the test that reports success after the sequence $(a,1)(b,2)(c,3)$ has been observed, sometimes fails and sometimes is successful, for both processes! They both *may* satisfy this test, but, for both processes, it is certainly not the case that they *must* satisfy this test.

In order to formalize the concept of testing we introduce P, the set of processes, E the set of experimenters, $\|$ the interconnection between processes and experimenters determining interactions between them and M a set of messages of any kind. The processes from P we denote by p, the experimenters from E by e. For the time being we refrain from further details

about what exactly processes and experimenters are. These will be defined a little bit further in this section.

As we have seen the experimenter and the process have to interact in order to let a test take place. The combination of experimenter e and process p will be denoted by $e \parallel p$, giving a pair in $E \times P$. We define the interaction relation as a predicate on experimenters, processes and actions. Its type can be defined as:

$$\textbf{def} \; \_ - \_ \rightarrow_{\parallel} \_ : (E \times P) \rightarrow (M \times M^*) \rightarrow (E \times P) \rightarrow \mathbb{B}$$

So the expression $e \parallel p - (i, o) \rightarrow_{\parallel} e' \parallel p'$ means that if e wants to test if process p on input i gives output o and p indeed can accept i and produce o, they together can perform an interaction step, each reaching a new "state" denoted by e' and p' respectively while performing the same "action" (i, o).

A *test* now is defined as a whole series of these kind of interactions:

$$e_0 \parallel p_0 - (i_0, o_0) \rightarrow_{\parallel} \ldots - (i_{k-1}, o_{k-1}) \rightarrow_{\parallel} e_k \parallel p_k - (i_k, o_k) \rightarrow_{\parallel} \ldots$$

This notation is a shorthand for:

$$\begin{array}{l} e_0 \parallel p_0 - (i_0, o_0) \rightarrow_{\parallel} e_1 \parallel p_1 \\ \wedge \\ e_1 \parallel p_1 - (i_1, o_1) \rightarrow_{\parallel} e_2 \parallel p_2 \\ \wedge \ldots \end{array}$$

A *computation* is a test of maximal length. This can be in principle finite or infinite. Maximal means that no further interactions can take place, due to a mismatch between what the experimenter wants to test and what a process can perform. A computation is a *success* if the experimenter passes through a certain state, called the *success state*. The success states are a subset of the set E.

To denote the whole testing situation we use the notion of *experimental system* which includes all relevant information.

Definition 4.1.2 *Experimental system*
An experimental system $\mathcal{ES}$ is a four-tuple $\langle P, E, \_ \rightarrow_{\parallel}, Success \rangle$ where :

P is a set of processes

E is a set of experimenters

$\_ \rightarrow_{\parallel} : (E \times P) \rightarrow (M \times M^*) \rightarrow (E \times P) \rightarrow \mathbb{B}$ a predicate defining an interaction relation

$Success \subseteq E$ the set of success states

□

Let us also introduce the set of computations denoted by $Comp(e, p)$, containing all computations that start with $e \parallel p$. Now we can define the notions of *may* satisfy and *must* satisfy more formally.

Definition 4.1.3

$p\ may\ e = Comp(e, p)$ contains a successful computation

$p\ must\ e = Comp(e, p)$ contains *only* successful computations

□

Based on this definition we can define the following testing preorders on processes. A relation is called a preorder if it is reflexive and transitive.

Definition 4.1.4 *Testing preorders on processes*
Given an experimental system $\mathcal{ES} \equiv \langle P, E, - \rightarrow_{\parallel}, Success \rangle$ we can define the following preorders on processes $p, p' \in P$:

i) $p \sqsubseteq_{may} p' = \forall(e : E \,.\, p\ may\ e \Rightarrow p'\ may\ e)$

ii) $p \sqsubseteq_{must} p' = \forall(e : E \,.\, p\ must\ e \Rightarrow p'\ must\ e)$

iii) $p \sqsubseteq p' = p \sqsubseteq_{may} p' \wedge p \sqsubseteq_{must} p'$

□

Testing equivalence, denoted by $\sim$, can now be defined as:

Definition 4.1.5 *Testing equivalence*

def $\sim : P \rightarrow P \rightarrow \mathbb{B}$
with $p \sim p' = p \sqsubseteq p' \wedge p' \sqsubseteq p$

□

This definition now formalizes the notion that two processes are testing equivalent if no experiment can be found that discriminates their behaviour. What remains is to formalize the notion of a process, an experimenter and the interaction relation between them. We do this by means of giving an operational semantics of a process. This semantics can be modeled by Labeled Transition Systems (*LTS*).

Definition 4.1.6 *Labeled transition system*
A labeled transition system is a triple $\langle P, Act, - \rightarrow \rangle$ where

P is an arbitrary set of processes

Act is an arbitrary set of actions

$- \rightarrow : P \times Act \times P \rightarrow \mathbb{B}$ a predicate defining a transition relation

□

The expression $p - a \rightarrow p'$ may be read as : "p may perform action a and then evolve to the process p'." When several steps are made at once we denote this by the expression $p - s \rightarrow_* p'$, where s is a sequence of actions.

Definition 4.1.7 *Processes*
The finite processes we consider in this chapter can be denoted by terms generated by the following grammar[1]:

$$P ::= \mathbf{STOP} \mid ?m!\sigma; P \mid P \,[\!]\, P$$

PA_{io} will denote the set of such terms. In the above grammar m is an element from M, which denotes an arbitrary set of messages, and σ is a sequence of messages, so $\sigma \in M^*$.

The operational nature of a process denoted by a term from the set PA_{io} can be expressed by means of the following labeled transition system: $\langle PA_{io}, M \times M^*, - \rightarrow_{pr} \rangle$.

The predicate $- \rightarrow_{pr}$ defines the transitions that can take place within the labeled transition system. We define this predicate in a style introduced in [MR92]:

[1] We choose this notation here as a shorthand for the equivalent Funmath definition because although the notation is less precise from the type point of view, it is very familiar to most computer scientists. And at this place in the sequel we want to avoid to disrupt the line of the story with extra explanation about the somewhat different sytax that is used in the Funmath definition. The Funmath definition would look like

def $E : T_\Sigma$
with $E =$ **STOP** $\mid \langle\langle ; \mid (m, \sigma) : E \rangle\rangle \mid \langle\langle [\!] \mid E, E \rangle\rangle$

$$\begin{aligned}
&\textbf{def} \;\; - \rightarrow_{pr} : PA_{io} \times (M \times M^*) \times PA_{io} \rightarrow \mathbb{B} \\
&\textbf{with} \;\; \textbf{STOP} - (m,\sigma) \rightarrow_{pr} B \equiv \mathit{false} \\
&\qquad ?m!\sigma; B - (a,\gamma) \rightarrow_{pr} B' \equiv a = m \wedge \sigma = \gamma \wedge B = B' \\
&\qquad B_0 \,[\!]\, B_1 - (m,\sigma) \rightarrow_{pr} B' \equiv \; B_0 - (m,\sigma) \rightarrow_{pr} B' \\
&\qquad\qquad\qquad\qquad\qquad\qquad \vee \\
&\qquad\qquad\qquad\qquad\qquad\qquad B_1 - (m,\sigma) \rightarrow_{pr} B'
\end{aligned}$$

In these equations $m \in M$, $a \in M$, $\gamma \in M$ and $B, B' \in PA_{io}$.

□

The part of a labeled transition system that represents the behaviour of an expression p can be represented as a tree, where the labels at the nodes are behaviour expressions (i.e. expressions from PA_{io} in our case), the labels at the edges the "actions" (i.e. input-ouput pairs in our case) and the expression at the root is p. The labeled transition system part of some process p handling input and output then looks like the tree in Fig. 4.2. For easyness of writing we call the subpart of the *LTS* for a process p also directly "the *LTS* p."

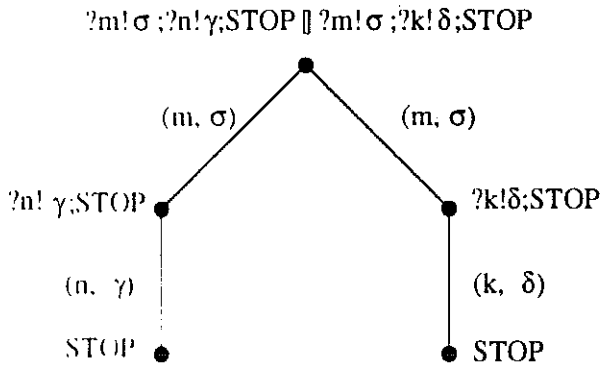

Figure 4.2: *LTS* handling input and output.

To simplify a bit we also use the form with only the action-labels. The expressions can be easily derived once we know what is the expression we started from. An example of such a picture starting from the same expression as in Fig. 4.2 is shown in Fig. 4.3.

Notice that the definition of processes gives an operational semantics to the language introduced in Chapter 3. In that chapter a denotational semnatics to the same language was given. *Experimenters* can be modeled in the same way as processes, but with some extra features. To give experimenters

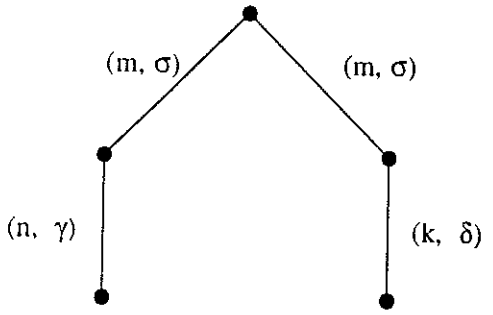

Figure 4.3: The *LTS* of Fig. 4.2 without behaviour expressions.

a bit more control over the process they are testing they are given the possibility to perform one special action independently from the process they are testing. Moreover they have a special action with which they can report success. The labeled transition system denoting the operational nature therefore looks very similar to that of processes.

Definition 4.1.8 *Experimenters*
An experimenter is a process denoted by any term generated by the following grammar:

$$E ::= \mathbf{STOP} \mid ?m!\sigma; E \mid E \parallel E$$

with $(m,\sigma) \in (M \times M^*) \cup \{(1,\tau\ 1),(w,\tau\ w)\}$. PE_{io} will denote the set of such terms. The operational semantics is denoted by the *LTS* $\langle PE_{io}, (M \times M^*) \cup \{(1,\tau\ 1),(w,\tau\ w)\}, -\rightarrow_{ex}\rangle$,
where $-\rightarrow_{ex} : PE_{io} \times ((M \times M^*) \cup \{(1,\tau\ 1),(w,\tau\ w)\}) \times PE_{io} \rightarrow \mathbb{B}$ is the predicate defining the transition relation by the following set of equations

$$\mathbf{STOP} - (m,\sigma) \rightarrow_{ex} E' \equiv false$$

$$?m!\sigma; E - (a,\gamma) \rightarrow_{ex} E' \equiv m = a \wedge \sigma = \gamma \wedge E = E'$$

$$E_0 \parallel E_1 - (m,\sigma) \rightarrow_{ex} E' \equiv \begin{array}{l} E_0 - (m,\sigma) \rightarrow_{ex} E' \\ \vee \\ E_1 - (m,\sigma) \rightarrow_{ex} E' \end{array}$$

□

As a shorthand we introduce the symbols **1** for $(1, \tau 1)$ and **W** for $(w, \tau w)$.

Having defined processes and experimenters we can formalize their interaction. A process and an experimenter that are compatible, i.e. can perform the same kind of actions, can form an "experimental system".

Definition 4.1.9 *Experimental system*
Let LTS_P and LTS_E be two compatible labeled transition systems $\langle P, Act, - \rightarrow_{pr} \rangle$ and $\langle E, Act \cup \{\mathbf{1}, \mathbf{W}\}, - \rightarrow_{ex} \rangle$ then $\mathcal{ES}\ (LTS_P, LTS_E)$ is the experimental system $\langle P, E, - \rightarrow_{\|}, Success \rangle$ where

- $- \rightarrow_{\|}$ is the predicate defining the interaction relation defined by the following interaction equation:

$$\begin{aligned} & e \parallel p - (m, \sigma) \rightarrow_{\|} e' \parallel p' \\ \equiv & \\ & (e - (m, \sigma) \rightarrow_{ex} e' \wedge p - (m, \sigma) \rightarrow_{pr} p') \vee \\ & (e - \mathbf{1} \rightarrow_{ex} e' \wedge p = p' \wedge (m, \sigma) = \mathbf{1}) \end{aligned}$$

- $Success = \{e : E \,.\, \exists (e' : E \,.\, e - \mathbf{W} \rightarrow_{ex} e')\}$

□

For reasoning about testing equivalence of processes denoted by the restrictive set of terms of PA_{io} we will consider the experimental system

$$\mathcal{ES}\ (\langle PA_{io}, M \times M^*, - \rightarrow_{pr} \rangle, \langle PE_{io}, (M \times M^*) \cup \{\mathbf{1}, \mathbf{W}\}, - \rightarrow_{ex} \rangle)$$

which is the four-tuple

$$\langle PA_{io}, PE_{io}, - \rightarrow_{\|}, Success \rangle$$

To illustrate the definitions we will work out a number of examples. For example the experimenter denoted by the expression $?m_1!\sigma_1; ?m_2!\sigma_2; \mathbf{W}; \mathbf{STOP}$ can be interpreted as an experimenter that tries to find out whether a process can perform the sequence $(m_1, \sigma_1) \succ (m_2, \sigma_2) \succ \varepsilon$. Which means that it tries to find out if a process accepts input m_1, and gives sequence σ_1 as output and then accepts input m_2 and gives output σ_2.

The experimenter $?m_1!\sigma_1; (?m_2!\sigma_2; \mathbf{W}; \mathbf{STOP} \,[\!]\, ?m_3!\sigma_3; \mathbf{W}; \mathbf{STOP})$ reports success if the process under test can either perform the sequence $(m_1, \sigma_1) \succ (m_2, \sigma_2) \succ \varepsilon$ or $(m_1, \sigma_1) \succ (m_3, \sigma_3) \succ \varepsilon$.

Let's consider a process p defined as

$$p = ?m_1!\sigma_1; ?m_2!\sigma_2; \mathbf{STOP}$$

and an experimenter e defined as:

$$e = ?m_1!\sigma_1; (\mathbf{1}; \mathbf{W}; \mathbf{STOP} \,[\!]\, ?m_2!\sigma_2; \mathbf{STOP})$$

and let's see which computations can be found in the experimental system. One computation is:

$$e \parallel p - (m_1, \sigma_1) \rightarrow_{\parallel} (\mathbf{1}; \mathbf{W}; \mathbf{STOP} \,[\!]\, ?m_2!\sigma_2; \mathbf{STOP}) \parallel ?m_2!\sigma_2; \mathbf{STOP}$$
$$- \mathbf{1} \rightarrow_{\parallel} \mathbf{W}; \mathbf{STOP} \parallel ?m_2!\sigma_2; \mathbf{STOP}$$

This computation is clearly a success, because the experimenter expression in the last step is $\mathbf{W}; \mathbf{STOP}$, and thus able to perform a $\mathbf{W}$-action.

Another computation starting from e and p is:

$$e \parallel p - (m_1, \sigma_1) \rightarrow_{\parallel} (\mathbf{1}; \mathbf{W}; \mathbf{STOP} \,[\!]\, ?m_2!\sigma_2; \mathbf{STOP}) \parallel ?m_2!\sigma_2; \mathbf{STOP}$$
$$- (m_2, \sigma_2) \rightarrow_{\parallel} \mathbf{STOP} \parallel \mathbf{STOP}$$

This computation is clearly *not* leading to success. Nowhere in the computation the experimenter is able to perform a $\mathbf{W}$-action as a first action. Note that now we can conclude p *must not* e, because not all computations starting from $e \parallel p$ lead to success. The process p', however, defined as $p' = ?m_1!\sigma_1; ?m_3!\sigma_3; \mathbf{STOP}$ *must* satisfy experimenter e.

So this test e may only fail in case it tests a process that is not able to perform (m_1, σ_1) as first "action", or after performing (m_1, σ_1) may perform (m_2, σ_2).

4.2 Testing Based on Labeled Transition Systems

In the previous section we formalized step by step the intuitive concept of what it means to test systems that can have non-deterministic behaviour. Also we defined in that case the notion of testing-equivalence. This formalization, that fits quite naturally the intuition we have of testing, turns out to be unpractical to use in actual testing.

To give an idea of the kind of reasoning one has to perform we work out an illustrative example taken from [Heu88], but modified a bit to fit to the input-output actions we work with.

Suppose we want to find out if the following holds:

$$(m_1, \sigma_1); (m_2, \sigma_2) \,[\!]\, (m_1, \sigma_1); (m_3, \sigma_3) \sqsubseteq_{must} (m_1, \sigma_1); ((m_2, \sigma_2) \,[\!]\, (m_3, \sigma_3))$$

For readability reasons we omitted the **STOP**-operator.

The "proof" could look like the following reasoning:
Suppose $(m_1, \sigma_1); (m_2, \sigma_2) \parallel (m_1, \sigma_1); (m_3, \sigma_3)$ *must* e , and e is some experimenter. The we have to show that $(m_1, \sigma_1); ((m_2, \sigma_2) \parallel (m_3, \sigma_3))$ *must* e.
We see several cases:

1. If e can immediately perform a **W**-action, then it is trivial.

2. If not, then suppose e could perform a series of **1**-actions and after that perform a **W**-action. This case is very similar to case one.

3. Suppose e could perform a series of **1**-actions followed by a (m_1, σ_1)-action. Then we have again several cases:

   (a) After the (m_1, σ_1) there are a number of **1**-actions followed by a **W**-action. This kind of experimenters *must* be satisfied also by the right hand side expression.

   (b) After (m_1, σ_1) there are a number of **1**-actions followed by (m_2, σ_2), ending up in say an expression e'. After e' a number of **1**-actions can follow and then a **W**-action. Indeed also this kind of experimenters *must* be satisfied by the right hand side expression.

   (c) The same as the previous point, only with (m_2, σ_2) replaced by (m_3, σ_3).

So indeed we can conclude that for all experimenters e such that the left hand side expression *must* e, also the right hand side expression *must* e.

It is probably clear now that for somewhat more extended examples this kind of reasoning runs out of hand. Therefore Hennessy proposed an alternative characterization of the pre-orders $\sqsubseteq_{may}$, $\sqsubseteq_{must}$ and $\sqsubseteq$, which is based on labeled transition systems and which makes reasoning about testing equivalence much simpler.

In order to reason about labeled transition systems it is useful to introduce the following operators on them.

Definition 4.2.1 *Operators on labeled transition systems*
For any labeled transition system$\langle P, Act, - \rightarrow \rangle$ with $p \in P$ and $s \in Act^*$ we can define:

- the language of P

$$\textbf{def } L : P \to \mathcal{P}\ Act^*$$
$$\textbf{with } L\ p = \{s : Act^* \ .\ \exists(p' : P \ .\ p - s \to_* p')\}$$

- the successors of p after s has been performed

$$\textbf{def } S : P \to Act^* \to \mathcal{P}\ Act$$
$$\textbf{with } S\ p\ s = \{a : Act \ .\ \exists((p', p'') : P^2 \ .\ p - s \to_* p' - a \to\ p'')\}$$

- the set of acceptances of p after s

$$\textbf{def } \mathcal{A} : P \to Act^* \to \mathcal{P}\ \mathcal{P}\ Act$$
$$\textbf{with } \mathcal{A}\ p\ s = \{S\ p'\ \varepsilon \ .\ \exists(p' : P \ .\ p - s \to_* p')\}$$

□

Sticking to the labeled transition system $\langle PA_{io}, M \times M^*, - \to_{pr} \rangle$ we can illustrate the above definitions by an example.

Suppose we define process p as:

$$(m_1, \sigma_1); ((m_2, \sigma_2) \parallel (m_3, \sigma_3)) \parallel (m_1, \sigma_1); (m_4, \sigma_4)$$

The corresponding labeled transition system presented in its simplified form, is given in Fig. 4.4.

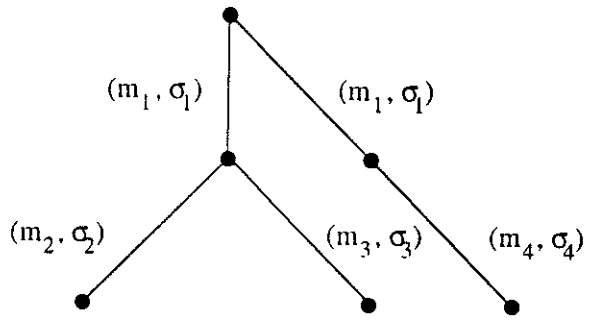

Figure 4.4: A labeled transition system

The language of this labeled transition system related to p is

$$L\ p = \{\varepsilon, (m_1, \sigma_1) \succ \varepsilon, (m_1, \sigma_1) \succ (m_2, \sigma_2) \succ \varepsilon,$$
$$(m_1, \sigma_1) \succ (m_3, \sigma_3) \succ \varepsilon, (m_1, \sigma_1) \succ (m_4, \sigma_4) \succ \varepsilon\}$$

Some examples of successor sets are:

$$S\ p\ \varepsilon = \iota\ (m_1, \sigma_1)$$

$$S\ p\ (m_1, \sigma_1) \succ \varepsilon = \{(m_2, \sigma_2), (m_3, \sigma_3), (m_4, \sigma_4)\}$$

Some examples of acceptance sets:

$$\mathcal{A}\ p\ \varepsilon = \iota\ \iota\ (m_1, \sigma_1)$$

$$\mathcal{A}\ p\ (m_1, \sigma_1) \succ \varepsilon = \{\{(m_2, \sigma_2), (m_3, \sigma_3)\}, \iota\ (m_4, \sigma_4)\}$$

We need one more non-standard operator on sets of sets before we can introduce an alternative definition of the preorders $\sqsubseteq_{may}$ and $\sqsubseteq_{must}$.

Definition 4.2.2

$$\textbf{poly}\ X : \mathcal{U}\ \textbf{def}\ \_ \subset\subset \_ : \mathcal{P}\ \mathcal{P}\ X \to \mathcal{P}\ \mathcal{P}\ X \to \mathbb{B}$$
$$\textbf{with}\ \Phi \subset\subset \Psi = \forall (A : \Phi\ .\ \exists (B : \Psi\ .\ B \subseteq A))$$

□

For example $\{\iota\ a, \{a, b\}\} \subset\subset \{\iota\ a, \iota\ b, \{a, b, c\}\}$ because $\iota\ a \subseteq \iota\ a$ and $\iota\ a \subseteq \{a, b\}$. Note that if $\Phi = \emptyset$ then trivially $\Phi \subset\subset \Psi$ for every Ψ.

The alternative definitions for the preorders can now be formulated as:

Definition 4.2.3 *Testing preorders for labeled transition systems*

$$\textbf{def}\ \ll_{may} : PA_{in} \to PA_{in} \to \mathbb{B}$$
$$\textbf{with}\ p \ll_{may} p' = L\ p \subseteq L\ p'$$

$$\textbf{def}\ \ll_{must} : PA_{in} \to PA_{in} \to \mathbb{B}$$
$$\textbf{with}\ p \ll_{must} p' = \forall (s : (M \times M^*)^*\ .\ \mathcal{A}\ p'\ s \subset\subset \mathcal{A}\ p\ s)$$

$$\textbf{def}\ \ll : PA_{in} \to PA_{in} \to \mathbb{B}$$
$$\textbf{with}\ p \ll p' = p \ll_{may} p' \wedge p \ll_{must} p'$$

□

A useful lemma related to these preorders is:

Lemma 4.2.4
For all p and p' in PA_{in}

$$\mathcal{A}\ p'\ s \subset\subset \mathcal{A}\ p\ s \Rightarrow L\ p' \subseteq L\ p$$

Proof
A proof of this lemma can be found in [Hen88].
□

Testing equivalence can now also be defined.

Definition 4.2.5 *Testing equivalence based on labeled transition systems*

$$\begin{aligned}&\textbf{def}\ \simeq : PA_{io} \to PA_{io} \to \mathbb{B}\\ &\textbf{with}\ p \simeq p' = p \ll p' \wedge p' \ll p\end{aligned}$$

□

Of course we have to show that the preorders on labeled transition systems, $\ll_{may}$ and $\ll_{must}$, correspond exactly to the preorders $\sqsubseteq_{\approx_{may}}$ and $\sqsubseteq_{\approx_{must}}$. We will restrict ourselves here to remark that the input-output tuples we have been using upto now can safely (trivially) be replaced by action names like a, b, c etc.. Different tuples getting different names. The whole formalization we did then boils down to exactly the theory and proofs Hennessy developed in his book [Hen88].

Since we did nothing really different from the theory developed in [Hen88] the following holds.

Lemma 4.2.6
For all p and p' in PA_{io} the following preorders are equivalent:

$$\begin{aligned}&p \sqsubseteq_{\approx_{may}} p' \equiv p \ll_{may} p'\\ &p \sqsubseteq_{\approx_{must}} p' \equiv p \ll_{must} p'\\ &p \sqsubseteq_{\approx} p' \equiv p \ll p'\end{aligned}$$

□

That the definitions based on labeled transition systems make it easier to reason about testing equivalence properties of processes is shown in the next example, which considers the same problem as in the example on page 43.

The problem was to find out if the following holds:

$$(m_1, \sigma_1); (m_2, \sigma_2) \parallel (m_1, \sigma_1); (m_3, \sigma_3) \sqsubseteq_{\approx_{must}} (m_1, \sigma_1); ((m_2, \sigma_2) \parallel (m_3, \sigma_3))$$

Now we can replace this by the following:

$$(m_1,\sigma_1);(m_2,\sigma_2) \parallel (m_1,\sigma_1);(m_3,\sigma_3) \ll_{must} (m_1,\sigma_1);((m_2,\sigma_2) \parallel (m_3,\sigma_3))$$

And according to the definition of $\ll_{must}$ the only thing we need to do is to check if for all $s \in \{(m_1,\sigma_1),(m_2,\sigma_2),(m_3,\sigma_3)\}^*$ it holds that $\mathcal{A}\ p'\ s \subset\subset \mathcal{A}\ p\ s$ if p represents the left hand side expression and p' the expression on the right hand side. We soon find out that we only need to check this for a few essential sequences because for sequences s that do not belong to $L\ p'$, $\mathcal{A}\ p'\ s$ will be the empty set, and in that case the property trivially holds.

So we check:

- $s = \varepsilon$

$$\begin{array}{l} \quad \mathcal{A}\ p'\ \varepsilon \\ = \\ \quad \iota\ \iota\ (m_1,\sigma_1) \\ \subset\subset \\ \quad \iota\ \iota\ (m_1,\sigma_1) \\ = \\ \quad \mathcal{A}\ p\ \varepsilon \end{array}$$

- $s = (m_1,\sigma_1)$

$$\begin{array}{l} \quad \mathcal{A}\ p'\ s \\ = \\ \quad \iota\ \{(m_2,\sigma_2),(m_3,\sigma_3)\} \\ \subset\subset \\ \quad \{\iota\ (m_2,\sigma_2), \iota\ (m_3,\sigma_3)\} \\ = \\ \quad \mathcal{A}\ p\ s \end{array}$$

- $s = (m_1,\sigma_1)\ (m_2,\sigma_2)$

$$\begin{array}{l} \quad \mathcal{A}\ p'\ s \\ = \\ \quad \iota\ \emptyset \\ \subset\subset \\ \quad \iota\ \emptyset \\ = \\ \quad \mathcal{A}\ p\ s \end{array}$$

- $s = (m_1, \sigma_1)\ (m_3, \sigma_3)$
  Same as previous.

For all other s, $\mathcal{A}\ p'\ s = \emptyset$. So we can conclude that $p \sqsubseteq_{must} p'$ in a much easier way.

4.3 Finite Acceptance Trees

Although the formalization of testing equivalence based on labeled transition systems makes it a lot easier to reason about testing equivalence of processes, it still uses a non-standard and not so straight forward operator $\subset\subset$.

Finite Acceptance Trees are introduced by Hennessy to give a different representation of processes such that the preorders $\sqsubseteq_{may}$ and $\sqsubseteq_{must}$ (or equivalently $\ll_{may}$ and $\ll_{must}$) can be defined in a more elegant way.

Finite Acceptance trees are deterministic, finitely branching, finite trees that represent the operational behaviour of a process. As an informal introduction we sketch the idea by means of an example.

Suppose we have a labeled transition system as in Fig. 4.5.

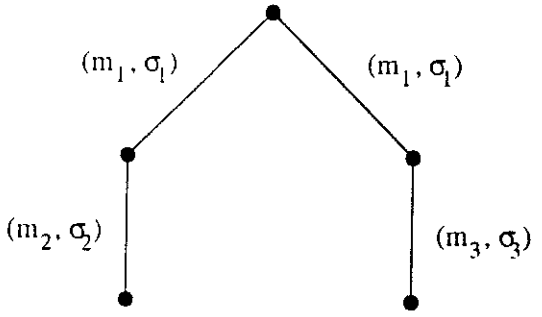

Figure 4.5: Example labeled transition system

The behaviour this labeled transition system represents can also be presented by a deterministic tree, which has as its labels at the nodes the set of sets of next actions which can be performed. We would get something like in Fig. 4.6.

The set $\{\iota\ (m_2, \sigma_2), \iota\ (m_3, \sigma_3)\}$ denotes that non-deterministically either (m_2, σ_2) is performed or (m_3, σ_3). This means that, independently of the environment, the process after having returned σ_1 on input m_1, will non-deterministically, either wait for m_2 as an input and produce σ_2, or wait for

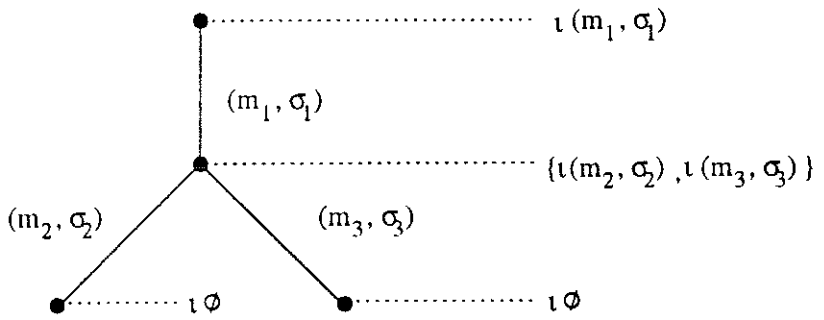

Figure 4.6: Finite acceptance tree representing the *LTS* in Fig. 4.5

m_3, producing σ 3. Now, if we look back to the labeled transition system in Fig. 4.5, we can verify that the labeled transition system in Fig. 4.3 is testing-equivalent to the first one. It is not only testing-equivalent, but also the most "complete" labeled transition system that is testing equivalent to the one in figure 4.5. Complete in the sense that no more branches can be added to the labeled transition system without introducing repetitions or loosing testing equivalence.

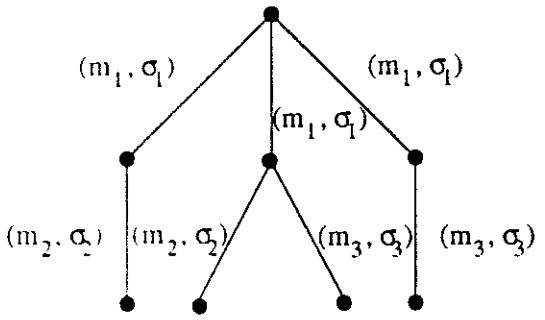

Figure 4.7: Completed version of *LTS* in Fig. 4.5

The representation of a completed tree as a deterministic tree with acceptance sets as nodes, is called a *finite Acceptance Tree* (*fAT*). A pictorial representation is given in Fig. 4.8.

It is also possible to derive directly from a, possibly not completed, *LTS* its corresponding finite Acceptance Tree representation. This can be done by means of a special "closure" operator on acceptance sets. Given a set of acceptances $\mathcal{A}$ *p after s* of a certain *LTS p* after sequence *s* has been performed, the closure operator applied to such a set adds exactly the sets that are missing

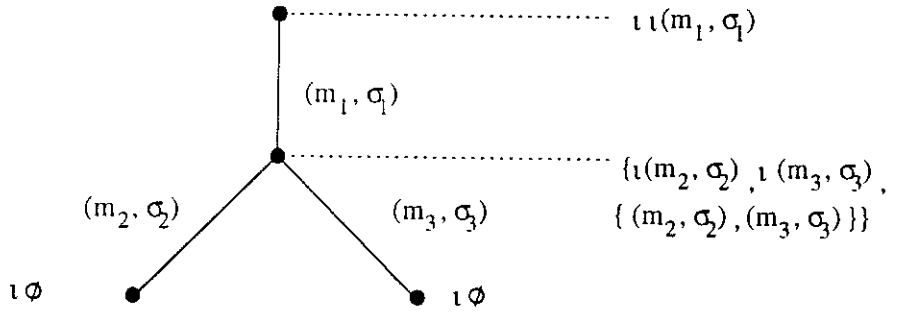

Figure 4.8: Finite acceptance tree of Fig. 4.3

when compared to the acceptance set we would have obtained from a completed version of p. This closure operator has been defined in Section 3.2.3 as:

$$
\begin{array}{l}
\textbf{def } c : \mathcal{P}\ \mathcal{P}\ X \to \mathcal{P}\ \mathcal{P}\ X \\
\textbf{with } c\ \Psi = Y \\
\textbf{where } \iota\ Y = min_{\subseteq} K \\
\qquad K = \{ Y\ .\ A \in Y \equiv\ A \in \Psi \vee \\
\qquad\qquad\qquad\qquad \exists((X_1, X_2) : Y\ .\ X_1 \cup X_2 = A) \vee \\
\qquad\qquad\qquad\qquad \exists((X_1, X_2) : Y\ .\ X_1 \subseteq A \subseteq X_2)\}
\end{array}
$$

We call a set of sets *closed* if the following holds:

Definition 4.3.1

$$
\begin{array}{l}
\textbf{def } closed : \mathcal{P}\ \mathcal{P}\ (M \times M^*) \to \mathbb{B} \\
\textbf{with } closed\ X = (c\ X = X)
\end{array}
$$

□

In order to define the set of finite acceptance trees it is useful to be able to speak about the collection of all the closed sets of subsets of a certain set R. We will call such a collection the subset related closed sets of a set R (*srcs* R). (In Hennessy this notion is called S-set but we chose a different name to avoid a too heavy overloading of the name S). The definition of *srcs* is:

$$
\begin{array}{l}
\textbf{poly } X : \mathcal{U}\ \textbf{def } srcs : \mathcal{P}\ X \to \mathcal{P}\ \mathcal{P}\ \mathcal{P}\ X \\
\textbf{with } srcs\ R = \{K : \mathcal{P}\ \mathcal{P}\ R\ .\ closed\ K \wedge R \in K\}
\end{array}
$$

In the following definition $L_{fAT}\ t$ denotes the language of a deterministic tree t. The language of a tree is its set of traces. $\mathcal{A}_{fAT}\ t\ s$ denotes the acceptance set at the node in t indicated by trace s in $L_{fAT}\ t$. $S_{fAT}\ t\ s$ denotes the set of labels of all the outgoing branches at the node indicated by trace s in the deterministic tree t. Note that in deterministic trees a trace in such a tree uniquely identifies a node. Finite acceptance trees can then be defined as:

Definition 4.3.2 *Finite acceptance trees*
A tree t is a Finite Acceptance Tree if it has a finite number of roots, if its branches are labeled by elements of $M \times M^*$, if its nodes are labeled by sets of subsets of $M \times M^*$, and if it satisfies the following three requirements:

- For every action (m, σ), every node in the tree t has at most one successor branch labeled by (m, σ)
- For every trace $s \in L_{fAT}\ t$, $S_{fAT}\ t\ s$, is finite (finitely branching)
- For each $s \in L_{fAT}\ t$, $\mathcal{A}_{fAT}\ t\ s$ is an element of $srcs\ (S_{fAT}\ t\ s)$

The set of Finite Acceptance trees over $M \times M^*$ is denoted by $fAT_{M \times M^*}$.

□

On finite Acceptance Trees we can define two basic operators $(,)_{fAT}$ and $+_{fAT}$ and we can define a basic tree called NIL_{fAT} that help in defining a translation from process expressions to finite Acceptance Trees.

Note that a finite Acceptance Tree is *deterministic* (different tuples are considered as different "actions"), *finitely branching* and *finite*. A finite acceptance tree is completely defined by its language and its acceptance sets at each node.

The basic acceptance tree and the two basic operators can be defined as:

Definition 4.3.3 *Finite acceptance tree operators*

def $NIL_{fAT} : fAT$
with $NIL_{fAT} = t$
where $L_{fAT}\ t = \iota\ \varepsilon \wedge$
$\mathcal{A}_{fAT}\ t\ s = \mathit{if}\ s \in L_{fAT}\ t\ \mathit{then}\ \iota\ \emptyset\ \mathit{else}\ \emptyset\ \mathit{fi}$

def $(—, —)_{fAT} : M \times M^* \to fAT \to fAT$
with $(m, \sigma)_{fAT}\ t =$
where $L_{fAT}\ t' = \iota\ \{(m, \sigma) \succ s\ .\ s \in L_{fAT}\ t\} \wedge$
$\mathcal{A}_{fAT}\ t'\ \varepsilon = \iota\ (m, \sigma) \wedge$
$\mathcal{A}_{fAT}\ t'\ ((n, \gamma) \succ s) = \mathit{if}\ (n, \gamma) \succ s \in L_{fAT}\ t'$

$$
\begin{array}{l}
\textit{then} \;\; \mathcal{A}_{fAT}\ t\ s \\
\textit{else}\ \emptyset \\
\textit{fi}
\end{array}
$$

$$
\begin{array}{lrl}
\multicolumn{3}{l}{\textbf{def} \ \text{—} +_{fAT} \text{—} : fAT \to fAT \to fAT} \\
\multicolumn{3}{l}{\textbf{with} \ t_1 +_{fAT} t_2 = t} \\
\textbf{where} & L_{fAT}\ t = & L_{fAT}\ t_1 \cup L_{fAT}\ t_2 \wedge \\
& \mathcal{A}_{fAT}\ t\ \varepsilon = & \mathcal{A}_{fAT}\ t_1\ \varepsilon\ u\ \mathcal{A}_{fAT}\ t_2\ \varepsilon \wedge \\
& \mathcal{A}_{fAT}\ t\ s = & \textit{if}\ s \in L_{fAT}\ t \\
& & \textit{then}\ c\ (\mathcal{A}_{fAT}\ t_1\ s \cup \mathcal{A}_{fAT}\ t_2\ s) \\
& & \textit{else}\ \emptyset \\
& & \textit{fi}
\end{array}
$$

□

In the last definition u denotes the pairwise union of two sets of sets. Its definition can be found in Chapter 3 on page 28.

There are a number of interesting and useful properties of c and also of c in combination with $\cup$ and u [Hen88]. Some of them, which we use in later sections, we list below. In this list the $\mathcal{B}$ are representing sets of sets.

c1. If $\mathcal{B} \subseteq \mathcal{B}'$ then $c\ \mathcal{B} \subseteq c\ \mathcal{B}'$

c2. $c\ \mathcal{B} \subseteq c\ (c\ \mathcal{B})$

c3. $c\ (\mathcal{B}_1 \cup \mathcal{B}_2) = c\ (c\ (\mathcal{B}_1) \cup c\ (\mathcal{B}_2))$

c4. If $A \in c\ \mathcal{B}$ then there is a $B \in \mathcal{B}$ such that $B \subseteq A$

c5. $\mathcal{B} \subseteq c\ \mathcal{B}$

c6. $c\ (\mathcal{B}_1 u\ \mathcal{B}_2) = c\ (\mathcal{B}_1)\ u$ [illegible]

When we have to prove [illegible] property holds for each element A in $c\ B$, for some B, we will use [illegible] duction to the length of the proof of "$A \in c\ B$". Of course this [illegible] shape of the definition of the closure operator c. As an illustration [illegible] proof technique we prove property c2.

To prove: $c\ \mathcal{B} \subseteq c\ (c\ \mathcal{B})$.
Which means: $\forall (X\ .\ X \in c\ \mathcal{B} \Rightarrow X \in c\ (c\ \mathcal{B}))$

There are three cases to be considered. The first one is the base case, wheras the second and third cases use the induction hypotesis (I.H.). Case

i)
The case the last rule used was that $X \in \mathcal{B}$.
To prove: $X \in \mathcal{B} \Rightarrow X \in c\ (c\ \mathcal{B})$
This is easy:

$$
\begin{array}{ll}
 & X \in \mathcal{B} \\
\Rightarrow & \quad \{\text{Definition of } c \text{ or rule c5}\} \\
 & X \in c\ \mathcal{B} \\
\Rightarrow & \quad \{\text{Definition of } c \text{ or rule c5}\} \\
 & X \in c\ (c\ \mathcal{B})
\end{array}
$$

Case ii)
This is the case that the last rule used was that there are X_1 and X_2 in $c\ \mathcal{B}$ such that $X = X_1 \cup X_2$. The length of the proof of each of the statements $X_1 \in c\ \mathcal{B}$, $X_2 \in c\ \mathcal{B}$ is less than that of $X \in c\ \mathcal{B}$. Then we get:

$$
\begin{array}{ll}
 & X \in c\ \mathcal{B} \\
\Rightarrow & \quad \{\text{Above assumption that } X \text{ in union closure}\} \\
 & \exists((X_1, X_2) : (c\ \mathcal{B})^2\ .\ X = X_1 \cup X_2) \\
\Rightarrow & \quad \{\text{I.H.}\} \\
 & \exists((X_1, X_2) : (c\ (c\ \mathcal{B}))^2\ .\ X = X_1 \cup X_2) \\
\Rightarrow & \quad \{\text{Definition of of } c, \text{ union-closure}\} \\
 & X \in c\ (c\ \mathcal{B})
\end{array}
$$

Case iii)
This is the case that the last rule used was that there are X_1 and X_2 in $c\ \mathcal{B}$ such that $X_1 \subseteq X \subseteq X_2$. The length of the proof of each of the statements $X_1 \in c\ \mathcal{B}$, $X_2 \in c\ \mathcal{B}$ is less than that of $X \in c\ \mathcal{B}$.
Then we get:

$$
\begin{array}{ll}
 & X \in c\ \mathcal{B} \\
\Rightarrow & \quad \{\text{Assumption of case iii)}\} \\
 & \exists((X_1, X_2) : (c\ \mathcal{B})^2\ .\ X_1 \subseteq X \subseteq X_2) \\
\Rightarrow & \quad \{\text{I.H.}\} \\
 & \exists((X_1, X_2) : (c\ (c\ \mathcal{B}))^2\ .\ X_1 \subseteq X \subseteq X_2) \\
\Rightarrow & \quad \{\text{Definition of } c, \text{ convex closure}\} \\
 & X \in c\ (c\ \mathcal{B})
\end{array}
$$

We can now map process algebra expressions onto finite acceptance tree representations by the transformation $PfAT$:

Definition 4.3.4

$$\begin{aligned}&\textbf{def } PfAT\ [\ ---\ ]: PA_{\iota o} \rightarrow fAT\\ &\textbf{with } PfAT\ [\textbf{STOP}] = NIL_{fAT}\\ &\qquad PfAT\ [?m!\sigma; p\ ] = (m,\sigma)_{fAT}\ PfAT\ [\ p\ ]\\ &\qquad PfAT\ [\ p_1\ []\ p_2] = PfAT\ [\ p_1] +_{fAT}\ PfAT\ [\ p_2]\end{aligned}$$

□

The link to testing-equivalence is now induced by a partial order relation on finite acceptance trees.

Definition 4.3.5

$$\begin{aligned}&\textbf{def } \leq_{fAT} : fAT \rightarrow fAT \rightarrow \mathbb{B}\\ &\textbf{with } t\ \leq_{fAT}\ t' = (\ L_{fAT}\ t = L_{fAT}\ t') \wedge\\ &\qquad\qquad \forall(s:\ L_{fAT}\ t'\ .\ \mathcal{A}_{fAT}\ t'\ s \subseteq \mathcal{A}_{fAT}\ t\ s)\end{aligned}$$

□

Lemma 4.3.6
The relation $\leq_{fAT}$ is a partial order.

Proof
Trivial.

□

If we compare this definition with definition 4.2.3 of a preorder on LTS we see that the $\subset\subset$ operator is substituted by a $\subseteq$ while working with closed acceptance sets. This is based on the following lemma:

Lemma 4.3.7
For all sets of sets A and B

$$actions\ A = actions\ B \Rightarrow B \subset\subset A \equiv c\ B \subseteq c\ A$$

Proof
The proof can be found in [Hen88].

□

The function *actions* is defined as:

$$\textbf{poly } X : \mathcal{U} \textbf{ def } actions : \mathcal{P}\,\mathcal{P}\,X \to \mathcal{P}\,X$$
$$\textbf{with } actions\ A = \{x\ .\ \exists(Y : A\ .\ x \in Y)\}$$

This leads to the following lemma which says that the partial order on finite acceptance trees is equivalent to the preorder we defined on labeled transition systems.

Lemma 4.3.8

$$\forall((p, q) : PA^2_{io}\ .\ p \ll q \equiv PfAT\ [\ p\ ]\ \leq_{fAT}\ PfAT\ [\ q\ ])$$

Proof
The proof of this lemma can be found in [Hen88].

□

Chapter 5

Testing Equivalence for Functional Specifications

In this chapter we develop functional finite Acceptance Trees. These are a modification of finite Acceptance Trees. They are constructed in such a way that they serve in a straightforward way as a representation for a *set of MSPF-functions*.

We show that, under a certain condition, the denotational model we defined in Chapter 3 for expressions built out of **STOP**, i/o-prefix and choice, is fully abstract with respect to testing equivalence as introduced in 4. A relation between a denotational model and a notion of behavioural equivalence is one of full abstraction if the denotational model and the behavioural equivalence induce exactly the same identifications between processes [Hen88].

The condition is that the finite Acceptance Tree of the processes denoted by the expressions contain only *functional acceptance sets*, i.e. each set in an acceptance set does not contain two actions that have the same input part. We call this restriction the *functionality restriction*. Informally this means that two algebraic expressions which are testing equivalent in the operational sense are represented by the same set of functions when these expressions are interpreted in the functional algebra and the other way around under the condition that the functionality restriction holds.

The main result in this chapter is that we can find a slightly different but very general notion of testing equivalence which exactly corresponds to equality of sets of *MSPF*-functions. This modified notion of testing is a slightly weaker than the testing equivalence defined by Hennessy, i.e. it identifies some more processes in specific cases.

As we will see in Section 5.3 this equivalence follows naturally from a slightly different notion of what it means to test systems that give output as a reaction to input.

In Section 5.1 we show how sets of *MSPF*-functions can be derived from *fAT*-representations under certain conditions. We introduce *functional fAT*s as a better representation from which sets of *MSPF* can be derived. We show that under the functionality restriction the preorder relation on *LTS*s corresponds to the partial order defined on *ffAT*.

In Section 5.2 we study the relation between equality of sets of *MSPF* and testing equivalence and we show why there is no exact correspondence.

In Section 5.3 we define a slightly modified notion of testing and testing equivalence and we show that the denotational model defined in Chapter 3 is fully abstract with respect to this modified notion of testing.

In Section 5.4 detailed transformational proofs can be found of all lemmas that are used in this chapter and are not proven in one of the previous sections.

5.1 Finite Acceptance Trees Representing Sets of Functions

In the previous chapter we gave a short introduction on testing theory, summarizing a part of the theory described by Hennessy. There was only one small syntactical difference, namely that we kept writing the actions as input-ouput pairs instead of representing different pairs by action names like a, b, c.

The reason for using pairs is that we want to use *fAT*s and a modification of them as structures from which we can derive sets of *MSPF*-functions. Since, in the remainder of this chapter, we will study only *finite* structures we consider only those *MSPF*-functions that can be represented by finite (functional) Acceptance Trees. Moreover we will also require the increase of output, as result of an input, of finite length.

To illustrate the idea we will first consider a few examples. Consider the simple and familiar *FIFO* in Fig. 5.1, and assume that different symbols such as m_i and σ_i represent different elements.

When we represent the *FIFO* in Fig. 5.1 as a finite acceptance tree we get a picture like in Fig. 5.2.

Note that in that representation all sets in every acceptance set are *functional*. This means that within these sets there is no occurence of two tuples that have the same input as first element. In such cases we can decompose the finite acceptance tree into a set of deterministic labeled transition systems by choosing everytime one set in each acceptance set for every labeled transition system. The result is presented in Fig. 5.3.

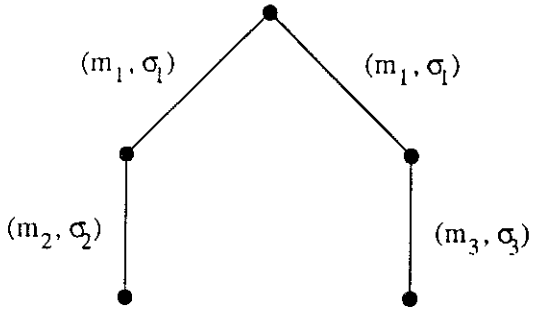

Figure 5.1: Example *LTS*

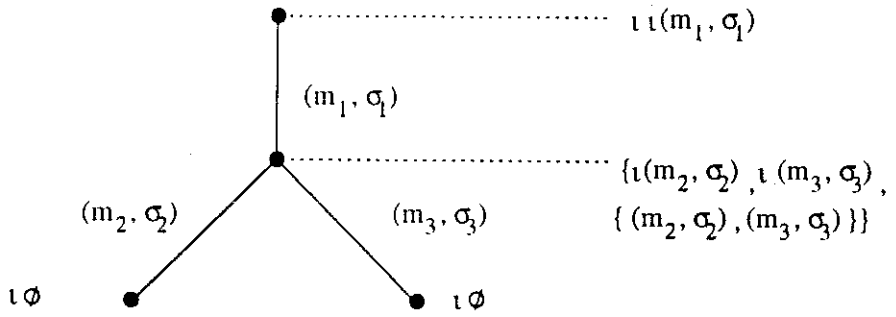

Figure 5.2: Finite acceptance tree belonging to *LTS* in Fig. 5.1

The kind of labeled transition systems we get are not only deterministic when we consider the complete input output tuples, but also deterministic when we consider only the input part of the tuples. This makes that we can find a bijection between this kind of trees and finite monotonic stream processing functions as they were defined in chapter 3. There we actually used this kind of deterministic *LTS*s as a convenient way to represent monotonic stream processing functions by means of a picture.

Of course not all labeled transition systems that we can describe by means of process expressions have acceptance sets that contain only functional sets. Consider for instance the example in Fig. 5.4 and its finite acceptance tree representation in Fig. 5.5.

The acceptance sets of the finite acceptance tree in the last figure do not only contain functional sets. However we could split the set $\{(m_1, \sigma'), (m_1, \sigma_1)\}$ into two sets $\iota\ (m_1, \sigma')$ and $\iota\ (m_1, \sigma_1)$ i.e. its maximal functional subsets, and we could simply replace the original set by these two sets. If we do this for all sets that are not functional, we end up with a set of

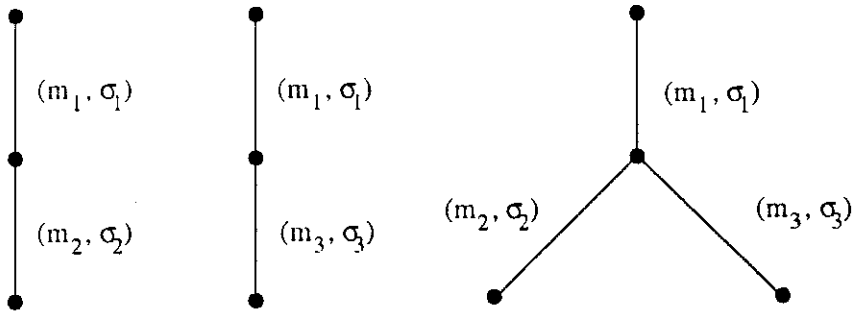

Figure 5.3: Functions from decomposing acceptance tree in Fig. 5.2

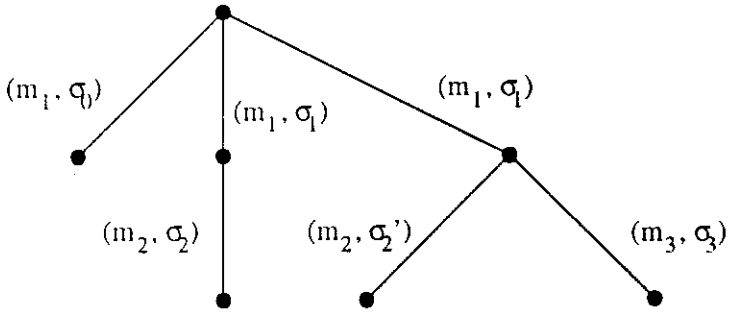

Figure 5.4: Example *LTS*

sets all of which are functional. The result of this can be found in Fig. 5.6.

This finite acceptance tree *can* be decomposed into a set of functions as presented in Fig. 5.7. This last kind of finite acceptance trees we will call *functional finite acceptance tree* (*ffAT*).

In order to check [illegible] this somewhat reorganised kind of finite acceptance trees we can [illegible] reason about testing equivalence of processes we formalize the repres[illegible] *ffAT*. We define a partial order on *ffAT*s and show that this part[illegible] order the functionality restriction, corresponds to the testing preord[illegible] *FTss*.

First we formalize [illegible] in which we split the non-functional sets into functional sets. This [illegible] we called maximal functional subset *mfs*. It takes a set of sets of pairs as its argument and results in a set of sets of pairs, but the sets of pairs are now all functional. It has already been defined in Chapter 3 on page 27. We repeat the definition here.

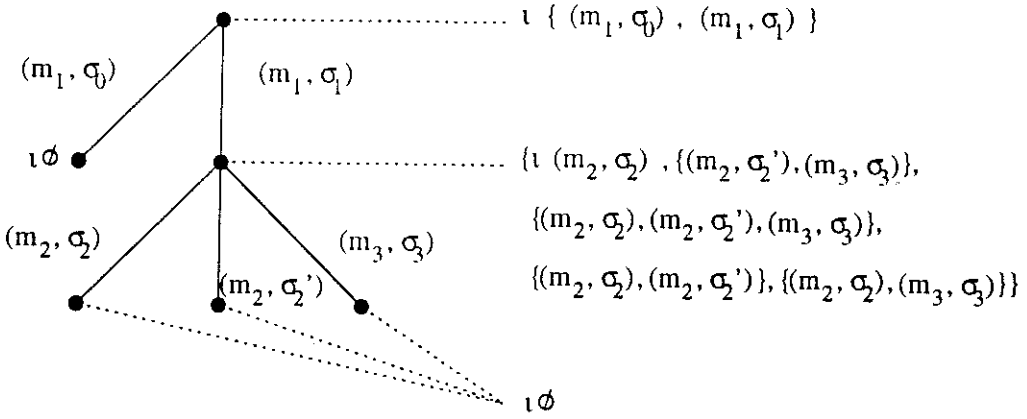

Figure 5.5: Resulting acceptance tree containing non-functional sets

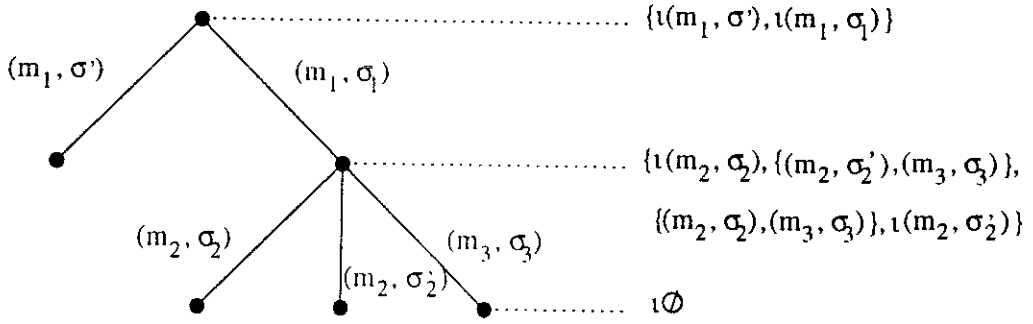

Figure 5.6: Functional finite acceptance tree

def $mfs : \mathcal{P}\,\mathcal{P}\,(M \times M^*) \rightarrow \mathcal{P}\,\mathcal{P}\,(M \times M^*)$
with $mfs\ \Psi = \bigcup(X : \Psi \mid smfs\ X)$

def $smfs : \mathcal{P}\,(M \times M^*) \rightarrow \mathcal{P}\,\mathcal{P}\,(M \times M^*)$
with $smfs\ X = max_{\subseteq}\{\ Y \mid Y \subseteq X \wedge func\ Y\}$

def $func : \mathcal{P}\,(M \times M^*) \rightarrow \mathbb{B}$
with $func\ X = \forall((x, y) : X^2 \mid x \neq y \Rightarrow x\ 0 \neq y\ 0)$

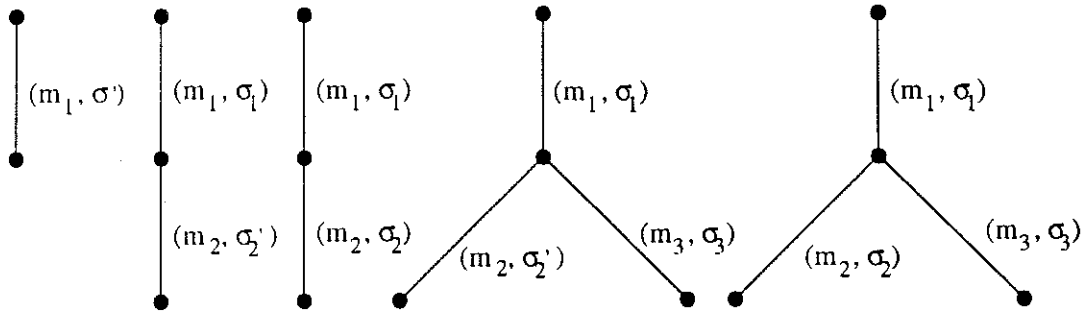

Figure 5.7: Resulting set of functions

We state two properties of *mfs* which we use later:

Lemma 5.1.1

1. *mfs* is monotonic
2. For all sets of sets $A, B \in \mathcal{P}\,\mathcal{P}\,(M \times M^*)$

$$mfs\ (A \cup B) = mfs\ A \cup mfs\ B$$

Proof
The proofs are straightforward and left to the reader.

□

Now we can define *ffATs*. In this definition $L_{ffAT}\ t$ stands for the language of the *ffAT*, $S_{ffAT}\ t\ s$ stands for the set of successors in t at the node identified by trace s. $\mathcal{A}_{ffAT}\ t\ s$ denotes the acceptance set in t at the node indicated by trace s.

Definition 5.1.2
A *functional finite Acceptance Tree* t is finitely rooted tree, with branches that are labeled by elements of $M \times M^*$, whith nodes that are labeled by sets of subsets of $M \times M^*$, and which satisfies the following three requirements:

- For every action (m, σ), every node in tree t has at most one successor branch labeled by (m, σ)
- For every trace $s \in L_{ffAT}\ t$, $S_{ffAT}\ t\ s$ is finite (finitely branching)
- For each $s \in L_{ffAT}\ t$ it holds that $\mathcal{A}_{ffAT}\ t\ s$ is the *mfs* of a set in *srcs* $(S\ t\ s)$.

We denote the set of finite functional Acceptance Trees over $M \times M^*$ by $\mathit{ffAT}_{M \times M^*}$.

□

Like with finite acceptance trees we can build the ffATs from a basic ffAT and two operators. For this we introduce the operator $\textcircled{u}$ which is the pairwise union on sets of sets, but which moreover replaces possible non-functional sets by their maximal functional subsets.

Definition 5.1.3

$$\begin{aligned}&\textbf{def } \textcircled{u}\colon \mathcal{P}\,\mathcal{P}\,(M \times M^*) \to \mathcal{P}\,\mathcal{P}\,(M \times M^*) \to \mathcal{P}\,\mathcal{P}\,(M \times M^*)\\ &\textbf{with } \Psi\ \textcircled{u}\ \Phi = \mathit{mfs}\ (\Psi\ u\ \Phi)\end{aligned}$$

□

Definition 5.1.4 *ffAT operators*

$$\begin{aligned}&\textbf{def } \mathit{NIL}_{\mathit{ffAT}} : \mathit{ffAT}\\ &\textbf{with } \mathit{NIL}_{\mathit{ffAT}} = t\\ &\textbf{where } \quad L_{\mathit{ffAT}}\ t = \iota\ \varepsilon \wedge\\ &\qquad\qquad \mathcal{A}_{\mathit{ffAT}}\ t\ s = \mathit{if}\ s \in L_{\mathit{ffAT}}\ t\ \mathit{then}\ \iota\ \emptyset\ \mathit{else}\ \emptyset\ \mathit{fi}\end{aligned}$$

$$\begin{aligned}&\textbf{def } (—,—)_{\mathit{ffAT}} : M \times M^* \to \mathit{ffAT} \to \mathit{ffAT}\\ &\textbf{with } (m,\sigma)_{\mathit{ffAT}}\ t = t'\\ &\textbf{where } \quad L_{\mathit{ffAT}}\ t' = \iota\ \varepsilon \cup \{(m,\sigma) \succ s\ .\ s \in L_{\mathit{ffAT}}\ t\} \wedge\\ &\qquad\qquad \mathcal{A}_{\mathit{ffAT}}\ t'\ \varepsilon = \iota\ \iota\ (m,\sigma) \wedge\\ &\qquad\qquad \mathcal{A}_{\mathit{ffAT}}\ t'\ ((n,\gamma) \succ s) = \mathit{if}\ (n,\gamma) \succ s \in L_{\mathit{ffAT}}\ t'\\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathit{then}\ \mathcal{A}_{\mathit{ffAT}}\ t\ s\\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathit{else}\ \emptyset\\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathit{fi}\end{aligned}$$

$$\begin{aligned}&\textbf{def } — +_{\mathit{ffAT}} — : \mathit{ffAT} \times \mathit{ffAT} \to \mathit{ffAT}\\ &\textbf{with } t_1 +_{\mathit{ffAT}} t_2 = t\\ &\textbf{where } \quad L_{\mathit{ffAT}}\ t = L_{\mathit{ffAT}}\ t_1 \cup L_{\mathit{ffAT}}\ t_2 \wedge\\ &\qquad\qquad \mathcal{A}_{\mathit{ffAT}}\ t\ \varepsilon = \mathcal{A}_{\mathit{ffAT}}\ t_1\ \varepsilon\ \textcircled{u}\ \mathcal{A}_{\mathit{ffAT}}\ t_2\ \varepsilon \wedge\\ &\qquad\qquad \mathcal{A}_{\mathit{ffAT}}\ t\ s = \mathit{if}\ s \in L_{\mathit{ffAT}}\ t\\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathit{then}\ \mathit{mfs}\ (c\ (\mathcal{A}_{\mathit{ffAT}}\ t_1\ s \cup \mathcal{A}_{\mathit{ffAT}}\ t_2\ s))\\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathit{else}\ \emptyset\\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathit{fi}\end{aligned}$$

□

Like with finite acceptance trees we can also map behaviour expressions into ffATs.

Definition 5.1.5

$$\begin{aligned}&\textbf{def } PffAT : PA_{io} \rightarrow ffAT\\ &\textbf{with } PffAT\ [\![\textbf{STOP}]\!] = NIL_{ffAT}\\ &\qquad PffAT\ [\![?m!\sigma; p\ ]\!] = (m,\sigma)_{ffAT}\ PffAT\ [\![\ p\ ]\!]\\ &\qquad PffAT\ [\![\ p_1 \parallel p_2]\!] = PffAT\ [\![\ p_1]\!] +_{ffAT}\ PffAT\ [\![\ p_2]\!]\end{aligned}$$

□

A partial order $\leq_{ffAT}$ on $ffAT$-representations can easily be defined as:

Definition 5.1.6

$$\begin{aligned}&\textbf{def } \leq_{ffAT} : ffAT \rightarrow ffAT \rightarrow \mathbb{B}\\ &\textbf{with } t \leq_{ffAT} t' = L_{ffAT}\ t = L_{ffAT}\ t' \wedge\\ &\qquad\qquad \forall(s : L_{ffAT}\ t'\ .\ \mathcal{A}_{ffAT}\ t'\ s \subseteq \mathcal{A}_{ffAT}\ t\ s)\end{aligned}$$

□

With the introduction of $ffAT$ we also reached a point where we cannot take profit of the results established by Hennessy. From now on the necessary properties are explicitely proven. From every lemma and theorem we introduce we give either its detailed transformational proof if it is not too much lengthy, or a proof outline with a reference to the last section of this chapter where a detailed transformational proof can be found.

First thing we need to show is that the partial order $\leq_{ffAT}$ on $ffAT$ is indeed a partial order, i.e. reflexive, anti-symmetric and transitive. The proof follows from the fact that the first condition of the definition of $\leq_{ffAT}$ can be replaced by $L_{ffAT}\ t \subseteq L_{ffAT}\ t'$. The converse is implied by the second condition of $\leq_{ffAT}$.

Further we have to show that $\langle ffAT, \leq_{ffAT}, \Sigma_{ffAT}\rangle$ is a Σ-partial order algebra. With Σ_{ffAT} the set of operators on $ffAT$ defined in Def. 5.1.4. This means that we have to show that i/o-prefix and $+_{ffAT}$ are monotonic with respect to the partial order $\leq_{ffAT}$.

Lemma 5.1.7
Given t' and t'' in $ffAT$ then:

i) $t' \leq_{ffAT} t'' \Rightarrow \forall((m,\sigma) : (M \times M^*) \,.\, (m,\sigma)_{ffAT}\ t' \leq_{ffAT} (m,\sigma)_{ffAT}\ t'')$

ii) $t' \leq_{ffAT} t'' \Rightarrow \forall(t : ffAT \,.\, t +_{ffAT} t' \leq_{ffAT} t +_{ffAT} t'')$

Proof
The proof is straightforward and depends mainly on the fact that the operators ⓝ, *mfs* and c are monotonic with respect to set inclusion. A detailed transformational proof can be found on page 105.

□

The next theorem shows that the partial order $\leq_{ffAT}$ corresponds to the testing preorder $\ll$ on labeled transition systems if the functionality restriction (FR) holds. This shows that $\leq_{ffAT}$ is as discriminative as $\leq_{fAT}$ provided that only process expressions are considered for which the finite acceptance tree contains only functional acceptance sets. In order to formulate the next theorem we introduce the notion of *c-functional* for *LTS*s.

Definition 5.1.8 *Functionality Restriction*
A process p is c-functional if its *LTS* satisfies the following condition:

$$\forall(s : (M \times M^*)^* \,.\, \forall(Y : c\ (\mathcal{A}\ p\ s) \,.\, func\ Y))$$

□

Formally, the main theorem we are concerned with in this section can be stated as:

Theorem 5.1.9 *Restricted correspondence of* $\leq_{ffAT}$ *with* $\ll$
For all p and q in PA_{io} which are c-functional

$$p \ll q \equiv PffAT\ [\ p\ ] \leq_{ffAT} PffAT\ [\ q\ ]$$

Proof

$$\begin{aligned}
& p \ll q \\
\equiv \quad & \{ \text{Definition of } \ll \} \\
& p \ll_{must} q \wedge p \ll_{may} q \\
\equiv \quad & \{ \text{Definition of } \ll_{must} \text{ and } \ll_{may} \} \\
& \forall(s : (M \times M^*)^* \,.\, \mathcal{A}\ q\ s \subset\subset \mathcal{A}\ p\ s) \wedge L\ p \subseteq L\ q \\
\equiv \quad & \{ \text{Lemma 4.2.4} \} \\
& \forall(s : (M \times M^*)^* \,.\, \mathcal{A}\ q\ s \subset\subset \mathcal{A}\ p\ s) \wedge L\ p = L\ q \\
\equiv \quad & \{ \text{Lemma 4.3.7} \}
\end{aligned}$$

$$
\begin{array}{ll}
 & \forall(s : (M \times M^*)^* \,.\, c\ (\mathcal{A}\ q\ s) \subseteq c\ (\mathcal{A}\ p\ s) \wedge L\ p = L\ q) \\
\equiv & \quad \{\, p \text{ and } q \text{ are c-functional} \,\} \\
 & \forall(s : (M \times M^*)^* \,.\, mfs\ (c\ (\mathcal{A}\ q\ s)) \subseteq mfs\ (c\ (\mathcal{A}\ p\ s))) \wedge L\ p = L\ q \\
\equiv & \quad \{\, \text{Lemma 5.1.11} \,\} \\
 & \forall(s : (M \times M^*)^* \,.\, \mathcal{A}_{ffAT}\ (PffAT\ [\, q \,])\ s \subseteq \mathcal{A}_{ffAT}\ (PffAT\ [\, p \,])\ s) \wedge \\
 & \qquad\qquad L\ p = L\ q \\
\equiv & \quad \{\, \text{Lemma 5.1.12} \,\} \\
 & \forall(s : (M \times M^*)^* \,.\, \mathcal{A}_{ffAT}\ (PffAT\ [\, q \,])\ s \subseteq \mathcal{A}_{ffAT}\ (PffAT\ [\, p \,])\ s) \wedge \\
 & \qquad\qquad L_{ffAT}\ PffAT\ [\, p \,] = L_{ffAT}\ PffAT\ [\, q \,] \\
\equiv & \quad \{\, \text{Definition of } \leq_{ffAT} \,\} \\
 & PffAT\ [\, p \,] \leq_{ffAT} PffAT\ [\, q \,]
\end{array}
$$

□

The proof of the Theorem above uses a few lemmas which are proven below. The next lemma states the formal relation between *LTS*s and *ffAT*s. In the proof another lemma is needed which states that:

Lemma 5.1.10
For all A in $\mathcal{P}\,\mathcal{P}\,(M \times M^*)$

$$mfs\ (c\ (mfs\ A)) = mfs\ (c\ A)$$

Proof
The proof is by induction to the length of the proof of an element being in $c\ (mfs\ A)$ for the proof of $mfs\ (c\ (mfs\ A)) \subseteq mfs\ (c\ A)$. The inclusion $mfs\ (c\ A) \subseteq mfs\ (c\ (mfs\ A))$ can be proven more directly. A detailed proof can be found on page 107.

□

The lemma stating the formal relation between *LTS*s and *ffAT*s says that we can obtain the acceptance sets of the *ffAT* by taking together the acceptance sets in the *LTS* after a certain path, close this set and then replace all non-functional sets with their maximal functional subsets.

Lemma 5.1.11
For all p in PA_{io} and s in $(M \times M^*)^*$

$$mfs\ (c\ (\mathcal{A}\ p\ s)) = \mathcal{A}_{ffAT}\ (PffAT\ [\, p \,])\ s$$

Proof

The proof is by structural induction on p. The base case is $p = \mathbf{STOP}$, and the other cases are $p = ?m!\sigma; q$ and $p = q \,[\!]\, r$. The only interesting sequences s are those that are in $L\ P$ and if relevant we give separate proofs for $s = \varepsilon$ and for $s \neq \varepsilon$.

The base case $p = \mathbf{STOP}$, and $s = \varepsilon$:

$$
\begin{array}{ll}
 & \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\mathbf{STOP}])\ \varepsilon \\
= & \quad \{\ \text{Definition of } \mathit{PffAT}\ \} \\
 & \mathcal{A}_{\mathit{ffAT}}\ \mathit{NIL}_{\mathit{ffAT}}\ \varepsilon \\
= & \quad \{\ \text{Definition of } \mathit{NIL}_{\mathit{ffAT}}\ \} \\
 & \iota\ \emptyset \\
= & \quad \{\ \text{Definition of } c\ \} \\
 & c\ (\iota\ \emptyset) \\
= & \quad \{\ \text{Definition of } \mathit{mfs}\ \} \\
 & \mathit{mfs}\ (c\ (\iota\ \emptyset)) \\
= & \quad \{\ \text{Definition of } \mathcal{A}\ \mathbf{STOP}\ \varepsilon\ \} \\
 & \mathit{mfs}\ (c\ (\mathcal{A}\ \mathbf{STOP}\ \varepsilon))
\end{array}
$$

The case $p = ?m!\sigma; q$ has two parts:

a) $s = \varepsilon$.

$$
\begin{array}{ll}
 & \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\ p\ ])\ \varepsilon \\
= & \quad \{\ \text{Definition of } \mathit{PffAT}\ \} \\
 & \mathcal{A}_{\mathit{ffAT}}\ (m,\sigma)_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\ q\ ])\ \varepsilon \\
= & \quad \{\ \text{Definition of } (m,\sigma)_{\mathit{ffAT}}\ \} \\
 & \iota\ \iota\ (m,\sigma) \\
= & \quad \{\ \text{Definition of } c\ \} \\
 & c\ \iota\ \iota\ (m,\sigma) \\
= & \quad \{\ \text{Definition of } \mathit{mfs}\ \} \\
 & \mathit{mfs}\ (c\ \iota\ \iota\ (m,\sigma)) \\
= & \quad \{\ \text{Definition of } \mathcal{A}\ (?m!\sigma; q)\ \varepsilon\ \} \\
 & \mathit{mfs}\ (c\ (\mathcal{A}\ (?m!\sigma; q)\ \varepsilon))
\end{array}
$$

b) $s \neq \varepsilon$ so suppose $s = (m,\sigma) \succ s'$.

$$
\begin{array}{ll}
 & \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [?m!\sigma; q\ ])\ (m,\sigma) \succ s' \\
= & \quad \{\ \text{Definition of } \mathit{PffAT}\ \}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathcal{A}_{\mathit{ffAT}}\ ((m,\sigma)_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\![\ q\ ]\!]))\ (m,\sigma) \succ s' \\
= & \quad \{\text{ Definition of } (m,\sigma)_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\![\ q\ ]\!]) \} \\
 & \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\![\ q\ ]\!])\ s' \\
= & \quad \{\text{ I.H. }\} \\
 & \mathit{mfs}\ (c\ (\mathcal{A}\ q\ s')) \\
= & \quad \{\text{ Definition of } \mathcal{A}\ (?m!\sigma; q)\ (m,\sigma) \succ s' \} \\
 & \mathit{mfs}\ (c\ (\mathcal{A}\ (?m!\sigma; q)\ (m,\sigma) \succ s'))
\end{array}
$$

The case $p = q \,[\!]\, r$ has also two parts:
a) $s = \varepsilon$
First we show that:

$$
\begin{array}{ll}
 & \mathcal{A}\ p\ \varepsilon \\
= & \quad \{\ \mathit{LTS} \text{ have only one root }\} \\
 & S\ p\ \varepsilon \\
= & \quad \{\text{ Definition of } S \text{ on } \mathit{LTS}\ \} \\
 & S\ q\ \varepsilon \cup S\ r\ \varepsilon \\
= & \quad \{\ \mathit{LTS} \text{ have only one root }\} \\
 & (\mathcal{A}\ q\ \varepsilon)\ u\ (\mathcal{A}\ r\ \varepsilon)
\end{array}
$$

The proof of part a) then is:

$$
\begin{array}{ll}
 & \mathit{mfs}\ (c\ (\mathcal{A}\ p\ \varepsilon)) \\
= & \quad \{\text{ Lemma above }\} \\
 & \mathit{mfs}\ (c\ (\mathcal{A}\ q\ \varepsilon\ u\ \mathcal{A}\ r\ \varepsilon)) \\
= & \quad \{\text{ Lemma 5.4.1 }\} \\
 & \mathit{mfs}\ (\mathit{mfs}\ (c\ (\mathcal{A}\ q\ \varepsilon))\ u\ \mathit{mfs}\ (c\ (\mathcal{A}\ r\ \varepsilon))) \\
= & \quad \{\text{ Definition of } \textcircled{u}\} \\
 & \mathit{mfs}\ (c\ (\mathcal{A}\ q\ \varepsilon)) \ \textcircled{u}\ \mathit{mfs}\ (c\ (\mathcal{A}\ r\ \varepsilon)) \\
= & \quad \{\text{ I.H. }\} \\
 & \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\![\ q\ ]\!])\ \textcircled{u}\ \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\![\ r\ ]\!])\ \varepsilon \\
= & \quad \{\text{ Definition of } \mathcal{A}_{\mathit{ffAT}}\ \} \\
 & \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\![\ p\ ]\!])\ \varepsilon
\end{array}
$$

b) $s \neq \varepsilon$

$$\mathit{mfs}\ (c\ (\mathcal{A}\ p\ s))$$

$$
\begin{array}{ll}
= & \{\, \mathcal{A}\ p\ s = (\mathcal{A}\ q\ s) \cup (\mathcal{A}\ r\ s)\,\} \\
& \mathit{mfs}\ (c\ ((\mathcal{A}\ q\ s) \cup (\mathcal{A}\ r\ s))) \\
= & \{\, \text{Rule c3 page 53}\,\} \\
& \mathit{mfs}\ (c\ (c\ (\mathcal{A}\ q\ s) \cup c\ (\mathcal{A}\ r\ s))) \\
= & \{\, \text{Lemma 5.1.10}\,\} \\
& \mathit{mfs}\ (c\ (\mathit{mfs}\ (c\ (\mathcal{A}\ q\ s) \cup c\ (\mathcal{A}\ r\ s)))) \\
= & \{\, \mathit{mfs} \text{ distributes over } \cup\,\} \\
& \mathit{mfs}\ (c\ (\mathit{mfs}\ (c\ (\mathcal{A}\ q\ s)) \cup \mathit{mfs}\ (c\ (\mathcal{A}\ r\ s)))) \\
= & \{\, \text{I.H.}\,\} \\
& \mathit{mfs}\ (c\ (\ \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\ q\ ])\ s \cup\ \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\ r\ ])\ s)) \\
= & \{\, \text{Definition of } +_{\mathit{ffAT}}\,\} \\
& \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\ p\ ])\ s
\end{array}
$$

□

The two lemmas above bridge the most crucial parts of the main proof. What is left is to prove a minor lemma to reach the right shape of the preorder definition on *ffAT*s. It states that the language of a certain process is the same as the language of the *ffAT*-representation of that process.

Lemma 5.1.12

$$\forall(p : PA_{io}\ .\ L\ p = \ L_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\ p\ ]))$$

Proof
By straightforward structural induction to p.
i) Base case $p = \mathbf{STOP}$.

$$
\begin{array}{ll}
& L\ \mathbf{STOP} \\
= & \{\, \text{Definition of } L\,\} \\
& \{s\ .\ \exists(q : PA_{io}\ .\ \mathbf{STOP} \stackrel{\varepsilon}{\longrightarrow} q)\} \\
= & \{\, \text{Definition of STOP}\,\} \\
& \iota\ \varepsilon \\
= & \{\, \text{Definition of } I \ldots\,\} \\
& L_{\mathit{ffAT}}\ \mathit{NIL}_{\mathit{ffAT}} \\
= & \{\, \text{Definition of } \mathit{NIL}_{\ldots}\,\} \\
& L_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\mathbf{STOP}])
\end{array}
$$

ii) $p = ?m!\sigma; q$:

$$
\begin{array}{ll}
 & L\ (?m!\sigma; q) \\
= & \quad \{\text{Definition of } L\} \\
 & \{s\ .\ \exists(r : PA_{io}\ .\ ?m!\sigma; q - s \rightarrow_* r)\} \\
= & \quad \{\text{Set theory, definition of } -\rightarrow_*\} \\
 & \{(m,\sigma) \succ s'\ .\ \exists(r : PA_{io}\ .\ q - s' \rightarrow_* r)\} \cup \iota\ \varepsilon \\
= & \quad \{\text{Definition of } L\} \\
 & \{(m,\sigma) \succ s'\ .\ s' \in L\ q\} \cup \iota\ \varepsilon \\
= & \quad \{\text{I.H.}\} \\
 & \{(m,\sigma) \succ s'\ .\ s' \in\ L_{ffAT}\ (PffAT\ [\![\ q\ ]\!])\} \cup \iota\ \varepsilon \\
= & \quad \{\text{Definition of } (m,\sigma)_{ffAT}\ (PffAT\ [\![\ q\ ]\!])\} \\
 & L_{ffAT}\ ((m,\sigma)_{ffAT}\ (PffAT\ [\![\ q\ ]\!])) \\
= & \quad \{\text{Definition of } PffAT\} \\
 & L_{ffAT}\ (PffAT\ [\![?m!\sigma; q\ ]\!])
\end{array}
$$

iii) $p = q \parallel r$:

$$
\begin{array}{ll}
 & L\ (q \parallel r) \\
= & \quad \{\text{Definition of } L\} \\
 & \{s\ .\ \exists(k : PA_{io}\ .\ q \parallel r - s \rightarrow_* k)\} \\
= & \quad \{\text{Definition of } q \parallel r - (m,\sigma) \rightarrow_{pr} k\} \\
 & \{s\ .\ \exists(k : PA_{io}\ .\ q - s \rightarrow_* k)\} \cup \\
 & \{s\ .\ \exists(k : PA_{io}\ .\ r - s \rightarrow_* k)\} \\
= & \quad \{\text{Definition of } L\} \\
 & (L\ q) \cup (L\ r) \\
= & \quad \{\text{I.H.}\} \\
 & (\ L_{ffAT}\ (PffAT\ [\![\ q\ ]\!]) \cup (\ L_{ffAT}\ (PffAT\ [\![\ r\ ]\!])) \\
= & \quad \{\text{Definition of } PffAT\ [q \parallel r]\} \\
 & L_{ffAT}\ (PffAT\ [\quad\quad\ ] \\
= & \quad \{\text{Definition of } \quad\quad\} \\
 & L_{ffAT}\ (PffAT\ [\quad\quad\ ]
\end{array}
$$

□

This completes the proof of full abstraction of $\sqsubseteq_{\approx}$ with respect to $\leq_{ffAT}$ given the functionality restriction.

In the next section we show the relation between $\leq_{ffAT}$ and the definition of a choice operator on functions and sets of functions.

5.2 Function Equality and Testing Equivalence

In Chapter 3 we introduced operators on specifications which were inspired by operators used in process algebra. In this section we show that equality of *ffAT* representation of processes corresponds to equality of specifications. From section 5.1 we also know that under the functionality restriction equality of the *ffAT*-representation of two expressions p_1 and p_2 corresponds to p_1 and p_2 being testing equivalent. Joining the two results we can indeed conclude that equality of specifications built from the basic specification **STOP** and the operators on specifications i/o-prefix and choice corresponds to testing equivalence of processes built from the process algebraic operators stop, action prefix and choice under the functionality restriction.

In order not to confuse all the different interpretations that are given to behaviour expressions in this chapter, we will denote the functional interpretation of an expression p in SPA_{io} explicitely as $fun\ [\ p\ ]$.

$$\begin{array}{ll}\textbf{def} & fun\ [\ -\ ] : SPA_{io} \rightarrow SPEC \\ \textbf{with} & fun\ [\textbf{STOP}] = \textbf{STOP} \\ & fun\ [?m!\sigma; p\ ] = ?m!\sigma; p \\ & fun\ [\ p_1 \ [\!]\ p_2] = p_1 \ [\!]\ p_2\end{array}$$

Also let us recall the definition of choice on functions and specifications introduced in Chapter 3.

$$\begin{array}{l}\textbf{def} \ - \ [\!]\ - : MSPF \rightarrow MSPF \rightarrow SPEC \\ \textbf{with}\ (f_1 \ [\!]\ f_2)\ f = tree\ f \in det\ (tree\ f_1 +_{ffAT} tree\ f_2)\end{array}$$

$$\begin{array}{l}\textbf{def} \ - \ [\!]\ - : SPEC \rightarrow SPEC \rightarrow SPEC \\ \textbf{with}\ (S_1 \ [\!]\ S_2)\ f = \exists (f_1, f_2 : MSPF^2 .\ S_1\ f_1 \wedge S_2\ f_2 \wedge (f_1 \ [\!]\ f_2)\ f)\end{array}$$

The definition of function *det* can be found on page 29 and of function *tree* on page 16, the definition of the $+_{ffAT}$-operator on *ffAT* in definition 5.1.4 on page 63. We introduce $\widetilde{tree}$ for the set extension of the function *tree*, i.e. the function that given a predicate that defines a set of finite *MSPF* gives the corresponding set of deterministic functional trees (*DFT*). The set *DFT* is a subset of *ffAT*. They are those *ffAT*'s that have at their nodes acceptance sets that contain exactly one set which is moreover a functional set.

$$\textbf{def } \widetilde{tree} : (MSPF \to \mathbb{B}) \to \mathcal{P}\ DFT$$
$$\textbf{with } \widetilde{tree}\ A = \{d\ .\ \exists(f : MSPF\ .\ A\ f \wedge d = tree\ f)\}$$

The theorem we described can now be formalized. The lemmas we use in the proof that are not proven in previous sections are marked with an $*$. They are proven lateron in this section.

Theorem 5.2.1 *Restricted Correspondence*
For all p_1 and p_2 in PA_{io} which are c-functional

$$p_1 \sim p_2 \equiv fun\ [\ p_1] = fun\ [\ p_2]$$

Proof

$p_1 \sim p_2$
$\equiv$ { Lemma 4.2.6 and definition 4.1.5 and 4.2.5 }
$p_1 \simeq p_2$
$\equiv$ { Lemma 5.1.9 and definition 5.1.6 and p_1 and p_2 c-functional }
$PffAT\ [\ p_1] = PffAT\ [\ p_2]$
$\equiv$ { Lemma 5.2.2 $*$ }
$dec\ (\ PffAT\ [\ p_1]\ ) = dec\ (\ PffAT\ [\ p_2]\ )$
$\equiv$ { Lemma 5.2.7 $*$ }
$\widetilde{tree}\ (\ fun\ [\ p_1]\ ) = \widetilde{tree}\ (\ fun\ [\ p_2]\ )$
$\equiv$ { Lemma 5.2.8 $*$ }
$fun\ [\ p_1] = fun\ [\ p_2]$

□

The first marked lemma that is used in the proof of the theorem states that the decomposition of $ffATs$ into sets of deterministic functional trees (DFT) is injective.

Lemma 5.2.2 *dec is injective*
For all t_1 and t_2 in $ffAT$

$$dec\ t_1 = dec\ t_2 \Rightarrow t_1 = t_2$$

Proof
By deriving a contradiction if we assume that the lemma does not hold. Suppose:

$$
\begin{array}{ll}
 & dec\ t_1 = dec\ t_2 \wedge t_1 \neq t_2 \\
\Rightarrow & \quad \{\ t_1 \neq t_2\ \} \\
 & (\ L_{ffAT}\ t_1 = L_{ffAT}\ t_2 \wedge \exists(s :\ L_{ffAT}\ t_1\ .\ \mathcal{A}_{ffAT}\ t_1\ s \neq \mathcal{A}_{ffAT}\ t_2\ s)) \\
 & \vee \\
 & (\ L_{ffAT}\ t_1 \neq L_{ffAT}\ t_2)
\end{array}
$$

Each part in the above disjunction is worked out in a separate case:
i)

$$
\begin{array}{ll}
 & \mathcal{A}_{ffAT}\ t_1\ s \neq \mathcal{A}_{ffAT}\ t_2\ s \\
\Rightarrow & \quad \{\text{Without loss of generality, set theory}\} \\
 & \exists(X\ .\ X \in \mathcal{A}_{ffAT}\ t_1\ s \setminus \mathcal{A}_{ffAT}\ t_2\ s) \\
\Rightarrow & \quad \{\text{Definition of } dec\} \\
 & \exists(t : dec\ t_1\ .\ S_{ffAT}\ t\ s = X \wedge t \notin dec\ t_2) \\
\Rightarrow & \quad \{\text{Definition of } dec\} \\
 & dec\ t_1 \neq dec\ t_2
\end{array}
$$

And this is in contradiction with our asumption that $dec\ t_1 = dec\ t_2$.
ii)

$$
\begin{array}{ll}
 & L_{ffAT}\ t_1 \neq L_{ffAT}\ t_2 \\
\Rightarrow & \quad \{\varepsilon \text{ is for all } t \text{ contained in } L_{ffAT}\ t\} \\
 & \exists(s :\ L_{ffAT}\ t_1 \setminus L_{ffAT}\ t_2\ .\ s \neq \varepsilon) \\
\Rightarrow & \quad \{\text{Assume } s = s' \prec m, \text{ definition of } \mathcal{A}_{ffAT}\} \\
 & \exists(X :\ \mathcal{A}_{ffAT}\ t_1\ s'\ .\ m \in X) \\
 & \wedge \\
 & \forall(Y :\ \mathcal{A}_{ffAT}\ t_2\ s'\ .\ m \notin Y) \\
\Rightarrow & \quad \{\text{Set theory}\} \\
 & \mathcal{A}_{ffAT}\ t_1\ s' \neq \mathcal{A}_{ffAT}\ t_2\ s' \\
\Rightarrow & \quad \{\text{Case i)}\} \\
 & dec\ t_1 \neq dec\ t_2
\end{array}
$$

So also this case leads to a contradiction, and we can thus conclude that the above lemma holds.

□

Now we arrived at the crucial lemma in the proof of the theorem. The proof of this lemma requires a number of non-trivial other lemma's. Some of

them we discuss here in full detail, others are only mentioned, but a complete proof can be found in section 5.4.

A central role in the proof of the crucial lemma is a lemma that states a particular property of the closure operator c. This property was not yet mentioned by Hennessy and thus we are proving it here. The lemma essentially states that when the closure is taken of the union of two sets of sets A and B, the same result can be reached by taking the union of all sets that are the result of the closure of a set from A and another from B.

Lemma 5.2.3 *Pairwise*
For all non-empty sets A and B in $\mathcal{P}\,\mathcal{P}\,(M \times M^*)$

$$c\ (A \cup B) = \bigcup((a, b) : (c\ A) \times (c\ B)\ .\ c\ (\{a, b\}))$$

Proof
The proof is on the length of the proof of set X being in $c\ (A \cup B)$ in one direction, and the same technique but for X in $c\ (\{a, b\})$ for a in $c\ A$ and b in $c\ B$ in the other direction. A detailed proof can be found on page 108.

□

Once we proved this "pairwise"-lemma, we can easily prove an extension of this lemma in which non-functional sets are replaced by their maximal functional subsets.

Lemma 5.2.4 *Extended pairwise*
For all A and B in $\mathcal{P}\,\mathcal{P}\,(M \times M^*)$

$$mfs\ (c\ (A \cup B)) = \bigcup((a, b) : (mfs\ (c\ A)) \times (mfs\ (c\ B))\ .\ mfs\ (c\ (\{a, b\})))$$

Proof

$$\begin{array}{ll} & mfs\ (c\ (A \cup B)) \\ = & \quad \{\text{Lemma 5.2.3}\} \\ & mfs\ (\bigcup((a, b) : (c\ A) \times (c\ B)\ .\ c\ (\{a, b\}))) \\ = & \quad \{\text{Property of } mfs\} \\ & \bigcup((a, b) : (c\ A) \times (c\ B)\ .\ mfs\ (c\ (\{a, b\}))) \\ = & \quad \{\text{Set theory } \{a, b\} = \iota\ a \cup \iota\ b\} \\ & \bigcup((a, b) : (c\ A) \times (c\ B)\ .\ mfs\ (c\ (\iota\ a \cup \iota\ b))) \\ = & \quad \{\text{Rule c3 on page 53}\} \\ & \bigcup((a, b) : (c\ A) \times (c\ B)\ .\ mfs\ (c\ (c\ (\iota\ a) \cup c\ (\iota\ b)))) \\ = & \quad \{\text{Lemma 5.1.10}\} \end{array}$$

$$
\begin{array}{ll}
 & \bigcup((a,b) : (c\ A) \times (c\ B)\ .\ mfs\ (c\ (mfs\ (c\ (\iota\ a) \cup c\ (\iota\ b)))))\\
= & \quad \{\text{ Property of } mfs \}\\
 & \bigcup((a,b) : (c\ A) \times (c\ B)\ .\ mfs\ (c\ (mfs\ (c\ (\iota\ a)) \cup mfs\ (c\ (\iota\ b)))))\\
= & \quad \{\text{ Definition of } c\text{; closure of singleton gives same singleton }\}\\
 & \bigcup((a,b) : (c\ A) \times (c\ B)\ .\ mfs\ (c\ (mfs\ (\iota\ a) \cup mfs\ (\iota\ b))))\\
= & \quad \{\text{ Pairwise lemma 5.2.3 }\}\\
 & \bigcup((a,b) : (c\ A) \times (c\ B)\ .\\
 & mfs\ (\bigcup((x,y) : (mfs\ (\iota\ a)) \times (mfs\ (\iota\ b))\ .\ c\ (\{x,y\}))))\\
= & \quad \{\text{ Set theory }\}\\
 & \bigcup((a,b) : (mfs\ (c\ A)) \times (mfs\ (c\ B))\ .\ mfs\ (c\ (\{a,b\})))
\end{array}
$$

□

Now we can almost prove the lemma in the main proof that says that a *ffAT* can be decomposed into a set of a special kind of *ffAT* which we called deterministic functional trees (*DFT*). These are *ffAT*s but at their nodes they have acceptance sets that contain exactly one set. Remark that $DFT \subseteq ffAT$ and that we therefore can use the operator $+_{ffAT}$ on them. When we do this it is of course not guaranteed that the result gives again an element in *DFT*. Therefore we define a special $+_{DFT}$ operator that takes sets of *DFT* as its arguments and results again in a set of *DFT*.

Definition 5.2.5

$$
\begin{array}{l}
\textbf{def } +_{DFT} : \mathcal{P}\ DFT \to \mathcal{P}\ DFT \to \mathcal{P}\ DFT\\
\textbf{with } D_1 +_{DFT} D_2 = \{d\ .\ \exists((d_1,d_2) : D_1 \times D_2\ .\ d \in dec\ (d_1 +_{ffAT} d_2))\}
\end{array}
$$

□

The following lemma shows that we can distribute decomposition over $+_{ffAT}$ using the plus on sets of *DFT*.

Lemma 5.2.6
For all p_1 and p_2 in SPA_m

$$
\begin{array}{l}
dec\ (PffAT\ [\![p_1]\!] +_{ffAT} PffAT\ [\![p_2]\!])\\
=\\
(dec\ PffAT\ [\![p_1]\!]) +_{DFT} (dec\ PffAT\ [\![p_2]\!])
\end{array}
$$

Proof
The proof can be found on page 116.

□

We now show that a *ffAT* can be decomposed into a set of deterministic functional trees.

Lemma 5.2.7 *Decomposition*
For all p in SPA_{io}

$$dec\ (PffAT\ [\ p\ ]) = \widetilde{tree}\ (\ fun\ [\ p\ ]\ )$$

Proof
By induction to the structure of expression p.
i) $p =$ **STOP**

$dec\ (PffAT$ **[STOP]**$)$
$=$ { Definition of $PffAT$ }
$dec\ (NIL_{ffAT})$
$=$ { Definition of dec }
$\iota\ NIL_{ffAT}$
$=$ { Definition of $\widetilde{tree}$ }
$\widetilde{tree}$ **(STOP)**
$=$ { Definition of fun [ ] }
$\widetilde{tree}\ (\ fun$ **[STOP]** $)$

ii) $p = ?m!\sigma; q.$

$dec\ (PffAT\ [?m!\sigma; q\ ])$
$=$ { Definition of $PffAT$ }
$dec\ ((m,\sigma)_{ffAT}\ (PffAT\ [\ q\ ]))$
$=$ { Definition of dec }
$(m,\sigma)_{ffAT}\ (dec\ (PffAT\ [\ q\ ]))$
$=$ { I.H. }
$(m,\sigma)_{ffAT}\ (\widetilde{tree}\ (fun\ [\ q\ ]\ ))$
$=$ { Easy property of $\widetilde{tree}$ }
$\widetilde{tree}\ ((m,\sigma)_{ffAT}\ (fun\ [\ q\ ]\ ))$
$=$ { Definition of of fun [ ] }
$\widetilde{tree}\ (\ fun\ [?m!\sigma; q\ ]\ )$

iii) $p = q \parallel r$.

$$
\begin{aligned}
& dec\ (PffAT\ [\ q \parallel r\ ]) \\
= & \quad \{ \text{Definition of } PffAT \} \\
& dec\ (PffAT\ [\ q\ ] +_{ffAT} PffAT\ [\ r\ ]) \\
= & \quad \{ \text{Lemma 5.2.6} \} \\
& dec\ (PffAT\ [\ q\ ]) +_{DFT} dec\ (PffAT\ [\ r\ ]) \\
= & \quad \{ \text{I.H.} \} \\
& \widetilde{tree}\ (\ fun\ [\ q\ ]\ ) +_{DFT} \widetilde{tree}\ (\ fun\ [\ r\ ]\ ) \\
= & \quad \{ \text{Lemma 5.4.6} \} \\
& \widetilde{tree}\ (\ fun\ [\ q \parallel r\ ]\ )
\end{aligned}
$$

□

What remains to be proven in the main proof on page 72 is the fact that $\widetilde{tree}$ is injective. Of course this proof is based on the idea that we can uniquely represent a finite *MSPF* by a finite tree with input-output tuples at its arcs. This construction was explained on page 16.

Lemma 5.2.8 *$\widetilde{tree}$ is injective*
For all p and q finite PA_{io}

$$\widetilde{tree}\ (\ fun\ [\ p\ ]\ ) = \widetilde{tree}\ (\ fun\ [\ q\ ]\ ) \Rightarrow\ fun\ [\ p\ ]\ =\ fun\ [\ q\ ]$$

Proof
The proof of this lemma can be found on page 116.

□

Note that there is a direct relation between the functional framework with the combinators we defined in Chapter 3 and process algebra. Let us take the process algebra which set of terms is called *PA* and which are defined by the following grammar where m are elements in the set M:

$$P ::= \mathbf{stop} \mid (m; P) \mid (P \mid P)$$

The operational semantics are defined by a *LTS* denoted by $\langle PA, M, - \rightarrow_{pr} \rangle$. The predicate $- \rightarrow_{pr}$ is defined by the following set of equations:

$$\begin{aligned}
&\mathbf{stop} \;-\; m \rightarrow_{pr} P \equiv \mathit{false}\\
&m; P \;-\; a \rightarrow_{pr} P' \equiv m = a \wedge P = P'\\
&P_1 \,[\!]\, P_2 \;-\; a \rightarrow_{pr} P' \equiv\; P_1 \;-\; a \rightarrow_{pr} P' \vee\\
&\qquad\qquad\qquad\qquad\quad P_2 \;-\; a \rightarrow_{pr} P'
\end{aligned}$$

The translation function from this process algebra PA to $SPEC$ can be defined as follows:

Definition 5.2.9 *Translation*

$$\begin{aligned}
&\mathbf{def}\ T : PA \rightarrow SPEC\\
&\mathbf{with}\ \ T\ [\mathbf{stop}\,] = \mathbf{STOP}\\
&\qquad T\ [\,m; B\,] = ?m!\tau\ m; T\ [\,B\,]\\
&\qquad T\ [\,B_1 \,[\!]\, B_2] = T\ [\,B_1] \,[\!]\, T\ [\,B_2]
\end{aligned}$$

□

Every f such that $T \,[\![\, B \,]\!]\, f$ represents a kind of "acceptor" for (a subset of) the traces of B in the sense that for all such traces σ it holds $f\sigma = \sigma$ whereas for all other σ' it holds $f\sigma' \sqsubseteq \sigma'$. Because of non-determinism, there might be more than one such function.

As a trivial corrolary we have that testing equivalence we have that for all p and q in PA $p \sim q$ if and only if $\backslash fun\ T\ [\,p\,] = \backslash fun\ T\ [\,q\,]$.

5.3 A Notion of Testing for the Functional Algebra

In this section we discuss the consequences of the functionality restriction we had to impose on the process expressions in order to prove full abstraction of testing equivalence w.r.t. equality of sets of functions. That this restriction is really necessary can be shown by the following simple example.

Consider the expression p defined as

$$(?m!\sigma; ?m_1!\sigma_1.\mathbf{STOP} \;[\!]\; ?m!\sigma; ?m_1!\sigma_2; \mathbf{STOP})$$

This expression can be represented by the *LTS* in Fig. 5.8 and by the *fAT* shown in Fig. 5.9. Its *fAT* representation is shown in Fig. 5.10 and it is clear this can be decomposed into a set of stream processing functions shown in Fig. 5.11.

Now consider the expression q defined as

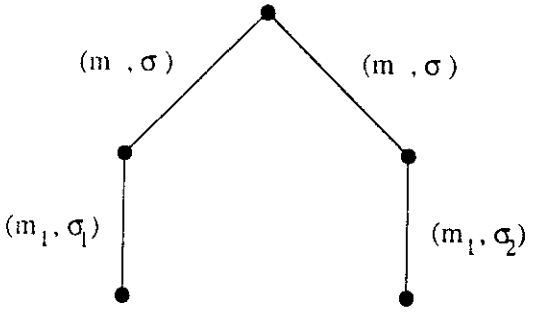

Figure 5.8: *LTS* of p

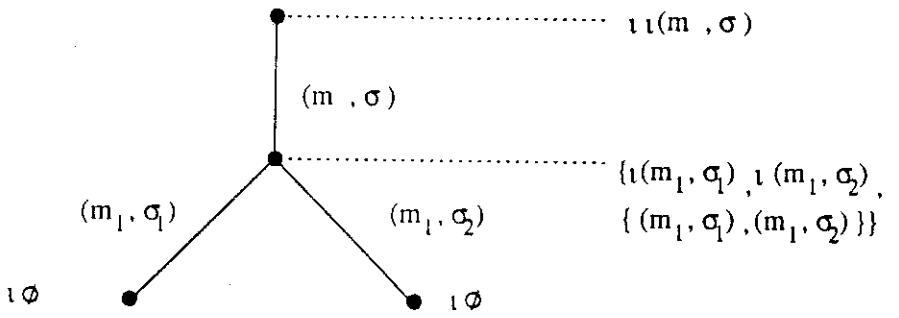

Figure 5.9: *fAT*-representation of p

$$?m!\sigma;(?m_1!\sigma_1;\mathbf{STOP} \parallel ?m_1!\sigma_2)$$

Also from this expression we can derive its *LTS*, *fAT* and *ffAT* representations and the set of functions. They are shown in the Figs. 5.12, 5.13, 5.14 and 5.15 respectively.

When we compare the *fAT* representations we immediately see that they are different, so based on the testing theory of Hennessy we have to conclude that the two processes are not testing equivalent. If we look at their *ffAT*-interpretations, however, we see that they *are* the same ! So we found two expressions which are not testing equivalent but which are represented by the same set of monotonic stream processing functions. At first glance this looks a rather disappointing result. But if we take a closer look at what kind of expressions exactly make problems, it turns out that it is only due to the way output is dealt with.

Suppose we indeed consider the two expressions above, p and q, equivalent. So suppose a new equivalence is defined as equality of the *ffAT*-

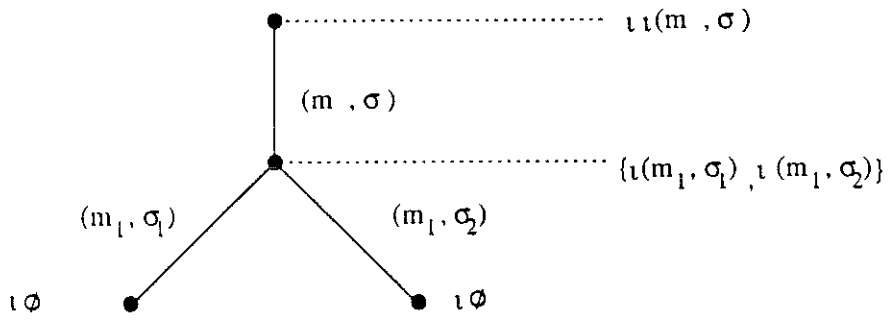

Figure 5.10: *ffAT*-representation of p

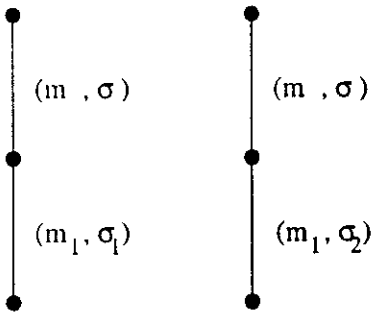

Figure 5.11: Set of functions representing p

representations of expressions. What does it mean to say that the example expressions p and q are equivalent? Well, actually, in this case it means that in our context of actions seen as input/output pairs we assume that if we give a certain input to a process we (as environment) cannot have any influence anymore on the output the process will give. So suppose in our example, we give input m_1 after having given input m and having got output σ. We know both systems are able to give as an output either σ_1 or σ_2. In process p this output is decided internally. Process q, however, shows that experimenters in the sense of the theory of Hennessy can have influence on the output after input m_1. We can specify an experimenter in such a way that process q *must* satisfy the test that after (m, σ) gives input m_1 and gets result σ_1. The test looks like:

$$?m!\sigma; ?m_1!\sigma_1; \mathrm{W}; \mathrm{STOP}$$

So the experimenter can somehow forse process q to give the result required by the experimenter for obtaining success.

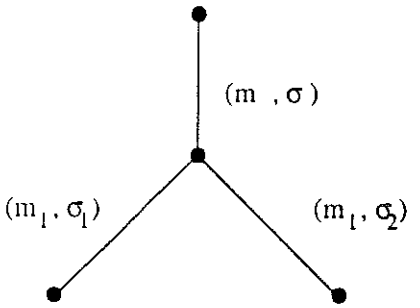

Figure 5.12: *LTS* of q

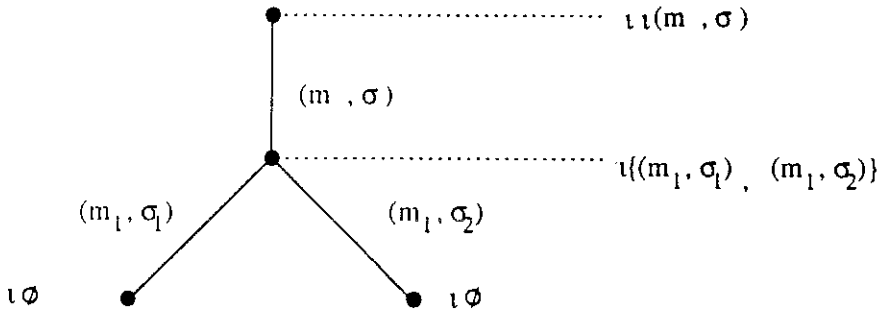

Figure 5.13: *fAT*-representation of q

We prefer, however, only the input part of the actions to be subject to external non-determinism. The equivalence, defined as equality of *ffAT*-representations, models the output as something which is a result of the incoming input upto a certain point, and the state of the process. If there is more then one output possible, then it is decided by the process itself which output to provide and thus the actual output is subject to internal non-determinism. The fact that after a certain input there are some different possibilities for output in this context can be interpreted as that in the specification an abstraction is made related to some part of the behaviour of a system that is not explicitely modeled. For example in the model of the behaviour of an unreliable channel we might not be interested in the exact cause of why and how the channel might sometimes not succeed in transferring a message in a proper way. We might only be interested in modeling the fact that every now and then some message is damaged by transport through the channel. That means that for some input we have different outputs, namely the correct message and a corrupted message. We don't want

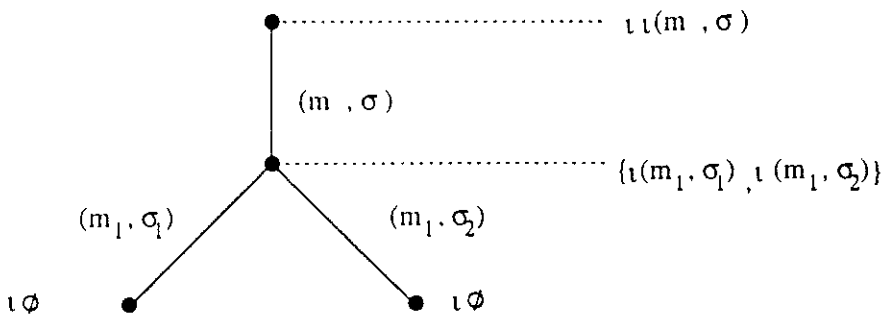

Figure 5.14: *ffAT*-representation of q

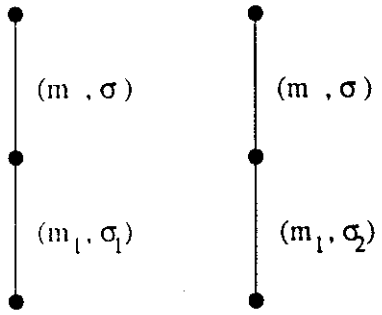

Figure 5.15: Set of functions representing q

to model, however, that the system can be forced to produce only one of the two possible outcomes when we test the system.

Our notion of output is therfore slightly different from that of Hennessy. In our model the output is only depending on the input and on the state the process reached. After some input has been supplied and the process reached a certain state in which it can produce output, we assume that the output is completely depending on the process itself and can in no way be influenced anymore by the environment. In the model proposed by Hennessy actions are not structured in the input/output parts. Moreover output actions are treated in the same way as input actions (actually, conceptually, there is no distinction between input and output actions). This leaves the possibility to specify systems where the environment can indeed *choose* which actual output a system should report when provided with a certain input for which different potential outputs are possible.

Consider for instance processes p and q in Fig. 5.16. Both processes accept input $in1$ and may produce either $out1$ or $out2$ as output. In partic-

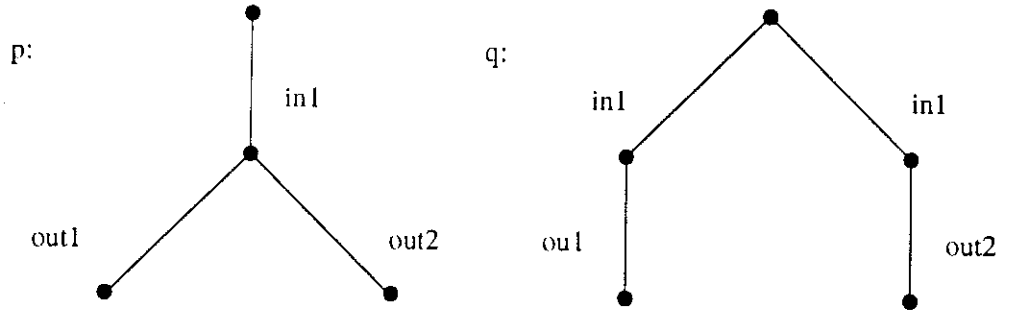

Figure 5.16: *LTS*-representation of p and q

ular, after input $in1$, process p reaches a state in which p can produce either output $out1$ or $out2$. If the environment (for example the experimenter) requires only output $out1$ and is not prepared to receive $out2$ the process p will always provide the required output $out1$, despite the process is in principle able to provide also $out2$.

Also process q, after input $in1$ has been given, is in principle able to provide either output $out1$ or $out2$. But the environment clearly cannot forse the output to be $out1$ because the choice which output will be provided is up to the process. This also shows why, in Hennessy's definition of testing equivalence, the processes are not considered testing equivalent, and moreover it shows that indeed the output can be influenced by the environment after the input has been given.

In this section we show that equality of sets of monotonic stream processing functions representing processes, which we proved fully abstract with respect to equality of the *ffAT*-representations of these processes, corresponds to a very intuitive and general notion of testing processes.

In section 5.2 we have proven that equality of sets of functions represented by expressions built from **STOP**, input/output prefix and choice corresponds to equality of the *ffAT*-representation of these expressions. Formally, as a direct corollary of Theorem 5.2.1 we obtain that

$$fun[\![p]\!] \subseteq fun[\![q]\!]$$
$$\equiv$$
$$PffAT\ p \leq_{ffAT} PffAT\ q$$

We can also find a relation $\ll_{..}$ on *LTS*s that corresponds to the $\leq_{ffAT}$-relation on *ffAT*-representations. We define the preorders $\ll_{mayio}$, $\ll_{mustio}$ and $\ll_{io}$ as follows.

Definition 5.3.1

$$\textbf{def} \; - \ll_{mayio} - : PA_{io} \to PA_{io} \to \mathbb{B}$$
$$\textbf{with} \; p \ll_{mayio} p' = L\ p \subseteq L\ p'$$

$$\textbf{def} \; - \ll_{mustio} - : PA_{io} \to PA_{io} \to \mathbb{B}$$
$$\textbf{with} \; p \ll_{mustio} p' = \forall (s : (M \times M^*)^* \;.\; mfs\ (c\ (\mathcal{A}\ p'\ s)) \subset\subset mfs\ (c\ (\mathcal{A}\ p\ s)))$$

$$\textbf{def} \; - \ll_{io} - : PA_{io} \to PA_{io} \to \mathbb{B}$$
$$\textbf{with} \; p \ll_{io} p' = p \ll_{mayio} p' \wedge p \ll_{mustio} p'$$

□

Defining preorders implies that we have to show that they are really preorders. The proof in this case follows directly from the definitions.

Lemma 5.3.2
The relations $\ll_{mayio}$, $\ll_{mustio}$ and $\ll_{io}$ are preorders.

Proof
Trivial.

□

We define also an equivalence based on the above preorders.

Definition 5.3.3

$$\textbf{def} \; - \simeq_{io} - : PA_{io} \to PA_{io} \to \mathbb{B}$$
$$\textbf{with} \; p \simeq_{io} p' = p \ll_{io} p' \wedge p' \ll_{io} p$$

□

Note that in the case that p and p' are c-functional and have the same set of input/output actions as labels of the corresponding *LTS*s the above definitions are equal to the original definitions $\ll_{may}$ and $\ll_{must}$ on *LTS*s.

Before proving the correspondence between $\leq_{fAT}$ and $\ll_{io}$ we prove two lemmas that are used in the correspondence proof.

Lemma 5.3.4
If Ψ and Φ are sets of sets of tuples, and if they contain the same labels then

$$mfs\ (c\ (\Psi)) \subset\subset mfs\ (c\ (\Phi))$$
$$\equiv$$
$$mfs\ (c\ (\Psi)) \subseteq mfs\ (c\ (\Phi))$$

Proof
The proof consists of two parts, one for each implication.
i) $\Leftarrow$-part is trivial.
ii) $\Rightarrow$-part uses a lemma which is proven in section 5.4.

$$\begin{array}{ll} & x \in mfs\ (c\ \Psi) \\ \Rightarrow & \quad \{ \text{Definition of } \subset\subset \} \\ & \exists (y : mfs\ (c\ \Phi)\ .\ y \subseteq x) \\ \Rightarrow & \quad \{ \text{Definition of } mfs \} \\ & \exists (z : c\ \Phi\ .\ y \in mfs\ (\iota\ z) \wedge y \subseteq x) \\ \Rightarrow & \quad \{ \text{Lemma 5.4.7} \} \\ & (x \cup z) \in c\ \Phi \wedge x \in mfs\ (\iota\ (x \cup z)) \\ \Rightarrow & \quad \{ \text{Definition of } mfs \} \\ & x \in mfs\ (c\ \Phi) \end{array}$$

□

The second lemma is a lemma which is very similar to lemma 4.2.4.

Lemma 5.3.5
For all p and p' in PA_{io}

$$p \ll_{mustio} p' \Rightarrow L\ p' \subseteq L\ p$$

Proof

$$\begin{array}{ll} & s \in L\ p' \\ \equiv & \quad \{ \text{Definition of } LTS \} \\ & \exists (r : PA_{io}\ .\ p' - s \rightarrow .\ r) \\ \Rightarrow & \quad \{ \text{Definition of } \mathcal{A}\ p'\ s \} \\ & \mathcal{A}\ p'\ s \neq \emptyset \\ \Rightarrow & \quad \{ \text{Definition of } c \} \\ & c\ (\mathcal{A}\ p'\ s) \neq \emptyset \\ \Rightarrow & \quad \{ \text{Definition of } mfs \} \\ & mfs\ (c\ (\mathcal{A}\ p'\ s)) \neq \emptyset \end{array}$$

$\Rightarrow$ { Definition of $\ll_{mustio}$ }

$mfs\ (c\ (\mathcal{A}\ p\ s)) \neq \emptyset$

$\Rightarrow$ { Definition of mfs and c }

$\mathcal{A}\ p\ s \neq \emptyset$

$\Rightarrow$ { Definition of L }

$s \in L\ p$

□

The next lemma states the correspondence between $\ll_{io}$ and $\leq_{\mathit{ffAT}}$.

Lemma 5.3.6
For all p and q in PA_{io}

$$PffAT\ [\![\ p\ ]\!] \leq_{\mathit{ffAT}} PffAT\ [\![\ q\ ]\!]$$
$$\equiv$$
$$p \ll_{io} q$$

Proof

$PffAT\ [\![\ p\ ]\!] \leq_{\mathit{ffAT}} PffAT\ [\![\ q\ ]\!]$

$\equiv$ { Definition of $\leq_{\mathit{ffAT}}$ }

$\forall(s : (M \times M^*)^* .\ \mathcal{A}_{\mathit{ffAT}}\ (PffAT\ [\![\ q\ ]\!])\ s \subseteq \mathcal{A}_{\mathit{ffAT}}\ (PffAT\ [\![\ p\ ]\!])\ s) \wedge$
$L_{\mathit{ffAT}}\ (PffAT\ [\![\ p\ ]\!]) = L_{\mathit{ffAT}}\ (PffAT\ [\![\ q\ ]\!])$

$\equiv$ { Lemma 5.1.12 }

$\forall(s : (M \times M^*)^* .\ \mathcal{A}_{\mathit{ffAT}}\ (PffAT\ [\![\ q\ ]\!])\ s \subseteq \mathcal{A}_{\mathit{ffAT}}\ (PffAT\ [\![\ p\ ]\!])\ s) \wedge$
$L\ p = L\ q$

$\equiv$ { Lemma 5.1.11 }

$\forall(s : (M \times M^*)^* .\ mfs\ (c\ (\mathcal{A}\ q\ s)) \subseteq mfs\ (c\ (\mathcal{A}\ p\ s)) \wedge L\ p = L\ q)$

$\equiv$ { Lemma 5.3.4 }

$\forall(s : (M \times M^*)^* .\ mfs\ (c\ (\mathcal{A}\ q\ s)) \subset\subset mfs\ (c\ (\mathcal{A}\ p\ s)) \wedge L\ p = L\ q)$

$\equiv$ { Lemma 5.3.5 }

$\forall(s : (M \times M^*)^* .\ mfs\ (c\ (\mathcal{A}\ q\ s)) \subset\subset mfs\ (c\ (\mathcal{A}\ p\ s)) \wedge L\ p \subseteq L\ q)$

$\equiv$ { Definition of $\ll_{mustio}$ and $\ll_{mustio}$ }

$p \ll_{mustio} q$ [illegible] q

$\equiv$ { Definition of $\ll_{io}$ }

$p \ll_{io} q$

□

Notice that for processes p of Fig. 5.8 and q of Fig. 5.12 $p \simeq_{io} q$.

In this last part of this section we will show that the definitions of $\ll_{mayio}$ and $\ll_{mustio}$ correspond to a very intuitive and general notion of testing processes which is only slightly different from that used in the standard testing theory by Hennessy.

Suppose we have a process p, and we want to test the behaviour of this process. What we will do is to supply it with an input and start waiting for possible output of the process. For every possible output the process might give the experimenter must be prepared to receive it and to decide how to go on with testing, i.e. what input to give next, how to react etc.. If the experimenter at a certain point is satisfied about the results of the test it can decide to report success.

We model this by means of a new definition for experimenters and for their interaction with processes. The set of processes are the same as those defined in definition 4.1.7 on page 39.

The set of experimenters is defined as follows.

Definition 5.3.7 *Output respecting experimenters*
An output respecting experimenter is a process denoted by a term in the set PEO_{io} and which behaviour is given by the *LTS* $\langle PEO_{io}, M \times M^*, - \rightarrow_{exo} \rangle$, where

- PEO_{io} is the set of terms generated by the grammar

$$E ::= \mathbf{STOP} \mid ?m!F \mid E \,[\!]\, E \mid \mathbf{1}; E \mid \mathbf{W}; E$$

with $m \in M$ and $F \in M^* \rightarrow E$ a total function from sequences over M^* to expressions

- $- \rightarrow_{exo} : PE_{io} \times (M \times M^*) \times PEO_{io} \rightarrow \mathbb{B}$ is the transition predicate defined by the following set of equations

$$\mathbf{STOP} - (m, \sigma) \rightarrow_{exo} E \equiv \mathit{false}$$

$$?m!F - (n, \sigma) \rightarrow_{exo} E' \equiv m = n \wedge \sigma \in M^* \wedge E' = F\ \sigma$$

$$E_0 \,[\!]\, E_1 - (m, \sigma) \rightarrow_{exo} E' \equiv \begin{array}{l} E_0 - (m, \sigma) \rightarrow_{exo} E' \\ \vee \\ E_1 - (m, \sigma) \rightarrow_{exo} E' \end{array}$$

$$\mathbf{W}; E - (m, \sigma) \rightarrow_{exo} E' \equiv (m, \sigma) = \mathbf{W} \wedge E' = E$$

$$\mathbf{1}; E - (m, \sigma) \rightarrow_{exo} E' \equiv (m, \sigma) = \mathbf{1} \wedge E' = E$$

□

In the grammar for PEO_{io} F stands for a total function from M^* to E the set of expressions. From computability theory we know that there does not exist one single language in which all and only total functions from M^* to E can be expressed. But of course we can take a language in which we can express all total functions and also partial functions (for example Funmath). It is then an obligation of the experimenter that the functions used to express the experiments are all total. Notice also that by requiring $F : M^* \to E$ we restrict our testing theory only to those processes (and then functions) which can react only with finite sequences on any single message. We justify this restriction by the assumption that any practical experimenter can recognize only output of finite length.

The interaction between processes and experimenters is defined in the following experimental system:

Definition 5.3.8 *Experimental system respecting output*
Let LTS_P and LTS_E be two compatible labeled transition systems $\langle P, Act, - \rightarrow_{pr} \rangle$ and $\langle E, Act, - \rightarrow_{exo} \rangle$ then $\mathcal{ES}(LTS_P, LTS_E)$ is the experimental system $\langle P, E, - \rightarrow_{\|}, Success \rangle$ where

- $- \rightarrow_{\|}$ is a predicate defining the interaction relation by the following equation:

$$\begin{aligned} & e \parallel p - (m, \sigma) \rightarrow_{\|} e' \parallel p' \\ \equiv & \\ & (e - (m, \sigma) \rightarrow_{exo} e' \wedge p - (m, \sigma) \rightarrow_{pr} p') \vee \\ & (e - \mathbf{1} \rightarrow_{exo} e' \wedge p = p' \wedge (m, \sigma) = \mathbf{1}) \end{aligned}$$

- $Success = \{e : E \mid \exists (e' : E \cdot e - \mathbf{W} \rightarrow_{exo} e')\}$

□

For uniformity, here we will rename the relations *may* and *must* by may_{io} and $must_{io}$. Notice that the definition of the experimental system is essentially the same as the one we gave on page 42. Only the definition of experimenters is changed in such a way that experimenters cannot influence the output of a process in a different way than via giving input to the process.

Like in Hennessy-Milner logic we can define two kinds of experiments that will turn out to be sufficient. This means they are enough (together with one more kind of test which is defined later) to discriminate all processes that can be discriminated by using the whole range of tests.

We introduce the following shorthand notation to represent these tests in a readable form.

Definition 5.3.9

$$?m!*; E = ?m!F \text{ where } F\ \sigma = E$$

$$?m!\sigma\ [E_1]; E_2 = ?m!F \text{ where } F\ \gamma = \begin{array}{l} \mathit{if}\ \gamma = \sigma \\ \mathit{then}\ E_2 \\ \mathit{else}\ E_1 \\ \mathit{fi} \end{array}$$

$$[\!]\ (a : A\ .\ ?a!F) = \ ?a_1!F\ [\!]\ \ldots\ [\!]\ ?a_k!F \quad \text{where } A = \{a_1, \ldots, a_k\}$$

□

Now we introduce the two kinds of essential experimenters. The first kind expresses that the experiment can only fail if a process, after having performed the sequence s, accepts input a and gives output σ_a. If we assume that the sequence s is the sequence $(m_1, \sigma_1) \succ (m_2, \sigma_2) \succ \ldots \succ (m_n, \sigma_n)$, we can express this test called $e\ (s, (a, \sigma_a))$ as:

$$\begin{aligned} e\ (s, (a, \sigma_a)) = \ & \mathbf{1}; \mathbf{W}\ [\!]\ (?m_1!\sigma_1[\mathbf{1}; \mathbf{W}; \mathbf{STOP}]; \\ & (\mathbf{1}; \mathbf{W}\ [\!]\ (?m_2!\sigma_2[\mathbf{1}; \mathbf{W}; \mathbf{STOP}]; \\ & \quad . \\ & \quad . \\ & \quad . \\ & (\mathbf{1}; \mathbf{W}\ [\!]\ (?m_n!\sigma_n[\mathbf{1}; \mathbf{W}; \mathbf{STOP}]; \\ & (\mathbf{1}; \mathbf{W}\ [\!]\ (?a!\sigma_a[\mathbf{1}; \mathbf{W}; \mathbf{STOP}]; \mathbf{STOP}))\ldots) \end{aligned}$$

The second type of essential test only fails if a process, after performing a sequence s as defined before, cannot accept any input denoted by a finite set $A = \{a_1, \ldots, a_k\}$.

$$\begin{aligned} e\ (s, A) = \ & \mathbf{1}; \mathbf{W}\ [\!]\ (?m_1!\sigma_1[\mathbf{1}; \mathbf{W}; \mathbf{STOP}]; \\ & (\mathbf{1}; \mathbf{W}\ [\!]\ (?m_2!\sigma_2[\mathbf{1}; \mathbf{W}; \mathbf{STOP}]; \\ & \quad . \\ & \quad . \\ & \quad . \\ & (\mathbf{1}; \mathbf{W}\ [\!]\ (?m_n!\sigma_n[\mathbf{1}; \mathbf{W}; \mathbf{STOP}]; \\ & [\!]\ (a : A\ .\ ?a!\sigma_a; \mathbf{W}; \mathbf{STOP})\ldots) \end{aligned}$$

Let E denote the set of all experiments of the form $e\ (s,(a,\sigma_a))$ or $e\ (s,A)$. We first prove a small lemma that will be used in the proof of correspondence.

Lemma 5.3.10
For all p and p' in PA_{io} and s in $(M \times M^*)^*$

$$p \sqsubseteq^E_{\approx_{mustio}} p' \wedge s \in L\ p' \Rightarrow s \in L\ p$$

Proof
If $s = \varepsilon$, then trivially $s \in L\ p$.
Suppose $s = s' \prec a$ and $a \in M \times M^*$. This part is proven by deriving a contradiction if we assume $s' \prec a \notin L\ p$.

$$\begin{array}{ll} & s' \prec a \notin L\ p \\ \Rightarrow & \quad \{\text{ Definition of of essential test } e\ (s', a)\ \} \\ & p\ must_{io}\ e\ (s', a) \\ \Rightarrow & \quad \{\text{ Definition of } p \sqsubseteq^E_{\approx_{mustio}} p'\ \} \\ & p'\ must_{io}\ e\ (s', a) \\ \Rightarrow & \quad \{\text{ Definition of } L\ p'\ \} \\ & (s' \prec a) \notin L\ p' \end{array}$$

But we had assumed that $(s' \prec a) \in L\ p'$, so we derived indeed a contradiction, and thus the lemma holds.

□

For proving the correspondence between $p \sqsubseteq_{\approx_{mustio}} p'$ and $p \ll_{mustio} p'$ we first prove that $p \sqsubseteq^E_{\approx_{mustio}} p'$ implies $p \ll_{mustio} p'$ in the following lemma.

Lemma 5.3.11
For all p and p' in PA_{io} and $E = \{e\ (s,a), e\ (s,A)\}$ and $s \in (M \times M^*)^*$

$$p \sqsubseteq^E_{\approx_{mustio}} p' \Rightarrow p \ll_{mustio} p'$$

Proof
From the definition of $\ll_{mustio}$ we know we must show that $mfs\ (e\ (\mathcal{A}\ p'\ s)) \subset\subset mfs\ (e\ (\mathcal{A}\ p\ s))$. Note that we only need to consider sequences of tuples s which are in $L\ p'$, because if $s \notin L\ p'$ then $\mathcal{A}\ p'\ s = \emptyset$, and thus $mfs\ (e\ (\mathcal{A}\ p'\ s)) = \emptyset$ and then the relation $\subset\subset$ trivially holds.

So, consider $s \in L\ p'$. From lemma 5.3.10 we know that then also $s \in L\ p$ and thus $\mathcal{A}\ p\ s \neq \emptyset$ and thus also $mfs\ (c\ (\mathcal{A}\ p\ s)) \neq \emptyset$.

Now we can continue the proof by deriving a contradiction if we assume that $mfs\ (c\ (\mathcal{A}\ p'\ s)) \subset\subset mfs\ (c\ (\mathcal{A}\ p\ s))$ does *not* hold.

$$
\begin{array}{ll}
 & mfs\ (c\ (\mathcal{A}\ p'\ s)) \subset\subset mfs\ (c\ (\mathcal{A}\ p\ s)) \equiv false \\
\Rightarrow & \quad \{\ mfs\ (c\ (\mathcal{A}\ q\ s)) \neq \emptyset \text{ for } q \in \{p, p'\}, \text{ definition of } \subset\subset\ \} \\
 & \exists (R : mfs\ (c\ (\mathcal{A}\ p'\ s)) \ .\ \forall (S : mfs\ (c\ (\mathcal{A}\ p\ s)) \ .\ S \not\subseteq R))
\end{array}
$$

Now we first show that in each set $S \in mfs\ (c\ (\mathcal{A}\ p\ s))$ we can choose an element (m, σ) in such a way that it is not only different from all elements in the set R, but such that the input part m is different from all input parts of elements in R. We show this by contradiction: for any $S \in mfs\ (c\ (\mathcal{A}\ p\ s))$ we assume that it is impossible to choose such an element and we reach a contradiction. For easyness of notation, let $mfs\ (c\ (\mathcal{A}\ p\ s))$ be the set $\{S_1, ..., S_k\}$. Let also function *inputs* be defined as follows:

$$
\begin{array}{l}
\textbf{def}\ inputs : \mathcal{P}\ (M \times M^*) \rightarrow \mathcal{P}\ M \\
\textbf{with}\ inputs\ A = \{m\ .\ \exists (\sigma : M^*\ .\ (m, \sigma) \in A)\}
\end{array}
$$

So, now suppose that S_i differs from R only by elements that differ only in their output part. Which is

$$inputs\ S_i \subseteq inputs\ R \wedge \exists ((m, \sigma) : S_i\ .\ (m, \sigma) \notin R)$$

Because of the definitions of *mfs* and c, for each S_i one of the following cases apply:

i $S_i \in \mathcal{A}\ p\ s$

ii $S_i \in c\ (\mathcal{A}\ p\ s) \wedge S_i \notin \mathcal{A}\ p\ s$

iii $S_i \in mfs\ (c\ (\mathcal{A}\ p\ s)) \wedge S_i \notin \mathcal{A}\ p\ s \wedge S_i \notin c\ (\mathcal{A}\ p\ s)$

Case i) Suppose $S_i \in \mathcal{A}\ p\ s$.

We first show that $actions\ (\mathcal{A}\ p'\ s) \subseteq actions\ (\mathcal{A}\ p\ s)$. The function *actions* has been defined on page 56. For $\mathcal{A}\ p'\ s = \emptyset$ or if $\mathcal{A}\ p'\ s$ contains only the empty set this is trivial. For $\mathcal{A}\ p'\ s \neq \emptyset$ and containing a non-empty set we can derive:

$$
\begin{array}{ll}
 & \exists(K : \mathcal{A}\ p'\ s\ .\ (m,\sigma) \in K) \\
\Rightarrow & \quad \{\text{ Definition of } L\ \} \\
 & s \prec (m,\sigma) \in L\ p' \\
\Rightarrow & \quad \{\ p \sqsubseteq^{E}_{\widetilde{minstio}} p',\ \text{lemma 5.3.10}\ \} \\
 & s \prec (m,\sigma) \in L\ p \\
\Rightarrow & \quad \{\text{ Definition of } \mathcal{A}\ \} \\
 & \exists(K' : \mathcal{A}\ p\ s\ .\ (m,\sigma) \in K') \\
\Rightarrow & \quad \{\text{ Definition of } \mathit{actions}\ \} \\
 & (m,\sigma) \in \mathit{actions}\ (\mathcal{A}\ p\ s)
\end{array}
$$

Now we can show that we can find in $c\ (\mathcal{A}\ p\ s)$ a set T which is an enrichment of S_i with exactly those elements in R which have the same input part as those in S_i. We can do this because T is an intermediate set between S_i, which is in $\mathcal{A}\ p\ s$ and $\mathit{actions}\ (\mathcal{A}\ p\ s)$ which is an element of $c\ (\mathcal{A}\ p\ s)$ by property of c and which contains all elements that are in R because we showed $\mathit{actions}\ (\mathcal{A}\ p'\ s) \subseteq \mathit{actions}\ (\mathcal{A}\ p\ s)$. Formally, given S_i and R as above:

$$
\begin{array}{ll}
 & \mathit{inputs}\ S_i \subseteq \mathit{inputs}\ R \wedge S_i \in \mathcal{A}\ p\ s \\
\Rightarrow & \quad \{\ \mathit{actions}\ (\mathcal{A}\ p'\ s) \subseteq \mathit{actions}\ (\mathcal{A}\ p\ s),\ \text{prop.}c\ \} \\
 & \exists(\ T : c\ (\mathcal{A}\ p\ s)\ .\ \mathit{inputs}\ S_i = \mathit{inputs}\ T \wedge \\
 & \quad T = S_i \cup \{(m,\sigma)\ .\ m \in \mathit{inputs}\ S_i \wedge (m,\sigma) \in R\})
\end{array}
$$

It is now easy to see that the set $T' = \{(m,\sigma)\ .\ m \in \mathit{inputs}\ S_i \wedge (m,\sigma) \in R\}$ is an element of $\mathit{mfs}\ (c\ (\mathcal{A}\ p\ s))$. In fact $T' \subseteq T \in c\ (\mathcal{A}\ p\ s)$; moreover T' is functional since $T' \subseteq R$ and it is maximal. This last fact can be proven by contradiction: suppose there exists an element $(n,\gamma) \in T \setminus T'$, then by definition of T', since $\mathit{inputs}\ T = \mathit{inputs}\ S_i \subseteq \mathit{inputs}\ R$, there exists $(n,\gamma') \in T'$ and since T' is functional we get $\gamma = \gamma'$. So, in the end we found an $S_j \in \mathit{mfs}\ (c\ (\mathcal{A}\ p\ s))$ with $S_i \subseteq R$. Such an S_j is exactly T'. Formally:

$$
\begin{array}{ll}
 & \exists(\ T : c\ (\mathcal{A}\ p\ s)\ .\ \mathit{inputs}\ S_i = \mathit{inputs}\ T \wedge \\
 & \quad T = S_i \cup \{(m,\sigma)\ .\ m \in \mathit{inputs}\ S_i \wedge (m,\sigma) \in R\}) \\
\Rightarrow & \quad \{\ \text{Def. of } T' = \{(m,\sigma)\ .\ m \in \mathit{inputs}\ S_i \wedge (m,\sigma) \in R\}\ \} \\
 & \exists(S_j : \mathit{mfs}\ (c\ (\mathcal{A}\ p\ s))\ .\ S_j = T' \wedge S_j \subseteq R)
\end{array}
$$

This last fact contradicts the assumption (page 91)

$$\exists(R : \mathit{mfs}\ (c\ (\mathcal{A}\ p'\ s))\ .\ \forall(S : \mathit{mfs}\ (c\ (\mathcal{A}\ p\ s))\ .\ S \not\subseteq R))$$

The reasoning above is illustrated by Fig. 5.17.
Case ii) Suppose $S_i \in c\ (\mathcal{A}\ p\ s)$ but not in $\mathcal{A}\ p\ s$ itself, and such that *inputs* $S_i \subseteq$ *inputs* R. In this case the reasoning is the same as in case i).

Case iii) Suppose $S_i \in mfs\ (c\ (\mathcal{A}\ p\ s))$ but $S_i \notin \mathcal{A}\ p\ s$ and $S_i \notin c\ (\mathcal{A}\ p\ s)$. In that case by definition of *mfs* we know that there exists a set $K \in c\ (\mathcal{A}\ p\ s)$ such that *inputs* $S_i =$ *inputs* K and then *inputs* $K \subseteq$ *inputs* R. For K we can setup a reasoning like in case ii) leading to the fact that there will exist a set in $mfs\ (c\ (\mathcal{A}\ p\ s))$ which is a subset of R which is in contradiction with the assumptions, and thus such a S_i cannot exist.

This ends the proof for each of the cases for S_i and shows that in all sets in $mfs\ (c\ (\mathcal{A}\ p\ s)) = \{S_1, ..., S_k\}$ we can choose an element, with an input part different from all elements in R. And because we can find such an element in each S_i we can also find it in each set in $\mathcal{A}\ p\ s$. In fact let A_j be a set in $c\ (\mathcal{A}\ p\ s)$. If it is functional then $A_j = S_i$ for some S_i. If it is not functional then it includes some functional set S_i. In particular this holds for all A_j in $\mathcal{A}\ p\ s$.

Now let's choose in each S_i, $1 \leq i \leq k$ one element x_i such that the input part of x_i is not in *inputs* R. Let's call A the set of input parts of all this elements x_i. So

$$A = \{x0\ .\ x \in \{x_1, ..., x_k\}\}$$

then we can be sure that

$$p\ must_{io}\ e\ (s, A)$$

because for each set $A_j \in \mathcal{A}\ p\ s$ there exists an element (x, σ) with $x \in A$ as shown before.

Now we show that $p'\ must\ e\ (s, A)$. So we will show that there exists a state r which can be reached by p' by performing s such that the following computation is unsuccessful

$$e\ (s, A) \parallel p' - s \rightarrow \ldots \quad \ldots; \mathbf{W}; \mathbf{STOP}) \parallel r$$

Which means that *inputs* $\ldots = \emptyset$.

There are three possibilities:
i) $R \in \mathcal{A}\ p'\ s$ which means that there exists an r such that $S\ r \in \mathcal{A}\ p'\ s$ and $S\ r = R$. And we know that *inputs* $R \cap$ *inputs* $A = \emptyset$ so $p'\ must_{io}\ e\ (s, A)$.

ii) $R \in c\ (\mathcal{A}\ p'\ s)$ but $R \notin \mathcal{A}\ p'\ s$. This implies that by property of closure there exists an r such that $S\ r \subseteq R$, so again we have that $p'\ must_{io}\ e\ (s, A)$.

iii) $R \in mfs\ (c\ (\mathcal{A}\ p'\ s))$ but $R \notin \mathcal{A}\ p'\ s$ and $R \notin c\ (\mathcal{A}\ p'\ s)$. This means that there exists a set K in $c\ (\mathcal{A}\ p'\ s)$ such that $R \in mfs\ (\iota\ K)$. Because of a property of closure we know that there exists an r such that $S\ r \subseteq K$ which implies $inputs\ (S\ r) \subseteq inputs\ K$. Since R is a maximal functional subset of K we know that $inputs\ K = inputs\ R$, so $inputs\ (S\ r) \subseteq inputs\ R$. Thus also in this case we can find an r and thus an $S\ r$ such that the computation cannot be continued to lead to a success. So once more $p'\ must_{io}\ e\ (s, A)$.

This proves the lemma.

□

The proven lemma plays a major role in the following theorem that states the correspondence between the preorders.

Theorem 5.3.12 *Correspondence*
For every p and p' in PA_{io}

$a)\ p \sqsubseteq_{may} p' \equiv p \ll_{mayio} p'$

$b)\ p \sqsubseteq_{mustio} p' \equiv p \ll_{mustio} p'$

$c)\ p \sqsubseteq_{io} p' \equiv p \ll_{io} p'$

Proof
Part a)
We first prove:

$$\forall(e : PEOio\ .\ p\ may_{io}\ e \Rightarrow p'\ may_{io}\ e)$$
$$\Rightarrow$$
$$p \ll_{mayio} p'$$

The proof is by deriving a contradiction if $p \ll_{mayio} p'$ does not hold. We first define the following test for a certain sequence s equal to $(m_1, \sigma_1) \succ \ldots \succ (m_k, \sigma_k)$ that only succeeds if the process under test performs exactly s.

$$\begin{aligned} e\ (s) = \ & ?\ m_1!\sigma_1\ \mathrm{STOP}]; \\ & (?m_2!\sigma_2\ \mathrm{STOP}]; \\ & \quad . \\ & \quad . \\ & \quad . \\ & ?\ m_k!\sigma_k[\mathrm{STOP}]; \mathrm{W})\ldots) \end{aligned}$$

Now we derive:

$$
\begin{array}{ll}
 & p \ll_{mayio} p' \equiv false \\
\Rightarrow & \quad \{ \text{Definition of } \ll_{mayio} \} \\
 & L\ p \not\subseteq L\ p' \\
\Rightarrow & \quad \{ \text{Set theory} \} \\
 & \exists (s : L\ p\ .\ s \notin L\ p') \\
\Rightarrow & \quad \{ \text{Testing and definition of } e\ (s) \} \\
 & \exists (s : L\ p\ .\ p\ may_{io} e\ (s) \wedge p'\ may_{io} e\ (s))
\end{array}
$$

And this is in contradiction with the assumption that all tests p may satisfy also p' may satisfy. And thus $p \ll_{mayio} p'$.

Now we prove the reverse direction of part a. So we prove:

$$
\begin{array}{l}
p \ll_{mayio} p' \\
\Rightarrow \\
\forall (e : PEO_{io}\ .\ p\ may_{io} e \Rightarrow p'\ may_{io} e)
\end{array}
$$

Suppose $p\ may_{io} e$, for a certain experiment e, so in the set of all computations starting from $e \parallel p$ there must be at least one computation leading to success. Take this computation.

$$e \parallel p = e_0 \parallel p_0 - (m_1, \sigma_1) \rightarrow e_1 \parallel p_1 \ldots \rightarrow e_k \parallel p_k$$

This computation gives rise to two derivations: $e - s' \rightarrow_* e_k$ and $p - s \rightarrow_* p_k$ where s' is equal to s upto some **1**-steps.

We know that by definition $p \ll_{mayio} p' = L\ p \subseteq L\ p'$. So also p' can perform $p' - s \rightarrow_* p_k'$ for a certain p_k', and thus can be composed with the derivation we had for e. And since this experimenter e reported success somewhere, also this time it will do so because it is the same derivation for e. Thus $p'\ may_{io} e$.

Part b)

The implication $p \sqsubseteq_{mustio} p' \Rightarrow p \ll_{mustio} p'$ is already proven by lemma 5.3.11. The implication in the reverse direction can be proven as follows.

Suppose $p \ll_{mustio} p'$ and $p\ must_{io} e$. We have to show that $p'\ must_{io} e$. Let's consider an arbitrary computation starting from $e \parallel p'$. Note that this cannot be extended.

$$e \parallel p' = e_0 \parallel p_0' - (m_1, \sigma_1) \rightarrow e_1 \parallel p_1' \ldots \rightarrow e_k \parallel p_k'$$

We must prove that for some $n \in \{0, ..., k\}$, $e_n \in Success$. This computation gives rise to two derivations: $e - s' \rightarrow_* e_k$ and $p' - s \rightarrow_* p_k'$ for some $s \in (M \times M^*)^*$. Furthermore s' is equal to s upto **1**-steps.

Now $S\ (p_k') \in \mathcal{A}\ p'\ \ s$ and then there exists $T \in mfs\ (c\ \mathcal{A}\ p'\ \ s))$ with $T \subseteq S\ (p_k')$. Moreover, since $mfs\ (c\ (\mathcal{A}\ p'\ s)) \subset\subset mfs\ (c\ (\mathcal{A}\ p\ s))$ we can find $S' \in mfs\ (c\ (\mathcal{A}\ p\ s))$ such that $S' \subseteq T$. So we get $S' \subseteq S\ (p_k')$.

Now there are three cases for S':

i) $S' \in \mathcal{A}\ p\ s$. In that case there exists an r such that $p - s \rightarrow_* r$ and $S\ r \subseteq S\ (p_k')$. So $e_k \parallel r$ cannot be extended and therefore the derivations $e - s' \rightarrow_* e_k$ and $p - s \rightarrow_* r$ can be combined to give a computation $e \parallel p \rightarrow ... \rightarrow e_k \parallel r$. And since $p\ must_{io} e$ it follows that $e_n \in Success$ for some $n \in \{1, ..., k\}$.

ii) $S' \in c\ (\mathcal{A}\ p\ s)$, so there exists a set $S\ r$ in $\mathcal{A}\ p\ s$ such that $S\ r \subseteq S'$ and thus $S\ r \subseteq S\ (p_k')$. Then a similar reasoning as in case i) can be performed.

iii) $S' \in mfs\ (c\ (\mathcal{A}\ p\ s))$, so there exists a set K in $c\ (\mathcal{A}\ p\ s)$ such that $S' \in mfs\ (i\ K)$. And there is a set $S\ r$ in $\mathcal{A}\ p\ s$ such that $S\ r \subseteq K$. We know that $S' \subseteq S\ (p_k')$. We know also that $e_k \parallel p_k'$ cannot be extended, and that $inputs\ (S\ r) \subseteq inputs\ S'$ because of the definition of mfs. This, together with the fact that $e_k \parallel p_k'$ cannot be extended, brings to the fact that also $e_k \parallel r$ cannot be extended.

Part c)
This obviously follows from parts a) and b).

□

Theorem 5.3.12 gives us a number of interesting results. The first result is that, as an immediate corollary we can conclude that $\sqsubseteq_{mustio}$ coincides with $\sqsubseteq^E_{mustio}$ where E is the set of experimenters characterised by $e\ (s, a)$ and $e\ (s, A)$ for arbitrary s, a and A.

If we add also $e\ (s)$ to E it is easy to see that also $\sqsubseteq_{io}$ coincides with $\sqsubseteq^E_{io}$. So the set E contains enough kinds of tests to discriminate the processes we want to discriminate and we know that adding more kinds of tests does not increase the discriminative power.

Moreover, for expressing the essential tests in E we showed that we needed only a few kinds of functions for the function part called F in the syntax of experimenters. We needed only constant functions that map every output stream into the same experimenter expression and we needed functions that

on one certain output stream give a certain expression e_1 and for all others the expression e_2.

$$\begin{array}{l}\textbf{def } F_{*\epsilon} : M^* \to PEO_{io} \\ \textbf{with } F_{*\epsilon}\ \sigma = \epsilon\end{array}$$

$$\begin{array}{l}\textbf{def } F_{\sigma,e_1,e_2} : M^* \to PEO_{io} \\ \textbf{with } F_{\sigma,e_1,e_2}\ \gamma = \ \textit{if } \gamma = \sigma \\ \qquad\qquad\qquad\qquad\ \textit{then } e_2 \\ \qquad\qquad\qquad\qquad\ \textit{else } e_1 \\ \qquad\qquad\qquad\qquad\ \textit{fi}\end{array}$$

We can immediately see that these two kinds of functions are total on the domain M^* and they provide us a syntax so that we can have a language to express all the experimenters we need.

Another result is that the cause of the difference between classical testing equivalence as defined by Hennessy for process algebra and the testing equivalence we found appropriate in our functional approach is a slightly different treatment of actions that denote output. In the theory of Hennessy no difference is made between actions that denote input and those that denote output, they are all just considered actions. This resulted in the fact that both input and output actions can be influenced by the environment. This means that if a process after a certain input is in a state in which it can perform more then one output action, and the environment (experimenter) requires a specific one of this outputs to occur, the process will always give the required output. The output in the theory of Hennessy therefore may depend on the preceding input, the state the process reached *and* the requirements of the environment not expressed by explicit communication to the process via input actions.

In our definition of testing equivalence for a functional framework there *is* a difference between input and output. A process reacts on an input provided by the environment depending on the state the process is in and its output can only depend on preceding input and the state the process reached. The result is that our definition of testing equivalence corresponds to equality of sets of monotonic stream processing functions. It is an equivalence which is very much similar to the classical testing equivalence for process algebra, but a little bit weaker because in some cases processes are identified which are not identified by the classical testing equivalence relation.

5.4 Detailed Transformational Proofs

This section contains a number of detailed transformational proofs of a number of lemmas that are stated in this chapter. Their proofs are included in a separate section in order to avoid too much lenghty and detailed proofs to interrupt the main line of the story.

First we state and prove a number of auxiliary lemmas which were not stated before, and which will be used in the sequel.

Lemma 5.4.1
For all A and B in $\mathcal{P}\,\mathcal{P}\,M \times M^*$

$$mfs\ (A \cup B) = mfs\ (mfs\ A \cup mfs\ B)$$

Proof
The proof consists of two parts:
i) $mfs\ (A \cup B) \subseteq mfs\ (mfs\ A \cup mfs\ B)$

$$\begin{array}{ll}
 & X \in mfs\ (A \cup B) \\
\equiv & \quad \{\text{Definition of } mfs\} \\
 & \exists (Y : A \cup B \,.\, X \in mfs\ Y) \\
\Rightarrow & \quad \{\text{Definition of } \cup\} \\
 & \exists ((Y_1, Y_2) : A \times B \,.\, Y = Y_1 \cup Y_2 \wedge X \in mfs\ Y) \\
\equiv & \quad \{\text{Logic}\} \\
 & \exists ((Y_1, Y_2) : A \times B \,.\, X \in mfs\ (Y_1 \cup Y_2)) \\
\Rightarrow & \quad \{\text{Set theory}\} \\
 & \exists (K_1 : mfs\ (\cdots Y_1), K_2 : mfs\ (\cdots Y_2) \,.\, X \in mfs\ (K_1 \cup K_2)) \\
\Rightarrow & \quad \{\ Y_1 \in A \text{ and } Y_2 \in B\ \} \\
 & \exists (K_1 : mfs\ A \,.\, K_2 : \cdots B \,.\, X \in mfs\ (K_1 \cup K_2)) \\
\Rightarrow & \quad \{\text{Definition}\ \cdots\} \\
 & X \in mfs\ ((\cdots \cup \cdots mfs\ B))
\end{array}$$

ii) $mfs\ (mfs\ A \cup mfs\ \cdots) \subseteq (A \cup B)$

$$\begin{array}{ll}
 & X \in mfs\ (\cdots \cup \cdots mfs\ B)) \\
\equiv & \quad \{\text{Definition } \cdots mfs\} \\
 & \exists (Y : ((mfs\ \cdots mfs\ B)) \,.\, X \in mfs\ Y) \\
\equiv & \quad \{\text{Definition of } \cup\} \\
 & \exists (K_1 : mfs\ A \,.\, K_2 : mfs\ B \,.\, Y = K_1 \cup K_2 \wedge X \in mfs\ Y)
\end{array}$$

$\Rightarrow$ { Definition of *mfs* over set of sets }
$\exists(A_1 : A, B_1 : B \;.\; \exists(K_1 : mfs\ (\iota\ A_1), K_2 : mfs\ (\iota\ B_1)\;.\; X \in mfs\ (K_1 \cup K_2)))$
$\Rightarrow$ { Set theory }
$\exists(A_1 : A, B_1 : B \;.\; X \in mfs\ (A_1 \cup B_1))$
$\Rightarrow$ { Definition of u }
$X \in mfs\ (A\ u\ B)$

□

Lemma 5.4.2
For all $A \in \mathcal{P}\ \mathcal{P}\ (M \times M^*)$

$$\forall(Y : c\ (mfs\ A)\;.\; X \in mfs\ Y \Rightarrow X \in mfs\ (c\ A))$$

Proof
The proof is by induction on the length of the proof of Y being in $c\ (mfs\ A)$, and because of the definition of c there are three cases.
i) $Y \in mfs\ A$

$Y \in mfs\ A$
$\Rightarrow$ $\{ A \subseteq c\ A \Rightarrow mfs\ A \subseteq mfs\ (c\ A)) \}$
$Y \in mfs\ (c\ A)$
$\Rightarrow$ { Y is functional and $X \in mfs\ (\iota\ Y)$, so $X = Y$ }
$X \in mfs\ (c\ A)$

ii) $Y = Y_1 \cup Y_2$ and $Y_1 \in c\ (mfs\ A)$ and $Y_2 \in c\ (mfs\ A)$. By I.H. we have that

$$\forall(X_1, X_2\;.\; X_1 \in mfs\ (\iota\ Y_1) \Rightarrow X_1 \in mfs\ (c\ A) \wedge X_2 \in mfs\ (\iota\ Y_2) \Rightarrow X_2 \in mfs\ (c\ A))$$

We can now prove that $X \in mfs\ (\iota\ (Y_1 \cup Y_2)) \Rightarrow X \in mfs\ (c\ A)$:

$X \in mfs\ (\iota\ Y)$
$\equiv$ { Definition of $Y_1 \cup Y_2$ }
$X \in mfs\ (\iota\ (Y_1 \cup Y_2))$
$\equiv$ { Definition of u }
$X \in mfs\ ((\iota\ Y_1)\ u\ (\iota\ Y_2))$
$\equiv$ { Lemma 5.4.1 }
$X \in mfs\ (mfs\ (\iota\ Y_1)\ u\ mfs\ (\iota\ Y_2))$

$\equiv$ { Definition of u }
$\exists(X_1 : mfs\ (\iota\ Y_1), X_2 : mfs\ (\iota\ Y_2)\ .\ X \in mfs\ (\iota\ (X_1 \cup X_2)))$
$\Rightarrow$ { I.H. }
$\exists(X_1 : mfs\ (c\ A), X_2 : mfs\ (c\ A)\ .\ X \in mfs\ (\iota\ (X_1 \cup X_2)))$
$\Rightarrow$ { Definition of mfs }
$\exists(K_1 : c\ A, K_2 : c\ A\ .\ X_1 \in mfs\ (\iota\ K_1) \wedge X_2 \in mfs\ (\iota\ K_2) \wedge$
$X \in mfs\ (\iota\ (X_1 \cup X_2)))$
$\equiv$ { Definition of u }
$\exists(K_1 : c\ A, K_2 : c\ A\ .\ X \in mfs\ (mfs\ (\iota\ K_1)\ u\ mfs\ (\iota\ K_2)))$
$\equiv$ { Lemma 5.4.1 }
$X \in mfs\ (\iota\ K_1\, u\ \iota\ K_2)$
$\equiv$ { Definition of u }
$X \in mfs\ (\iota\ (K_1 \cup K_2))$
$\Rightarrow$ { $(K_1 \cup K_2) \in c\ A$ }
$X \in mfs\ (c\ A)$

iii) $Y_1 \subseteq Y \subseteq Y_2$ and $Y_1 \in c\ (mfs\ A)$ and $Y_2 \in c\ (mfs\ A)$.
By I.H. we have that

$$\forall(X_1, X_2\ .\ X_1 \in mfs\ (\iota\ Y_1) \Rightarrow X_1 \in mfs\ (c\ A) \wedge X_2 \in mfs\ (\iota\ Y_2) \Rightarrow X_2 \in mfs\ (c\ A))$$

We can now prove that $X \in mfs\ Y \Rightarrow X \in mfs\ (c\ A)$.

$X \in mfs\ (\iota\ Y)$
$\Rightarrow^1$ { $Y_1 \subseteq Y \subseteq Y_2$,prop. of closure }
$\exists(Y_1'\ .\ Y_1 \subseteq Y_1' \subseteq Y \wedge$
$\exists(X_1 : mfs\ (\iota\ Y_1').\ X_2 \in mfs\ (\iota\ Y_2)\ .\ X_1 \subseteq X \subseteq X_2))$
$\Rightarrow$ { I.H. }
$\exists(X_1 : mfs\ (c\ A).\ X_2 : mfs\ (c\ A)\ .\ X_1 \subseteq X \subseteq X_2)$
$\Rightarrow$ { Definition of mfs }
$\exists(K_1 : c\ A.\ K_2 : c\ A\ .$
$\exists(X_1 : mfs\ (\iota\ K_1).\ X_2 : mfs\ (\iota\ K_2)\ .\ X_1 \subseteq X \subseteq X_2))$
$\Rightarrow$ { $X_1 \subseteq X_2$ and set theory }
$\exists(K_1 : c\ A.\ K_2 : c\ A\ .\ \exists(X_1 : mfs\ (\iota\ K_1), X_2 : mfs\ (\iota\ K_2)\ .$
$\exists(K : c\ A\ .\ K_1 \subseteq K \subseteq (K_1 \cup K_2) \wedge X \in mfs\ (\iota\ K))))$
$\Rightarrow$ { $K \in c\ A$ }
$X \in mfs\ (c\ A)$

Note at derivation step marked 1) in the above proof.
This step can be done because convex closure. If $Y_1 \in c\ (mfs\ A)$ and $Y_2 \in c\ (mfs\ A)$ then also all Y_1' s.t. $Y_1 \subseteq Y_1' \subseteq Y_2$ in $c\ (mfs\ A)$. So we can take a bigger Y_1' inbetween Y_1 and Y as a lowerbound of Y. This is needed in case Y itself is not functional. From this we can get an $X_1 \in mfs\ Y_1'$ s.t. $X_1 \subseteq X$.

□

Lemma 5.4.3
For all sets a and b

$$X \in c\ (\{a, b\}) \Rightarrow a \subseteq X \vee b \subseteq X$$

Proof
The proof is by induction to the length of the proof of $X \in c\ (\{a, b\})$, leading to three cases.
Case a) $X \in \{a, b\}$, so $X = a$ or $X = b$ and this, trivially, ends the proof of this case.
Case b)

$$\begin{array}{ll} & X = X_1 \cup X_2 \wedge X_1 \in c\ (\{a, b\}) \wedge X_2 \in c\ (\{a, b\}) \\ \Rightarrow & \quad \{\text{I.H.and logics}\} \\ & (a \subseteq X_1 \vee a \subseteq X_2) \vee \\ & (a \subseteq X_1 \vee b \subseteq X_2) \vee \\ & (b \subseteq X_1 \vee a \subseteq X_2) \vee \\ & (b \subseteq X_1 \vee b \subseteq X_2) \\ \Rightarrow & \quad \{\text{Set theory}\} \\ & a \subseteq X_1 \cup X_2 \vee \\ & a \cup b \subseteq X_1 \cup X_2 \vee \\ & b \subseteq X_1 \cup X_2 \\ \equiv & \quad \{X = X_1 \cup X_2\} \\ & a \subseteq X \vee a \cup b \subseteq X \vee b \subseteq X \\ \Rightarrow & \quad \{\text{Set theory}\} \\ & a \subseteq X \vee b \subseteq X \end{array}$$

Case c)

$$\begin{array}{ll} & X_1 \subseteq X \subseteq X_2 \wedge X_1 \in c\ (\{a, b\}) \wedge X_2 \in c\ (\{a, b\}) \\ \Rightarrow & \quad \{\text{I.H.}\} \end{array}$$

$$
\begin{array}{ll}
 & (a \subseteq X_1 \vee b \subseteq X_1) \wedge \\
 & (a \subseteq X_2 \vee b \subseteq X_2) \\
\Rightarrow & \quad \{ X_1 \subseteq X \subseteq X_2 \} \\
 & a \subseteq X \vee b \subseteq X
\end{array}
$$

□

Lemma 5.4.4
For sets a and b and for all X

$$X \in c\ (\{a, b\}) \Rightarrow X \subseteq a \cup b$$

Proof
Proof by induction to the length of the proof of $X \in c\ (\{a, b\})$ leading to three cases.
Case a)
$X \in \{a, b\}$ implies $X = a$ or $X = b$, and thus trivially $X \subseteq a \cup b$.
Case b)

$$
\begin{array}{ll}
 & X = X_1 \cup X_2 \wedge X_1 \in c\ (\{a, b\}) \wedge X_2 \in c\ (\{a, b\}) \\
\Rightarrow & \quad \{ \text{I.H.} \} \\
 & X_1 \subseteq a \cup b \wedge X_2 \subseteq a \cup b \\
\Rightarrow & \quad \{ \text{Set theory} \} \\
 & X_1 \cup X_2 \subseteq a \cup b \\
\equiv & \quad \{ \text{Definition of of } X \} \\
 & X \subseteq a \cup b
\end{array}
$$

Case c)

$$
\begin{array}{ll}
 & X_1 \subseteq X \subseteq X_2 \wedge X_1 \in c\ (\{a, b\}) \wedge X_2 \in c\ (\{a, b\}) \\
\Rightarrow & \quad \{ \text{I.H.} \} \\
 & X_1 \subseteq X \subseteq X_2 \wedge X_1 \subseteq a \cup b \wedge X_2 \subseteq a \cup b \\
\Rightarrow & \quad \{ \text{Set theory} \} \\
 & X \subseteq X_2 \subseteq a \cup b \\
\Rightarrow & \quad \{ \text{Set theory} \} \\
 & X \subseteq a \cup b
\end{array}
$$

□

Lemma 5.4.5 *Function tac is injective*
For all f and g finite *MSPF*

$$tree\ f = tree\ g \Rightarrow f = g$$

Proof
Trivial from the construction of trees.

□

Lemma 5.4.6
For all p and q in PA_{io}

$$\widetilde{tree}\ (p \parallel q) = \widetilde{tree}\ p +_{DFT} \widetilde{tree}\ q$$

Proof

$$\begin{array}{ll} & d \in \widetilde{tree}\ (p \parallel q) \\ \equiv & \quad \{ \text{Definition of } \widetilde{tree} \} \\ & \exists(f \cdot (p \parallel q)\ f \wedge d = tree\ f) \\ \equiv & \quad \{ \text{Definition of } \parallel \text{ on specifications} \} \\ & \exists(f \cdot \exists((f_1, f_2) \cdot p\ f_1 \wedge q\ f_2 \wedge (f_1 \parallel f_2)\ f \wedge d = tree\ f)) \\ \equiv & \quad \{ \text{Definition of } \parallel \text{ on functions} \} \\ & \exists(f \cdot \exists((f_1, f_2) \cdot p\ f_1 \wedge q\ f_2 \wedge \\ & tree\ f \in dec\ (tree\ f_1 +_{IFAT} tree\ f_2) \wedge d = tree\ f)) \\ \equiv & \quad \{ \text{Logic} \} \\ & \exists((f_1, f_2) \cdot p\ f_1 \wedge q\ f_2 \wedge d \in dec\ (tree\ f_1 +_{IFAT} tree\ f_2)) \\ \equiv & \quad \{ \text{Logic} \} \\ & \exists(d_1, d_2, f_1, f_2 \cdot p\ f_1 \wedge q\ f_2 \wedge d_1 = tree\ f_1 \wedge d_2 = tree\ f_2 \wedge \\ & d \in dec\ (d_1 +_{IFAT} d_2)) \\ \equiv & \quad \{ \text{Definition of } \widetilde{tree} \} \\ & \exists(d_1, d_2 \cdot d_1 \in (\widetilde{tree}\ p) \cdot (\widetilde{tree}\ q) \wedge d \in (d_1 +_{IFAT} d_2)) \\ \equiv & \quad \{ \text{Definition of } \cdot \} \\ & d \in (\widetilde{tree}\ p) +_{DFT} (\widetilde{tree}\ q) \end{array}$$

□

Lemma 5.4.7
For all Ψ and Φ sets of sets of tuples such that they have the same labels and x is functional

$$z \in c\ \Phi \wedge y \in mfs\ (\iota\ z) \wedge y \subseteq x$$
$$\Rightarrow$$
$$x \cup z \in c\ \Phi \wedge x \in mfs\ (\iota\ (x \cup z))$$

Proof
First we prove that $x \cup z \in c\ \Phi$. We know that $z \in c\ \Phi$ and $z \subseteq x \cup z$. Moreover, from a property of c we know that $\bigcup \Phi \in c\ \Phi$ ($\bigcup$ over a set of sets is the union of all the sets this set contains). Also clearly $\bigcup \Phi = labels\ \Psi$. So:

$$\begin{array}{ll} & x \subseteq labels\ \Psi \\ \Rightarrow & \quad \{\ labels\ \Psi = labels\ \Phi\ \} \\ & x \subseteq labels\ \Phi \\ \Rightarrow & \quad \{\ labels\ \Phi = \bigcup \Phi\ \} \\ & x \subseteq \bigcup \Phi \\ \Rightarrow & \quad \{\ z \in c\ \Phi\ \} \\ & x \subseteq \bigcup \Phi \wedge z \in c\ \Phi \\ \Rightarrow & \quad \{\ z \in c\ \Phi \Rightarrow z \subseteq \bigcup \Phi\ \} \\ & x \cup z \subseteq \bigcup \Phi \\ \Rightarrow & \quad \{\ z \subseteq x \cup z\ \} \\ & z \subseteq (x \cup z) \wedge (x \cup z) \subseteq \bigcup \Phi \\ \Rightarrow & \quad \{\ \text{Definition of } c, \text{ convex closure}\ \} \\ & x \cup z \in c\ \Phi \end{array}$$

Now we prove that $x \in mfs\ (\iota\ (x \cup z))$. The proof is by derivation of a contradiction. Suppose $x \notin mfs\ (\iota\ (x \cup z))$.

$$\begin{array}{ll} & x \notin mfs\ (\iota\ (x \cup z)) \\ \Rightarrow & \quad \{\ \text{Definition of } mfs\ \} \\ & \exists (k : \mathcal{P}\ (x \cup z)\ .\ \iota\ \subseteq k \wedge func\ k) \\ \Rightarrow & \quad \{\ \text{Set theory}\ \} \\ & \exists (a : z\ .\ a \notin \iota\ \wedge\ a \in k) \\ \Rightarrow & \quad \{\ y \in mfs\ \ldots\ \wedge\ \iota \cap z = y \text{ see note below}\ \} \\ & k \cap z \supseteq y \cup \{a\} \\ \Rightarrow & \quad \{\ \text{Logic and } y \text{ is maximal functional subset}\ \} \\ & k \text{ is not } functional \end{array}$$

The fact that we derive that k is not functional is in contradiction with assumptions. So we proved $x \in mfs\ (\iota\ (x \cup z))$. Note that it cannot be

the case that in the one but last step in the proof above $y \subset (x \cap z)$. The reason is that x is functional, so also $x \cap z$ is functional. Moreover $y \subseteq x$ and $y \in mfs\ (\iota\ z)$, which means that y is maximal and thus should contain all elements in $x \cap z$. So $y = x \cap z$.

□

With the extra lemmas proven above we can give a formal proof of the lemmas stated in previous sections of this chapter.

Lemma 5.1.7
Given t' and t'' in $ffAT$ then:

i $t' \leq_{ffAT} t'' \Rightarrow \forall((m,\sigma) : (M \times M^*)\ .\ (m,\sigma)_{ffAT}\ t' \leq_{ffAT} (m,\sigma)_{ffAT}\ t'')$

ii $t' \leq_{ffAT} t'' \Rightarrow \forall(t : ffAT\ .\ t +_{ffAT} t' \leq_{ffAT} t +_{ffAT} t'')$

Proof

Case i:
This case can be proven in two subcases:

$a)\ L_{ffAT}\ (m,\sigma)_{ffAT}\ t' = L_{ffAT}\ (m,\sigma)_{ffAT}\ t''$
$b)\ \forall(\pi \in L_{ffAT}\ .\ \mathcal{A}_{ffAT}\ ((m,\sigma)_{ffAT}\ t'')\ \pi \subseteq \mathcal{A}_{ffAT}\ ((m,\sigma)_{ffAT}\ t')\ \pi)$

Case i.a:

$$
\begin{array}{ll}
 & L_{ffAT}\ ((m,\sigma)_{ffAT}\ t') \\
= & \quad \{\text{Definition of } (m,\sigma)_{ffAT}\ t'\} \\
 & \iota\ \varepsilon \cup \{(m,\sigma) \succ \pi\ .\ \pi \in L_{ffAT}\ t'\} \\
= & \quad \{t' \leq_{ffAT} t'' \Rightarrow L_{ffAT}\ t' = L_{ffAT}\ t''\} \\
 & \iota\ \varepsilon \cup \{(m,\sigma) \succ \pi\ .\ \pi \in L_{ffAT}\ t''\} \\
= & \quad \{\text{Definition of } (m,\sigma)_{ffAT}\ t''\} \\
 & L_{ffAT}\ ((m,\sigma)_{ffAT}\ t'')
\end{array}
$$

Case i.b:
First in case $\pi = \varepsilon$:

$$
\begin{array}{ll}
 & \mathcal{A}_{\mathit{ffAT}}\ ((m,\sigma)_{\mathit{ffAT}}\ t'')\ \varepsilon \\
= & \quad \{\text{Definition of } (m,\sigma)_{\mathit{ffAT}}\} \\
 & t\ (m,\sigma) \\
= & \quad \{\text{Definition of } (m,\sigma)_{\mathit{ffAT}}\} \\
 & \mathcal{A}_{\mathit{ffAT}}\ ((m,\sigma)_{\mathit{ffAT}}\ t')\ \varepsilon
\end{array}
$$

And in case $\pi \neq \varepsilon$:
Let $\pi = (u,\gamma) \succ \pi'$

$$
\begin{array}{ll}
 & (\pi\ 0 \neq (m,\sigma) \Rightarrow \ \mathcal{A}_{\mathit{ffAT}}\ ((m,\sigma)_{\mathit{ffAT}}\ t'')\ \pi = \emptyset \wedge \\
 & \qquad\qquad\qquad\qquad \mathcal{A}_{\mathit{ffAT}}\ ((m,\sigma)_{\mathit{ffAT}}\ t')\ \pi = \emptyset) \wedge \\
 & (\pi\ 0 = (m,\sigma) \Rightarrow \ \mathcal{A}_{\mathit{ffAT}}\ ((m,\sigma)_{\mathit{ffAT}}\ t'')\ \pi = \ \mathcal{A}_{\mathit{ffAT}}\ t''\ \pi' \wedge \\
 & \qquad\qquad\qquad\qquad \mathcal{A}_{\mathit{ffAT}}\ ((m,\sigma)_{\mathit{ffAT}}\ t')\ \pi = \ \mathcal{A}_{\mathit{ffAT}}\ t'\ \pi') \\
\Rightarrow & \quad \{\ t' \leq_{\mathit{ffAT}}\ t'' \Rightarrow \ \mathcal{A}_{\mathit{ffAT}}\ t''\ \pi' \subseteq \ \mathcal{A}_{\mathit{ffAT}}\ t'\ \pi'\} \\
 & \pi\ 0 \neq (m,\sigma) \Rightarrow \ \mathcal{A}_{\mathit{ffAT}}\ ((m,\sigma)_{\mathit{ffAT}}\ t'')\ \pi = \ \mathcal{A}_{\mathit{ffAT}}\ ((m,\sigma)_{\mathit{ffAT}}\ t')\ \pi \wedge \\
 & \pi\ 0 = (m,\sigma) \Rightarrow \ \mathcal{A}_{\mathit{ffAT}}\ ((m,\sigma)_{\mathit{ffAT}}\ t'')\ \pi \subseteq \ \mathcal{A}_{\mathit{ffAT}}\ ((m,\sigma)_{\mathit{ffAT}}\ t')\ \pi \\
\Rightarrow & \quad \{\text{Set theory}\} \\
 & \forall \pi : (\ L_{\mathit{ffAT}}\ t'')\ \backslash\ t\ \varepsilon\ \ \mathcal{A}_{\mathit{ffAT}}\ ((m,\sigma)_{\mathit{ffAT}}\ t'')\ \pi \subseteq \ \mathcal{A}_{\mathit{ffAT}}\ ((m,\sigma)_{\mathit{ffAT}}\ t')\ \pi
\end{array}
$$

Case ii:
This case can be proven in two subcases:

$$
\begin{array}{l}
a)\ \ L_{\mathit{ffAT}}\ (t +_{\mathit{ffAT}}\ t') = \ L_{\mathit{ffAT}}\ (t +_{\mathit{ffAT}}\ t'') \\
b) \forall (\pi :\ L_{\mathit{ffAT}}\ (t +_{\mathit{ffAT}}\ t'')\ .\ \ \mathcal{A}_{\mathit{ffAT}}\ (t +_{\mathit{ffAT}}\ t'')\ \pi \subseteq \ \mathcal{A}_{\mathit{ffAT}}\ (t +_{\mathit{ffAT}}\ t')\ \pi)
\end{array}
$$

Case ii.a:

$$
\begin{array}{ll}
 & L_{\mathit{ffAT}}\ (t +_{\mathit{ffAT}}\ t') \\
= & \quad \{\text{Definition of } +_{\mathit{ffAT}}\} \\
 & L_{\mathit{ffAT}}\ t \cup \ L_{\mathit{ffAT}}\ t' \\
= & \quad \{\ t' \leq_{\mathit{ffAT}}\ t'' \Rightarrow \ L_{\mathit{ffAT}}\ t' = \ L_{\mathit{ffAT}}\ t''\} \\
 & L_{\mathit{ffAT}}\ t \cup \ L_{\mathit{ffAT}}\ t'' \\
= & \quad \{\text{Definition of } +_{\mathit{ffAT}}\} \\
 & L_{\mathit{ffAT}}\ (t +_{\mathit{ffAT}}\ t'')
\end{array}
$$

Case ii.b:
First the case that $\pi = \varepsilon$:

$$
\begin{array}{ll}
 & \mathcal{A}_{\mathit{IIAT}}\ (t +_{\mathit{IIAT}} t'')\ \varepsilon \\
= & \quad \{\text{ Definition of } +_{\mathit{IIAT}} \} \\
 & \mathcal{A}_{\mathit{IIAT}}\ t\ \varepsilon \mathbin{\textcircled{u}} \mathcal{A}_{\mathit{IIAT}}\ t''\ \varepsilon \\
\subseteq & \quad \{\text{ Definition of } \textcircled{u},\ \mathit{mfs} \text{ monotonic w.r.t. } \subseteq \\
 & \text{and } \mathcal{A}_{\mathit{IIAT}}\ t''\ \varepsilon \subseteq \mathcal{A}_{\mathit{IIAT}}\ t'\ \varepsilon \} \\
 & \mathcal{A}_{\mathit{IIAT}}\ t\ \varepsilon \mathbin{\textcircled{u}} \mathcal{A}_{\mathit{IIAT}}\ t'\ \varepsilon \\
= & \quad \{\text{ Definition of } +_{\mathit{IIAT}} \} \\
 & \mathcal{A}_{\mathit{IIAT}}\ (t +_{\mathit{IIAT}} t')\ \varepsilon
\end{array}
$$

The case that $\pi \neq \varepsilon$:

$$
\begin{array}{ll}
 & \mathcal{A}_{\mathit{IIAT}}\ (t +_{\mathit{IIAT}} t'')\ \pi \\
= & \quad \{\text{ Definition of } +_{\mathit{IIAT}} \} \\
 & \mathit{mfs}\ (c\ (\ \mathcal{A}_{\mathit{IIAT}}\ t\ \pi \cup \mathcal{A}_{\mathit{IIAT}}\ t''\ \pi)) \\
\subseteq & \quad \{\ \mathit{mfs} \text{ and } c \text{ are monotonic w.r.t. } \subseteq \\
 & \text{and } \mathcal{A}_{\mathit{IIAT}}\ t''\ \pi \subseteq \mathcal{A}_{\mathit{IIAT}}\ t'\ \pi \} \\
 & \mathit{mfs}\ (c\ (\ \mathcal{A}_{\mathit{IIAT}}\ t\ \pi \cup \mathcal{A}_{\mathit{IIAT}}\ t'\ \pi)) \\
= & \quad \{\text{ Definition of } +_{\mathit{IIAT}} \} \\
 & \mathcal{A}_{\mathit{IIAT}}\ (t +_{\mathit{IIAT}} t')\ \pi
\end{array}
$$

Remark: The fact that *mfs* and u are monotonic w.r.t. $\subseteq$ is easy to see and the proof is left to the reader. The proof that c is monotonic w.r.t. $\subseteq$ can be found in [Hen88].

Lemma 5.1.10

$$\forall(A : \mathcal{P}\ \mathcal{P}\ (M \times M^*)\ \ldots\ c\ (\mathit{mfs}\ A)) = \mathit{mfs}\ (c\ A))$$

Proof

First we prove $\mathit{mfs}\ (c\ (\mathit{mfs}\ A)) \subseteq \mathit{mfs}\ (c\ A))$ which means that we have to prove that $\forall(x\ .\ x \in \mathit{mfs}\ (c\ (\mathit{mfs}\ A)) \Rightarrow x \in \mathit{mfs}\ (c\ A)))$.

$$
\begin{array}{ll}
 & x \in mfs\ (c\ (mfs\ A)) \\
\equiv & \quad \{ \text{Definition of } mfs \} \\
 & \exists (Y : c\ (mfs\ A)\ .\ x \in mfs\ (c\ Y)) \\
\Rightarrow & \quad \{ \text{Lemma 5.4.2} \} \\
 & x \in mfs\ (c\ A)
\end{array}
$$

Second we prove that $mfs\ (c\ A)) \subseteq mfs\ (c\ (mfs\ A))$. We start from the fact that $c\ A \subseteq c\ (mfs\ A)$ which is easy to prove. Then we can derive:

$$
\begin{array}{ll}
 & c\ A \subseteq c\ (mfs\ A) \\
\Rightarrow & \quad \{ mfs \text{ is monotonic w.r.t. } \subseteq \} \\
 & mfs\ (c\ A) \subseteq mfs\ (c\ (mfs\ A))
\end{array}
$$

Lemma 5.2.3 Pairwise
For all non-empty sets A and B in $\mathcal{P}\ \mathcal{P}\ (M \times M^*)$

$$c\ (A \cup B) = \bigcup((a, b) : (c\ A) \times (c\ B)\ .\ c\ (\{a, b\}))$$

Remark that $c\ (A \cup B)$ is equal to $c\ (c\ A \cup c\ B)$ because of rule c3 of properties of closure on page 53. With this we can restate the lemma as:
For all non-empty sets A and B in $\mathcal{P}\ \mathcal{P}\ (M \times M^*)$

$$X \in c\ (c\ A \cup c\ B) \equiv \exists((a, b) : (c\ A) \times (c\ B)\ .\ X \in c\ (\{a, b\}))$$

Proof

The proof is in two parts, one part for each implication, which together establish equivalence ($\equiv$).
i) Part $\Rightarrow$ is proved by induction to the length of the proof of $X \in c\ (c\ A \cup c\ B)$ and the proof consists of three parts as a consequence of the definition of c.
Take $X \in c\ (c\ A \cup c\ B)$ and it holds to the following three cases:

a) $X \in c\ A \cup c\ B$

b) $\exists((X_1, X_2) : c\ (c\ A \cup c\ B)^2\ .\ X = X_1 \cup X_2)$

c) $\exists((X_1, X_2) : c\ (c\ A \cup c\ B)^2\ .\ X_1 \subseteq X \subseteq X_2)$

a)

$$
\begin{array}{ll}
 & X \in c\ A \cup c\ B \\
\equiv & \quad \{ \text{Set theory} \} \\
 & X \in c\ A \vee X \in c\ B \\
\Rightarrow & \quad \{ \text{Definition of } c \} \\
 & (X \in c\ (\{X, b\}) \wedge b \in c\ B) \vee \\
 & (X \in c\ (\{a, X\}) \wedge a \in c\ A) \\
\Rightarrow & \quad \{ a = X \wedge b \in c\ B \text{ or } a \in c\ A \wedge b = X \} \\
 & \exists((a, b) : c\ A \times c\ B\ .\ X \in c\ (\{a, b\})) \\
\equiv & \quad \{ \text{Definition of } \bigcup \} \\
 & X \in \bigcup((a, b) : (c\ A) \times (c\ B)\ .\ c\ (\{a, b\}))
\end{array}
$$

b) $\exists((X_1, X_2) : c\ (c\ A \cup c\ B)^2\ .\ X = X_1 \cup X_2)$.

The proofs that X_1 (X_2) in $c\ (c\ A \cup c\ B)$ is shorter than $X \in c\ (c\ A \cup c\ B)$, so by induction hypothesis we get that:

$$\exists((a_1, b_1) : c\ A \times c\ B\ .\ X_1 \in c\ (\{a_1, b_1\}))$$
$$\exists((a_2, b_2) : c\ A \times c\ B\ .\ X_2 \in c\ (\{a_2, b_2\}))$$

and we have to prove that $\exists((a, b) : c\ A \times c\ B\ .\ X_1 \cup X_2 \in c\ (\{a, b\}))$.

From lemma 5.4.3, which says $Y \in c\ (\{a, b\}) \Rightarrow (a \subseteq Y \vee b \subseteq Y)$, and lemma 5.4.4 we see that we have the following subcases:

b.1) $X_1 = a_1 \cup y_1 \wedge y_1 \subseteq b_1$
$X_2 = a_2 \cup y_2 \wedge y_2 \subseteq b_2$

b.2) $X_1 = k_1 \cup b_1 \wedge k_1 \subseteq a_1$
$X_2 = k_2 \cup b_2 \wedge k_2 \subseteq a_2$

b.3) $X_1 = k_1 \cup b_1 \wedge k_1 \subseteq a_1$
$X_2 = a_2 \cup y_2 \wedge y_2 \subseteq b_2$

b.4) $X_1 = a_1 \cup y_1 \wedge y_1 \subseteq b_1$
$X_2 = k_2 \cup b_2 \wedge k_2 \subseteq$

Proof of subcase b.1):
We consider the case that

$$X_1 = a_1 \cup y_1 \wedge y_1 \subseteq b_1$$
$$X_2 = a_2 \cup y_2 \wedge y_2 \subseteq b_2$$

Let's take $a = a_1 \cup a_2$ and $b = b_1 \cup b_2$, then we have to show the following things:

- $a_1 \cup a_2 \in c\ A$

$$
\begin{array}{ll}
 & a_1 \in c\ A \wedge a_2 \in c\ A \\
\Rightarrow & \quad \{ \text{Definition of } c \} \\
 & a_1 \cup a_2 \in c\ A
\end{array}
$$

- $b_1 \cup b_2 \in c\ B$
  Similar to previous case.

- $X_1 \cup X_2 \in c\ (\{a_1 \cup a_2, b_1 \cup b_2\})$
  We show this by showing that there exist elements L and R in $c\ (\{a_1 \cup a_2, b_1 \cup b_2\})$ such that $L \subseteq X_1 \cup X_2 \subseteq R$.
  For L we have:

$$
\begin{array}{ll}
 & \exists (L : c\ (\{a_1 \cup a_2, b_1 \cup b_2\})\ .\ L \subseteq X_1 \cup X_2) \\
\Leftarrow & \quad \{ \text{Take } L = a_1 \cup a_2 \in c\ (\{a_1 \cup a_2, b_1 \cup b_2\} \} \\
 & a_1 \cup a_2 \subseteq X_1 \cup X_2 \\
\Leftarrow & \quad \{ X_1 = a_1 \cup y_1 \text{ and} \\
 & X_2 = a_2 \cup y_2 \} \\
 & a_1 \cup a_2 \subseteq a_1 \cup y_1 \cup a_2 \cup y_2 \\
\Leftarrow & \quad \{ \text{Set theory} \} \\
 & true
\end{array}
$$

  For R we have:

$$
\begin{array}{ll}
 & \exists (R : c\ (\{a_1 \cup a_2, b_1 \cup b_2\})\ .\ X_1 \cup X_2 \subseteq R) \\
\Leftarrow & \quad \{ \text{Take } R = a_1 \cup a_2 \cup b_1 \cup b_2, \text{ which is in closure} \} \\
 & X_1 \cup X_2 \subseteq a_1 \cup a_2 \cup b_1 \cup b_2 \\
\Leftarrow & \quad \{ \text{Definition of } X_1 \text{ and } X_2 \} \\
 & a_1 \cup y_1 \cup a_2 \cup y_2 \subseteq a_1 \cup a_2 \cup b_1 \cup b_2 \\
\Leftarrow & \quad \{ y_1 \subseteq b_1 \text{ and } y_2 \subseteq b_2 \} \\
 & true
\end{array}
$$

Proof of subcase b.2):
We consider the case that

$$
\begin{array}{l}
X_1 = k_1 \cup b_1 \wedge k_1 \subseteq a_1 \\
X_2 = k_2 \cup b_2 \wedge k_2 \subseteq a_2
\end{array}
$$

This case is similar to the case b.1).
Proof of subcase b.3):
We consider the case that

$$X_1 = k_1 \cup b_1 \wedge k_1 \subseteq a_1$$
$$X_2 = a_2 \cup y_2 \wedge y_2 \subseteq b_2$$

Let's take $a = a_2 \cup k_1$ and $b = b_1 \cup y_2$. Now we have to show that

$$X_1 \cup X_2 \in c\ (\{a_2 \cup k_1, b_1 \cup y_2\})$$

And this requires that we prove the following things:

- $a_2 \cup k_1 \in c\ A$

$$\begin{array}{ll} & a_1 \in c\ A \wedge a_2 \in c\ A \\ \Rightarrow & \quad \{\text{Definition of } c\text{, union-closure}\} \\ & a_1 \cup a_2 \in c\ A \\ \Rightarrow & \quad \{k_1 \subseteq a_1 \Rightarrow a_2 \subseteq a_2 \cup k_1 \subseteq a_1 \cup a_2\text{, convex closure}\} \\ & a_2 \cup k_1 \in c\ A \end{array}$$

- $b_1 \cup y_2 \in c\ B$

$$\begin{array}{ll} & b_1 \in c\ B \wedge b_2 \in c\ B \\ \Rightarrow & \quad \{\text{Definition of } c\} \\ & b_1 \cup b_2 \in c\ B \\ \Rightarrow & \quad \{y_2 \subseteq b_2 \text{ and Definition of } c\} \\ & b_1 \cup y_2 \in c\ B \end{array}$$

- $\exists((L, R) : c\ (\{a_2 \cup k_1, b_1 \cup y_2\})\ .\ L \subseteq X_1 \cup X_2 \subseteq R)$.
  Take $L = a_2 \cup k_1$

$$\begin{array}{ll} & L \subseteq X_1 \cup X_2 \\ \Leftarrow & \quad \{\text{Definition of } L,\ X_1 \text{ and } X_2\} \\ & a_2 \cup k_1 \subseteq k_1 \cup b_1 \cup a_2 \cup y_2 \\ \Leftarrow & \quad \{\text{Set theory}\} \\ & true \end{array}$$

  Take $R = a_2 \cup k_1 \cup b_1 \cup y_2$

$$
\begin{array}{ll}
 & X_1 \cup X_2 \subseteq R \\
\Leftarrow & \quad \{ \text{Definition of } R, X_1 \text{ and } X_2 \} \\
 & k_1 \cup b_1 \cup a_2 \cup y_2 \subseteq a_2 \cup k_1 \cup b_1 \cup y_2 \\
\Leftarrow & \quad \{ \text{Set theory} \} \\
 & true
\end{array}
$$

So indeed $X = X_1 \cup X_2 \in c\ (\{a_2 \cup k_1, b_1 \cup y_2\})$.

Proof of subcase b.4):
We consider the case that

$$X_1 = a_1 \cup y_1 \wedge y_1 \subseteq b_1$$
$$X_2 = k_2 \cup b_2 \wedge k_2 \subseteq a_2$$

This proof is similar to case b.3).
This ends case b).
c) $\exists((X_1, X_2) : (c\ (c\ A \cup c\ B))^2\ .\ X_1 \subseteq X \subseteq, X_2)$.

From lemma 5.4.3 we derive the same four subcases as in case b) and in the following we give a proof for each of these subcases.
Subcase c.1)

$$X_1 = a_1 \cup y_1 \wedge y_1 \subseteq b_1$$
$$X_2 = a_2 \cup y_2 \wedge y_2 \subseteq b_2$$

Define the intersection of X with a_2 so $k_2 = X \cap a_2$. We assumed that $X_1 \subseteq X \subseteq X_2$ and we prove that $X \in c\ (\{a_1 \cup k_2, b_2\})$ by showing the following things:

c.1.a $a_1 \cup k_2 \in c\ A$

c.1.b $b_2 \in c\ B$

c.1.c $\forall(X\ .\ X_1 \subseteq X \subseteq \ldots L, R) : c\ (\{a_1 \cup k_2, b_2\})^2\ .\ L \subseteq X \subseteq R))$

Proof of case c.1.a)

$$
\begin{array}{ll}
 & a_1 \in c\ A \wedge a_{\ldots} \\
\Rightarrow & \quad \{ \text{Set theory} \} \\
 & a_1 \cup a_2 \in c\ A \\
\Rightarrow & \quad \{ k_2 \subseteq a_2 \text{ and definition of } c \} \\
 & a_1 \cup k_2 \in c\ A
\end{array}
$$

Proof of case c.1.b)
Trivial.

Proof of case c.1.c)

$$
\begin{array}{ll}
 & X_1 \subseteq X \subseteq X_2 \\
\equiv & \quad \{ \text{Definition of } X_1 \text{ and } X_2 \} \\
 & a_1 \cup y_1 \subseteq X \subseteq a_2 \cup y_2 \\
\Rightarrow & \quad \{ a_1 \subseteq a_1 \cup y_1 \} \\
 & a_1 \subseteq X \subseteq a_2 \cup y_2 \\
\Rightarrow & \quad \{ y_2 \subseteq b_2 \} \\
 & a_1 \subseteq X \subseteq a_2 \cup b_2 \\
\Rightarrow & \quad \{ X \cap a_2 = k_2 \} \\
 & a_1 \cup k_2 \subseteq X \subseteq a_2 \cup b_2 \\
\equiv & \quad \{ \text{Set theory} \} \\
 & a_1 \cup k_2 \subseteq X \subseteq (X \cap a_2) \cup (a_2 \setminus X) \cup b_2 \\
\equiv & \quad \{ \text{Set theory} \} \\
 & a_1 \cup k_2 \subseteq X \subseteq (X \cap a_2) \cup b_2 \\
\equiv & \quad \{ X \cap a_2 = k_2 \text{ as before defined} \} \\
 & a_1 \cup k_2 \subseteq X \subseteq a_1 \cup k_2 \cup b_2 \\
\Rightarrow & \quad \{ a_1 \cup k_2 \text{ and } a_1 \cup k_2 \cup b_2 \text{ are in } c\ (\{a_1 \cup k_2, b_2\}) \} \\
 & X \in c\ (\{a_1 \cup k_2, b_2\})
\end{array}
$$

Subcase c.2)

$$X_1 = k_1 \cup b_1 \wedge k_1 \subseteq a_1$$
$$X_2 = k_2 \cup b_2 \wedge k_2 \subseteq a_2$$

This proof is similar to subcase c.1.

Subcase c.3)

$$X_1 = k_1 \cup b_1 \wedge k_1 \subseteq a_1$$
$$X_2 = a_2 \cup y_2 \wedge y_2 \subseteq b_2$$

Define the intersection of X with y_2: $X \cap Y_2 = z_2$, and the intersection of X with a_2: $X \cap a_2 = k_2$. We assumed $X_1 \subseteq X \subseteq X_2$ and we prove that $X \in c\ (\{a_1 \cup k_2, b_1 \cup z_2\})$. We do this by showing that:

$$
\begin{array}{ll}
 & X_1 \subseteq X \subseteq X_2 \wedge X_1 \in c\ (c\ A \cup c\ B) \wedge X_2 \in c\ (c\ A \cup c\ B) \\
\Rightarrow & \quad \{ \text{Definition of } c \} \\
 & X \in c\ (c\ A \cup c\ B)
\end{array}
$$

This ends the proof of the pairwise lemma.

Lemma 5.2.8 $\widetilde{tree}$ is injective
For all p and q in SPA_{io}

$$\widetilde{tree}\ (\ fun\ [\ p\ ]\ ) = \widetilde{tree}\ (\ fun\ [\ q\ ]\ ) \Rightarrow\ fun\ [\ p\ ]\ =\ fun\ [\ q\ ]$$

Proof

$$
\begin{array}{ll}
 & f \in\ fun\ [\ p\ ] \\
\Rightarrow & \quad \{ \text{Definition of } tree \} \\
 & \exists (d : DFT\ .\ d = tree\ f) \wedge f \in\ fun\ [\ p\ ] \\
\Rightarrow & \quad \{ \text{Definition of } \widetilde{tree} \} \\
 & \exists (d : DFT\ .\ d = tree\ f) \wedge f \in\ fun\ [\ p\ ] \wedge d \in \widetilde{tree}\ \ fun\ [\ p\ ] \\
\equiv & \quad \{\ \widetilde{tree}\ \ fun\ [p]\ = \widetilde{tree}\ \ fun\ [q]\ \} \\
 & \exists (d : DFT\ .\ d = tree\ f) \wedge f \in\ fun\ [\ p\ ] \wedge d \in \widetilde{tree}\ \ fun\ [\ q\ ] \\
\Rightarrow & \quad \{ \text{Definition of } \widetilde{tree} \} \\
 & \exists (d : DFT\ .\ d = tree\ f) \wedge f \in\ fun\ [\ p\ ] \wedge \\
 & \exists (g\ .\ \ fun\ [\ q\ ]\ \ g \wedge d = tree\ g) \\
\Rightarrow & \quad \{ tree \text{ is injective, lemma 5.4.5} \} \\
 & f \in\ fun\ [\ q\ ]
\end{array}
$$

Similarly if $f \in\ fun\ [\ \ ]$

Lemma 5.2.6
For all p_1 and p_2 in SPA

$$
\begin{array}{l}
dec\ (PffAT\ [\ p_1] \rightarrow_{AT}\ PffAT\ [\ p_2]) \\
= \\
(dec\ PffAT\ [\ p_1]) \rightarrow_{DFT} (dec\ PffAT\ [\ p_2])
\end{array}
$$

Proof

The proof is in two parts. First the direction $\Leftarrow$ is proven, then the direction $\Rightarrow$.

i) Direction $\Leftarrow$. We prove:

$$
\begin{array}{l}
(dec\ PffAT\ [\ p_1]) +_{DFT} (dec\ PffAT\ [\ p_2]) \\
\subseteq \\
dec\ (PffAT\ [\ p_1] +_{ffAT} PffAT\ [\ p_2])
\end{array}
$$

So

$$
\begin{array}{ll}
 & d \in ((dec\ PffAT\ [\ p_1]) +_{DFT} (dec\ PffAT\ [\ p_2])) \\
\equiv & \quad \{ \text{Definition of } +_{DFT} \} \\
 & d \in dec\ (d_1 +_{ffAT} d_2) \\
\equiv & \quad \{ \text{Definition of } dec \} \\
 & \forall(\pi :\ L_{ffAT}\ d\ .\ S_{ffAT}\ d\ \pi \in\ \mathcal{A}_{ffAT}\ (d_1 +_{ffAT} d_2)\ \pi) \\
\equiv & \quad \{ \text{Definition of } \mathcal{A}_{ffAT} \} \\
 & \forall(\pi :\ L_{ffAT}\ d\ .\ (\pi = \varepsilon \Rightarrow\ S_{ffAT}\ d\ \varepsilon \in\ \mathcal{A}_{ffAT}\ d_1\ \varepsilon\ \textcircled{u}\ \mathcal{A}_{ffAT}\ d_2\ \varepsilon) \wedge \\
 & \qquad\qquad (\pi \in (\ L_{ffAT}\ d) \setminus (\iota\ \varepsilon) \Rightarrow \\
 & \qquad\qquad\ S_{ffAT}\ d\ \pi \in mfs\ (c\ (\ \mathcal{A}_{ffAT}\ d_1\ \pi \cup\ \mathcal{A}_{ffAT}\ d_2\ \pi)))) \\
\equiv & \quad \{ \pi \in\ L_{ffAT}\ d \Rightarrow \neg(\ \mathcal{A}_{ffAT}\ d_1\ \pi = \emptyset \wedge\ \mathcal{A}_{ffAT}\ d_2\ \pi = \emptyset) \} \\
 & \forall(\pi :\ L_{ffAT}\ d\ .\ (\pi = \varepsilon \Rightarrow\ S_{ffAT}\ d\ \varepsilon \in\ \mathcal{A}_{ffAT}\ d_1\ \varepsilon\ \textcircled{u}\ \mathcal{A}_{ffAT}\ d_2\ \varepsilon) \\
 & \wedge \\
 & (\\
 & (\ \mathcal{A}_{ffAT}\ d_1\ \pi \neq \emptyset \wedge\ \mathcal{A}_{ffAT}\ d_2\ \pi = \emptyset \wedge \\
 & S_{ffAT}\ d\ \pi \in\ \mathcal{A}_{ffAT}\ d_1\ \pi \wedge \pi \in\ L_{ffAT}\ d_1) \\
 & \vee \\
 & (\ \mathcal{A}_{ffAT}\ d_1\ \pi = \emptyset \wedge\ \mathcal{A}_{ffAT}\ d_2\ \pi \neq \emptyset \\
 & \wedge\ S_{ffAT}\ d\ \pi \in\ \mathcal{A}_{ffAT}\ d_2\ \pi \in\ L_{ffAT}\ d_2) \\
 & \vee \\
 & (\ \mathcal{A}_{ffAT}\ d_1\ \pi \neq \emptyset \wedge\ \mathcal{A}_{ffAT}\ d_2\ \pi \neq \emptyset \\
 & \wedge\ S_{ffAT}\ d\ \pi \in mfs\ (c\ (\ \mathcal{A}_{ffAT}\ d_1\ \pi \cup\ \mathcal{A}_{ffAT}\ d_2\ \pi)) \wedge \\
 & \pi \in\ L_{ffAT}\ d_1 \cap\ L_{ffAT}\ d_2)) \\
 &)
\end{array}
$$

$\Rightarrow$ { Cases a to d below and definition of dec }
$d \in dec\ (PffAT\ [\ p_1] +_{ffAT}\ PffAT\ [\ p_2])$

Case a)

$$\pi = \varepsilon \Rightarrow\ S_{ffAT}\ d\ \varepsilon \in\ \mathcal{A}_{ffAT}\ d_1\ \varepsilon\ \textcircled{u}\ \mathcal{A}_{ffAT}\ d_2\ \varepsilon \Rightarrow$$
$$S_{ffAT}\ d\ \varepsilon \in\ \mathcal{A}_{ffAT}\ (A_1 +_{ffAT}\ A_2)\ \varepsilon$$

Proof:

$\pi = \varepsilon \Rightarrow\ S_{ffAT}\ d\ \varepsilon \in\ \mathcal{A}_{ffAT}\ d_1\ \varepsilon\ \textcircled{u}\ \mathcal{A}_{ffAT}\ d_2\ \varepsilon$
$\Rightarrow$ { Logic }
$S_{ffAT}\ d\ \varepsilon \in\ \mathcal{A}_{ffAT}\ d_1\ \varepsilon\ \textcircled{u}\ \mathcal{A}_{ffAT}\ d_2\ \varepsilon$
$\Rightarrow$ { $d_j \in dec\ (PffAT\ [\ p_j])$ and
$d_j \in dec\ (PffAT\ [\ p_j]) \Rightarrow\ S_{ffAT}\ d_j\ \varepsilon =\ S_{ffAT}\ (PffAT\ [\ p_j])\ \varepsilon$ }
$S_{ffAT}\ d\ \varepsilon \in (\iota\ S_{ffAT}\ (PffAT\ [\ p_1])\ \varepsilon)\ \textcircled{u}\ (\iota\ S_{ffAT}\ (PffAT\ [\ p_2])\ \varepsilon)$
$\Rightarrow$ { Definition of $+_{ffAT}$ }
$S_{ffAT}\ d\ \varepsilon \in\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_1] +_{ffAT}\ PffAT\ [\ p_2])\ \varepsilon$

Case b) If $\mathcal{A}_{ffAT}\ d_1\ \pi \neq \emptyset$ and $\mathcal{A}_{ffAT}\ d_2\ \pi = \emptyset$ then

$$S_{ffAT}\ d\ \pi \in\ \mathcal{A}_{ffAT}\ d_1\ \pi \wedge \pi \in\ L_{ffAT}\ d_1 \Rightarrow$$
$$S_{ffAT}\ d\ \pi \in mfs\ (c\ (\ \mathcal{A}_{ffAT}\ A_1\ \pi \cup\ \mathcal{A}_{ffAT}\ A_2\ \pi))$$

Proof:

$S_{ffAT}\ d\ \pi \in\ \mathcal{A}_{ffAT}\ d_1\ \pi \wedge \pi \in\ L_{ffAT}\ d_1$
$\Rightarrow$ { Logic }
$S_{ffAT}\ d\ \pi \in\ \mathcal{A}_{ffAT}\ d_1\ \pi$
$\Rightarrow$ { $d_1 \in DET$ }
$S_{ffAT}\ d\ \pi \in$ [illegible]
$\Rightarrow$ { $d_1 \in dec$ [illegible] and
$d_1 \in dec\ (PffAT\ [\ p_1]) \Rightarrow\ S_{ffAT}\ d_1\ \pi \in\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_1])\ \pi$ }
$S_{ffAT}\ d\ \pi \in$ [illegible]
$\Rightarrow$ { $d \in DET$ [illegible]
$S_{ffAT}\ d\ \pi \in mfs$ [illegible] $\mathcal{A}_{ffAT}\ (PffAT\ [\ p_1])\ \pi))$
$\Rightarrow$ { Monotonicity of c and mfs }
$S_{ffAT}\ d\ \pi \in mfs$ [illegible] $\mathcal{A}_{ffAT}\ (PffAT\ [\ p_1])\ \pi \cup\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_2])\ \pi))$

Case c) If $\mathcal{A}_{ffAT}\ d_1\ \pi = \emptyset$ and $\mathcal{A}_{ffAT}\ d_2\ \pi \neq \emptyset$ then

$$S_{ffAT}\ d\ \pi \in \mathcal{A}_{ffAT}\ d_2\ \pi \wedge \pi \in L_{ffAT}\ d_2 \Rightarrow$$
$$S_{ffAT}\ d\ \pi \in mfs\ (c\ (\mathcal{A}_{ffAT}\ A_1\ \pi \cup \mathcal{A}_{ffAT}\ A_2\ \pi))$$

Proof: Similar to proof of case b).

Case d) If $\mathcal{A}_{ffAT}\ d_1 \pi \neq \emptyset$ and $\mathcal{A}_{ffAT}\ d_2\ \pi \neq \emptyset$ then

$$S_{ffAT}\ d\ \pi \in mfs\ (c\ (\mathcal{A}_{ffAT}\ d_1\ \pi \cup \mathcal{A}_{ffAT}\ d_2\ \pi)) \wedge \pi \in L_{ffAT}\ d_1 \cap L_{ffAT}\ d_2 \Rightarrow$$
$$S_{ffAT}\ d\ \pi \in mfs\ (c\ (\mathcal{A}_{ffAT}\ A_1\ \pi \cup \mathcal{A}_{ffAT}\ A_2\ \pi))$$

Proof:

$$\mathcal{A}_{ffAT}\ d_1\pi \neq \emptyset \wedge\ \mathcal{A}_{ffAT}\ d_2\pi \neq \emptyset \wedge$$
$$S_{ffAT}\ d\ \pi \in mfs\ (c\ (\mathcal{A}_{ffAT}\ d_1\pi \cup \mathcal{A}_{ffAT}\ d_2\pi)) \wedge \pi \in L_{ffAT}\ d_1 \cap L_{ffAT}\ d_2$$
$\Rightarrow$ { Logics }
$$S_{ffAT}\ d\ \pi \in mfs\ (c\ (\mathcal{A}_{ffAT}\ d_1\pi \cup \mathcal{A}_{ffAT}\ d_2\pi))$$
$\Rightarrow$ { d_1 and d_2 in DFT }
$$S_{ffAT}\ d\ \pi \in mfs\ (c\ \{S_{ffAT}\ d_1\ \pi,\ S_{ffAT}\ d_2\ \pi\})$$
$\Rightarrow$ { $d_j \in dec\ (PffAT\ [\![p_j]\!])$ }
$$S_{ffAT}\ d\ \pi \in mfs\ (c\ \{S_{ffAT}\ d_1\ \pi,\ S_{ffAT}\ d_2\ \pi\}) \wedge$$
$$S_{ffAT}\ d_1\ \pi \in \mathcal{A}_{ffAT}\ (PffAT\ [\![p_1]\!])\ \pi \wedge\ S_{ffAT}\ d_2\pi \in \mathcal{A}_{ffAT}\ (PffAT\ [\![p_2]\!])\ \pi$$
$\Rightarrow$ { Lemma pairwise 5.2.3 }
$$S_{ffAT}\ d\ \pi \in mfs\ (c\ (\mathcal{A}_{ffAT}\ (PffAT\ [\![p_1]\!])\ \pi \cup \mathcal{A}_{ffAT}\ (PffAT\ [\![p_2]\!])\ \pi))$$
$\Rightarrow$ { Definition of $+_{ffAT}$ }
$$S_{ffAT}\ d\ \pi \in mfs\ (c\ (\mathcal{A}_{ffAT}\ (PffAT\ [\![p_1]\!] +_{ffAT} PffAT\ [\![p_2]\!])\ \pi))$$
$\Rightarrow$ { Definition of dec }
$$d \in dec\ (PffAT\ [\![p_1]\!] +_{ffAT} PffAT\ [\![p_2]\!])$$

ii) Direction $\Rightarrow$.

Now we prove that:

$$dec\ (PffAT\ [\![p_1]\!] +_{ffAT} PffAT\ [\![p_2]\!])$$
$$\subseteq$$
$$dec\ (PffAT\ [\![p_1]\!]) +_{ffAT} dec\ (PffAT\ [\![p_2]\!])$$
$$=$$
$$\{d\ .\ \exists(d_1 : dec\ (PffAT\ [\![p_1]\!]).\ d_2 : dec\ (PffAT\ [\![p_2]\!])\ .\ d \in dec\ (d_1 +_{ffAT} d_2))\}$$

We first elaborate some preparations for the proof.
For the ε case:

$$\begin{array}{l}\forall(S : (\ \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\ p_1])\ \varepsilon\ \textcircled{u}\ \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\ p_2])\ \varepsilon)) \cdot \\ \exists(K_1 :\ \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\ p_1])\ \varepsilon, K_2 :\ \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\ p_2])\ \varepsilon \cdot \\ S \in (\iota\ K_1)\ \textcircled{u}\ (\iota\ K_2)))\end{array}$$

From lemma 5.2.4 we know that:

$$\begin{array}{l}\forall(\pi : (M \times M^*)^\omega \setminus \iota\ \varepsilon \cdot \\ \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\ p_1])\ \pi \neq \emptyset \wedge\ \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\ p_2])\ \pi \neq \emptyset \\ \Rightarrow \\ \forall(S : \mathit{mfs}\ (\iota\ (\ \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\ p_1])\ \pi \cup\ \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\ p_2])\ \pi) \cdot \\ \exists(K_1 :\ \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\ p_1])\ \pi, K_2 :\ \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\ p_2])\ \cdot \\ S \in \mathit{mfs}\ (\iota\ \{K_1, K_2\})))\end{array}$$

Given π under the above conditions, w.l.g. we can choose particular K_1 and K_2 and we call them K_1^π and K_2^π. Define d_1 as follows for $d \in \mathit{dec}\ (\mathit{PffAT}\ [\ p_1] +_{\mathit{ffAT}}\ \mathit{PffAT}\ [\ p_2])$. $L_{\mathit{ffAT}}\ d_1$ is the smallest subset of $L_{\mathit{ffAT}}\ d \cap\ L_{\mathit{ffAT}}\ \mathit{PffAT}\ [\ p_1])$ such that:

- $\varepsilon \in\ L_{\mathit{ffAT}}\ d_1$
- $\pi \in\ L_{\mathit{ffAT}}\ d_1 \wedge\ \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\ p_2])\pi = \emptyset \wedge$
  $(m, s) \in\ S_{\mathit{ffAT}}\ d\ \pi \Rightarrow \pi \prec (m, s) \in\ L_{\mathit{ffAT}}\ d_1$
- $\pi \in\ L_{\mathit{ffAT}}\ d_1 \wedge\ \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\ p_2])\pi \neq \emptyset \wedge$
  $(m, s) \in K_1^\pi \Rightarrow \pi \prec (m, s) \in\ L_{\mathit{ffAT}}\ d_1$

Notice that for all $\tau \in\ L_{\mathit{ffAT}}\ d_1$ we have $\mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\ p_1])\ \tau \neq \emptyset$ since $\tau \in\ L_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\ p_1])$. And also $\tau \in\ L_{\mathit{ffAT}}\ d$ so that $S_{\mathit{ffAT}}\ d\ \tau$ is defined. $L_{\mathit{ffAT}}\ d_1$ is prefix closed by construction. Moreover, $d_1 \in \mathit{dec}\ (\mathit{PffAT}\ [\ p_1])$, i.e. $\forall(\tau :\ L_{\mathit{ffAT}}\ d_1 \cdot\ S_{\mathit{ffAT}}\ d_1\ \tau \in\ \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\ p_1])\ \tau)$.

$$\begin{array}{ll} & \mathit{true} \\ \Rightarrow & \quad \{\text{Definition of } S_{\mathit{ffAT}}\} \\ & \forall(\tau :\ L_{\mathit{ffAT}}\ d_1 \cdot\ S_{\mathit{ffAT}}\ d_1\ \tau = \{(m, s) \cdot \tau \prec (m, s) \in\ L_{\mathit{ffAT}}\ d_1\}) \\ \equiv & \quad \{\text{Predicate calculus}\} \\ & \forall(\tau :\ L_{\mathit{ffAT}}\ d_1 \cdot\ S_{\mathit{ffAT}}\ d_1\tau = \{(m, s) \cdot \tau \prec (m, s) \in\ L_{\mathit{ffAT}}\ d_1\} \wedge \\ & (\ \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\ p_2])\ \tau = \emptyset \vee\ \mathcal{A}_{\mathit{ffAT}}\ (\mathit{PffAT}\ [\ p_2])\ \tau \neq \emptyset)) \\ \equiv & \quad \{\text{Predicate calculus}\} \end{array}$$

$$
\begin{array}{l}
\forall(\tau:\ L_{ffAT}\ d_1\ .\ (\ S_{ffAT}\ d_1\tau = \{(m,s)\ .\ \tau \prec (m,s) \in\ L_{ffAT}\ d_1\} \wedge \\
\mathcal{A}_{ffAT}\ (PffAT\ [\ p_2])\ \tau = \emptyset) \\
\vee \\
(\ S_{ffAT}\ d_1\tau = \{(m,s)\ .\ \tau \prec (m,s) \in\ L_{ffAT}\ d_1\} \wedge \\
\mathcal{A}_{ffAT}\ (PffAT\ [\ p_2])\ \tau \neq \emptyset)) \\
\Rightarrow \qquad \{\text{Two cases ii.a and ii.b below}\} \\
S_{ffAT}\ d_1\tau \in\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_1])\ \tau
\end{array}
$$

Case ii.a)

$$
\begin{array}{l}
S_{ffAT}\ d_1\ \tau = \{(m,s)\ .\ \tau \prec (m,s) \in\ L_{ffAT}\ d_1\} \wedge\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_2])\ \tau = \emptyset \\
\Rightarrow \qquad \{\text{Definition of } d_1\} \\
S_{ffAT}\ d_1\ \tau = S\ d\ \tau \wedge\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_2])\ \tau = \emptyset \\
\Rightarrow \qquad \{\ d \in dec\ (PffAT\ [\ p_1] +_{ffAT}\ PffAT\ [\ p_2])\ \} \\
S_{ffAT}\ d_1\ \tau \in\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_1])\ \tau
\end{array}
$$

Case ii.b)

$$
\begin{array}{l}
S_{ffAT}\ d_1\ \tau = \{(m,s)\ .\ \tau \prec (m,s) \in\ L_{ffAT}\ d_1\} \wedge \\
\mathcal{A}_{ffAT}\ (PffAT\ [\ p_2])\ \tau \neq \emptyset \\
\Rightarrow \qquad \{\text{Definition of } d_1\} \\
S_{ffAT}\ d_1\ \tau = K_1^\tau \\
\Rightarrow \qquad \{\ K_1^\tau \in\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_1])\ \tau\ \} \\
S_{ffAT}\ d_1\ \tau \in\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_1])\ \tau
\end{array}
$$

Let us now define d_2 similarly:
$L_{ffAT}\ d_2$ is the smallest subset of $L_{ffAT}\ d \cap\ L_{ffAT}\ (PffAT\ [\ p_2])$ such that:

- $\varepsilon \in\ L_{ffAT}\ d_2$
- $\pi \in\ L_{ffAT}\ d_2 \wedge\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_1])\ \pi = \emptyset \wedge$
  $(m,s) \in\ S_{ffAT}\ d\ \pi \Rightarrow \pi \prec (m,s) \in\ L_{ffAT}\ d_2$
- $\pi \in\ L_{ffAT}\ d_2 \wedge\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_1])\ \pi \neq \emptyset \wedge$
  $(m,s) \in K_2^\pi \Rightarrow \pi \prec (m,s) \in\ L_{ffAT}\ d_2$

The main proof follows:

$$
\begin{array}{l}
d \in dec\ (PffAT\ [\ p_1] +_{ffAT}\ PffAT\ [\ p_2]) \\
\Rightarrow \qquad \{\tau \in\ L_{ffAT}\ d \Rightarrow \neg(\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_i])\ \tau = \emptyset) \text{ for } i \in \{1,2\}\ \} \\
\forall(\tau:\ L_{ffAT}\ d\ .
\end{array}
$$

$$
\begin{array}{ll}
 & (\mathcal{A}_{ffAT}\ (PffAT\ [\ p_1]\ )\ \tau \neq \emptyset \wedge\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_2])\ \tau = \emptyset \wedge \tau \neq \varepsilon) \vee \\
 & (\mathcal{A}_{ffAT}\ (PffAT\ [\ p_1]\ )\ \tau = \emptyset \wedge\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_2])\ \tau \neq \emptyset \wedge \tau \neq \varepsilon) \vee \\
 & (\mathcal{A}_{ffAT}\ (PffAT\ [\ p_1]\ )\ \tau \neq \emptyset \wedge\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_2])\ \tau \neq \emptyset)) \\
\Rightarrow & \quad \{\text{Three cases ii.1, ii.2, ii.3a, ii.3b below}\} \\
 & \exists (d_1, d_2\ .\ S_{ffAT}\ d\ \tau \in\ \mathcal{A}_{ffAT}\ (d_1 +_{ffAT}\ d_2)\ \tau) \\
\equiv & \quad \{\text{Definition of } dec\} \\
 & d \in dec\ (d_1 +_{ffAT}\ d_2)
\end{array}
$$

Case ii.1)

$$
\begin{array}{l}
\mathcal{A}_{ffAT}\ (PffAT\ [\ p_1]\ )\ \tau \neq \emptyset \wedge\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_2])\ \tau = \emptyset \wedge \tau \neq \varepsilon \\
\Rightarrow \\
S_{ffAT}\ d\ \tau \in\ \mathcal{A}_{ffAT}\ (d_1 +_{ffAT}\ d_2)\ \tau
\end{array}
$$

Proof

$$
\begin{array}{ll}
 & \mathcal{A}_{ffAT}\ (PffAT\ [\ p_1]\ )\ \tau \neq \emptyset \wedge\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_2])\ \tau = \emptyset \\
\Rightarrow & \quad \{\text{Take } d_1 \text{ as described above}\} \\
 & S_{ffAT}\ d_1\ \tau =\ S_{ffAT}\ d\ \tau \wedge \\
 & \mathcal{A}_{ffAT}\ (PffAT\ [\ p_1]\ )\ \tau \neq \emptyset \wedge\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_2])\ \tau = \emptyset \\
\Rightarrow & \quad \{\text{Definition of } \mathcal{A}_{ffAT} \text{ for } \mathcal{A}_{ffAT}\ (PffAT\ [\ p_2])\ \tau\} \\
 & S_{ffAT}\ d_1\ \tau =\ S_{ffAT}\ d\ \tau \wedge \tau \notin\ L_{ffAT}\ (PffAT\ [\ p_2]) \\
\Rightarrow & \quad \{\text{Take } d_2 \text{ as described above}\} \\
 & S_{ffAT}\ d_1\ \tau =\ S_{ffAT}\ d\ \tau \wedge \tau \notin\ L_{ffAT}\ d_2 \\
\Rightarrow & \quad \{\text{Definition of } \mathcal{A}_{ffAT}\} \\
 & S_{ffAT}\ d_1\ \tau =\ S_{ffAT}\ d\ \tau \wedge\ \mathcal{A}_{ffAT}\ d_2\ \tau = \emptyset \\
\Rightarrow & \quad \{\ mfs\ (e\ (\ \mathcal{A}_{ffAT}\ d_1\ \tau \cup\ \mathcal{A}_{ffAT}\ d_2\ \tau)) =\ \mathcal{A}_{ffAT}\ d_1\ \tau = \iota\ S_{ffAT}\ d_1\ \tau \\
 & \text{and definition of } \mathcal{A}_{ffAT}\} \\
 & S_{ffAT}\ d\ \tau \in\ \mathcal{A}_{ffAT}\ (d_1 +_{ffAT}\ d_2)\ \tau
\end{array}
$$

Case ii.2)

$$
\mathcal{A}_{ffAT}\ (PffAT\ [\ p_1]\ )\ \tau = \emptyset \wedge\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_2])\ \tau \neq \emptyset \wedge \tau \neq \varepsilon
$$

Proof
Similar to the proof of previous case.

Case ii.3a) $\tau = \varepsilon$

$$\mathcal{A}_{ffAT}\ (PffAT\ [\ p_1])\ \tau \neq \emptyset \wedge\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_2])\ \tau \neq \emptyset$$
$$\Rightarrow$$
$$S_{ffAT}\ d\ \varepsilon \in\ \mathcal{A}_{ffAT}\ (d_1 +_{ffAT}\ d_2)\ \varepsilon$$

Proof

$$\mathcal{A}_{ffAT}\ (PffAT\ [\ p_1])\ \tau \neq \emptyset \wedge\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_2])\ \tau \neq \emptyset$$
$\Rightarrow$ { $d \in dec\ (PffAT\ [\ p_1] +_{ffAT}\ PffAT\ [\ p_2])$ by hypothesis, definition of $\mathcal{A}_{ffAT}$ and $+_{ffAT}$ }
$$S_{ffAT}\ d\ \varepsilon \in\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_1])\ \varepsilon\ \textcircled{u}\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_2])$$
$\Rightarrow$ { Definition of $\textcircled{u}$ and choose K_1^ε and K_2^ε as shown above }
$$S_{ffAT}\ d\ \varepsilon \in mfs\ (K_1^\varepsilon \cup K_2^\varepsilon)$$
$\Rightarrow$ { Take d_i as above }
$$S_{ffAT}\ d\ \varepsilon \in mfs\ (\ S_{ffAT}\ d_1\ \varepsilon \cup\ S_{ffAT}\ d_2\ \varepsilon)$$
$\Rightarrow$ { Definition of $\mathcal{A}_{ffAT}$ }
$$S_{ffAT}\ d\ \varepsilon \in mfs\ (\ \mathcal{A}_{ffAT}\ d_1\ \varepsilon\ u\ \mathcal{A}_{ffAT}\ d_2\ \varepsilon)$$
$\Rightarrow$ { Definition of $+_{ffAT}$ and $\textcircled{u}$ }
$$S_{ffAT}\ d\ \varepsilon \in\ \mathcal{A}_{ffAT}\ (d_1 +_{ffAT}\ d_2)\ \varepsilon$$

Case ii.3b) $\tau \neq \varepsilon$

$$\mathcal{A}_{ffAT}\ (PffAT\ [\ p_1])\ \tau \neq \emptyset \wedge\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_2])\ \tau \neq \emptyset$$
$$\Rightarrow$$
$$S_{ffAT}\ d\ \tau \in\ \mathcal{A}_{ffAT}\ (d_1 +_{ffAT}\ d_2)\ \tau$$

Proof

$$\mathcal{A}_{ffAT}\ (PffAT\ [\ p_1])\ \tau \neq \emptyset \wedge\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_2])\ \tau \neq \emptyset$$
$\Rightarrow$ { $d \in dec\ (PffAT\ [\ p_1] +_{ffAT}\ PffAT\ [\ p_2])$ by hypothesis, definition of $\mathcal{A}_{ffAT}$ and $+_{ffAT}$ }
$$S_{ffAT}\ d\ \tau \in\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_1] +_{ffAT}\ PffAT\ [\ p_2])\ \tau$$
$\Rightarrow$ { Definition of $+_{ffAT}$ }
$$S_{ffAT}\ d\ \tau \in mfs\ (c\ (\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_1])\ \tau \cup\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_2])\ \tau))$$
$\Rightarrow$ { Lemma 5.2.4 there exist $K_j^\tau \in\ \mathcal{A}_{ffAT}\ (PffAT\ [\ p_j])$ }

$$
\begin{array}{ll}
 & S_{ffAT}\ d\ \tau \in mfs\ (c\ (\{K_1^\tau, K_2^\tau\})) \\
\Rightarrow & \quad \{\text{Take } d_j \text{ as defined above}\} \\
 & S_{ffAT}\ d\ \tau \in mfs\ (c\ (\{\ S_{ffAT}\ d_1\ \tau,\ S_{ffAT}\ d_2\ \tau\})) \\
\Rightarrow & \quad \{\ d_j \text{ are in } DFT\ \} \\
 & S_{ffAT}\ d\ \tau \in mfs\ (c\ (\ \mathcal{A}_{ffAT}\ d_1\tau \cup\ \mathcal{A}_{ffAT}\ d_2\ \tau)) \\
\Rightarrow & \quad \{\text{Definition of } \mathcal{A}_{ffAT}\} \\
 & S_{ffAT}\ d\ \tau \in\ \mathcal{A}_{ffAT}\ (d_1 +_{ffAT}\ d_2)\ \tau
\end{array}
$$

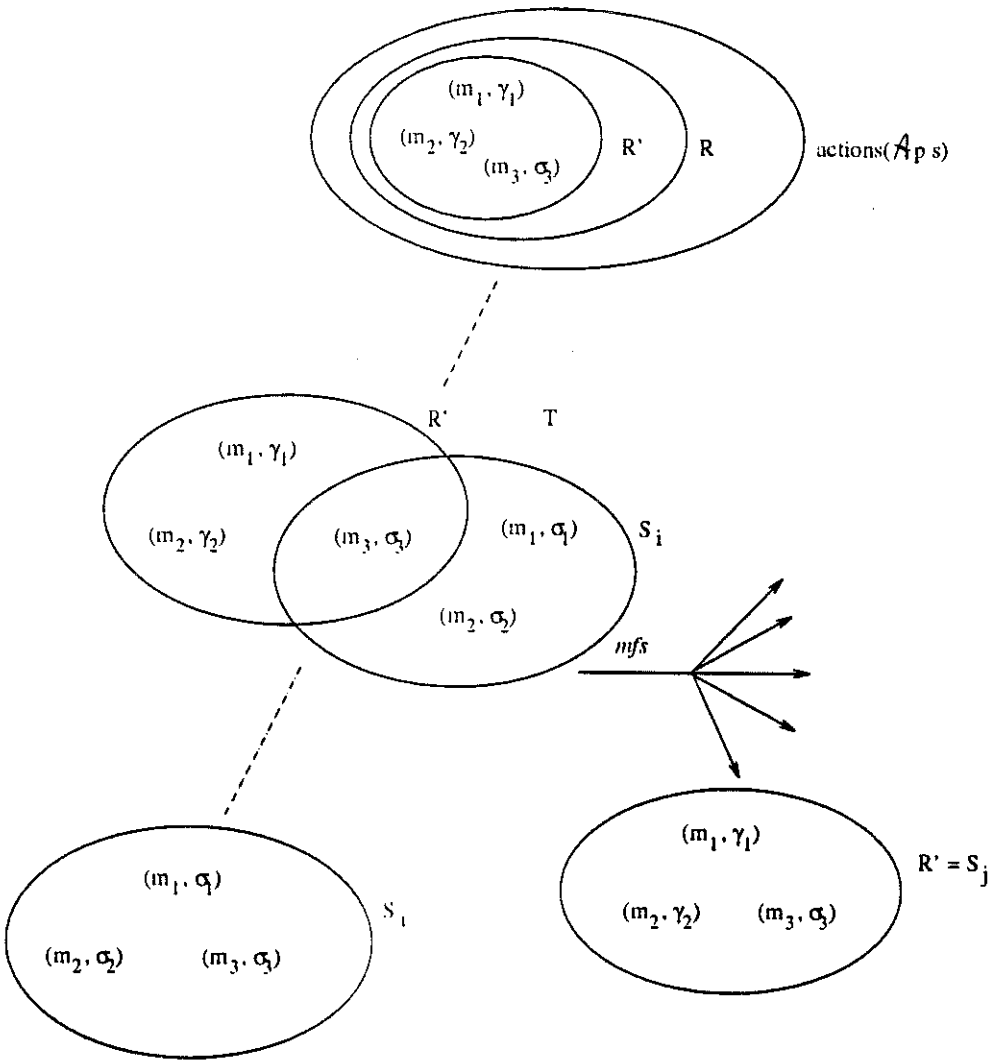

Figure 5.17: Illustration to proof of Lemma 5.3.11

Bibliography

[Abr84] S. Abramsky. Reasoning about concurrent systems. In P. Jones Chambers, Duce, editor, *Distributed Computing.* Academic Press, 1984.

[Abr89] S. Abramsky. A generalized kahn principle for abstract asynchronous networks. In *Mathematical Foundations of Programming Language Semantics*, 1989.

[Bou92a] R. Boute. A declarative formalism supporting hardware/software co-design. In *IFIP International Workshop on Hardware / Software Co-design*, 1992.

[Bou92b] R. Boute. Foundamentals of hardware description languages and declarative languages. In *Program Design Calculi - International Summer School* [NAT92].

[Bou93] R. Boute. Funmath illustrated: A declarative formalism and application examples. Technical Report Declarative Systems Series n.1, Univerity of Nijmegen, jul 1993.

[Bro90] M. Broy. Functional specification of time sensitive communicating systems. In J. W. de Bakker, W. P. de Roever, and Grzegorz Rozenberg, editors, *Stepwise Refinement of Distributed Systems — Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[Bro92a] M. Broy. Compositional refinement of interactive systems. In *Program Design Calculi - International Summer School* [NAT92].

[Bro92b] M. Broy. (inter-)action refinement: The easy way. In *Program Design Calculi - International Summer School* [NAT92].

[BW88] R. Bird and Ph. Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.

[DC89] E.W. Dijkstra and Scholten C.S. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1989.

[DN87] R. De Nicola. Extensional equivalences for transition systems. *Acta Informatica*, (24):211–237, 1987.

[DS89] P. Dybjer and H.P Sander. A functional programming approach to the specification and verifivation of concurrent systems. *Formal Aspects of Computing*, 1:303–319, 1989.

[Dyb86] P. Dybjer. Program verification in a logic theory of construction. In *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 334–349. Springer-Verlag, 1986.

[Hen88] M. Hennessy. *Algebraic Theory of Processes*. The MIT Press, 1988.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prantice Hall, 1985.

[Jon87] B. Jonsson. A fully abstract trace model for dataflow networks. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE — Parallel Architectures and Languages Europe*, volume 259 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[Kah92] G. Kahn. The semantics of a simple language for parallel programming. In R. Aiken, editor, *Proceedings of the 1974 IFIP Congress*. IFIP, Elsevier Science Publishers B.V., 1992.

[Mil89] R. Milner. *Communication and Concurrency*. Series in Computer Science. Prantice Hall, 1989.

[Mis90] J. Misra. Equational reasoning about nondeterministic processes. *Formal Aspects of Computing*, 2:167–195, 1990.

[MNV73] Z. Manna, S. Ness and J. Vuillemin. Inductive methods for proving properties of programs. *Communications of the ACM*, 16(8), 1973.

[MR92] M. Massink and L. Rooijakkers. Equational semantics for basic LOTOS and an example of its use in a transformational proof style. In P. Dewilde and Vandewalle J., editors, *Computer Systems and Software Engineering - the 6th Annual European Computer Conference*. IEEE. IEEE Computer Society Press, 1992.

[NAT92] Program design calculi - international summer school, 1992.

[Ong93] C.H.L. Ong. Non-determinism in a functional setting. In *IEEE Symposium on Logic in Computer Science*. IEEE, Computer Society Press, 1993.

[Plo76] J. D. Plotkin. A powerdomain contruction. *SIAM Journal of Computing*, 5(3):452–487, 1976.

[Rei85] W. Reisig. *Petri-Nets - an introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[San92] H. P. Sander. *A Logic of Functional Programs with an Application to Concurrency*. PhD thesis, Univ. of Göteborg and Chalmers Univ. of Technology, 1992.

[vG90] R.J. van Glabbeek. *Comparative Concurrency Semantics and Refinement of Actions*. PhD thesis, University of Amsterdam, 1990.

[vT94] H. A. van Thienen. *It's About Time - Using Funmath for the Specification and Analysis of Discrete Dynamic Systems*. PhD thesis, Univerity of Nijmegen, 1994.