NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

ISBN 0-315-56394-X

Trace Analysis of LOTOS Behaviours

By

M. Souheil Gallouzi

THESIS SUBMITTED
TO THE SCHOOL OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE IN COMPUTER SCIENCE
UNDER THE AUSPICES OF THE
OTTAWA-CARLETON INSTITUTE FOR COMPUTER SCIENCE

at the
UNIVERSITY OF OTTAWA
August 1989
Revised and Accepted October 89

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Thesis Committee

Supervisor:

Prof. Luigi Logrippo, University of Ottawa.

Examiners:

Prof. Scott A. Smolka, State University of New York at Stonybrook.

Prof. Philip J. Scott, University of Ottawa.

Abstract

ISO has developed LOTOS for the formal description of standardized data communication protocols and services. The component of LOTOS that deals with the description of process behaviours and interactions (the so-called "control part" of LOTOS or "pure LOTOS") is based on a clever mixture of Milner's CCS and Hoare's CSP. Most of the theoretical framework of this component is based on Milner's work, especially nondeterminism which is modelled by internal actions as in CCS rather than using special operators as in CSP. The dynamic semantics of this component is expressed in operational terms by inference rules as in CCS, and therefore, it has been possible to prove a rich set of algebraic laws, similar to those of CCS, for all LOTOS process constructors. In this thesis, we attempt a characterization of the control part of LOTOS by using concepts borrowed from CSP, i.e. traces, refusals, and failures.

We first provide a characterization of LOTOS processes by using *traces*. To this end, we adapt the existent trace theory and define new composition functions on traces. The result is a trace semantics presented in denotational style, the trace sets of compound processes being built up from the trace sets of their components. This characterization is found to be suitable for reasoning about communication sequences, but unable to cope properly with nondeterminism.

Following the lead of TCSP we adapt the *failures model* to LOTOS. This is a direct extension of the trace model, obtained by associating to every process a so-called *refusal set*. The result is an alternative semantics for LOTOS restricted to strongly convergent processes. This semantics is basically denotational but having strong connections with operational behaviours.

We also propose a Hoare-style proof system for LOTOS defined in terms of *proof rules* based on the structure of processes. We use Hoare's satisfaction relation to define these rules. This system is believed to be adequate to allow proofs of correctness of compound processes to be constructed from proofs of correctness of its parts.

Acknowledgments

I would like to express my foremost thanks to my supervisor, Prof. Luigi Logrippo, for his valuable time, patience and advice throughout my graduate studies. His accuracy in reviewing the drafts of the thesis have greatly improved the contents and its presentation. The many discussions we have had since I started research work towards this thesis have been of great influence. I am also much indebted to Abdellatif Obaid for initiating the idea of using traces to characterize LOTOS behaviours and design a logical system to prove their properties. We have had many inspiring and fruitful discussions together. Also, his useful comments on early versions of some parts of the thesis were very much appreciated. I would like to thank Dr. Jan de Meer for reading and commenting on an early version of Chapter 2. I would also like to express my gratitude to Prof. Scott Smolka for his careful reading and suggestions which improved the presentation of the final version of the thesis.

The members of the LOTOS Group and Protocol Research Group at the University of Ottawa are thanked for the pleasant atmosphere within the two groups. The help and technical support of Jacques Sincennes is greatly acknowledged.

I would like to express my gratitude to the Tunisian government for its financial support under the scholarship program sponsored by CIDA.

I want to mention specifically the crucial support provided by my wife Leila, with her patience, understanding, cheerfulness, and much more. The words are not sufficient to express my gratitude to Leila. Finally, my deepest thanks go to my parents who have given me an optimistic outlook, a taste for quality, and the freedom to manage my own life. They have always supported

and encouraged whatever I wanted to do.

Contents

Chapter 1

Introduction

1.1 Background

Robin Milner was the first person to use an algebraic methodology to attempt a formalization of the theory of communicating systems. His work led to CCS [Mil80], which consists of an algebraic language of processes, a set of behavioural equivalence relations, and a calculus that supports reasoning about processes based on a set of equations. This established the link, for the first time, between an equational theory and a behaviour equivalence, namely *observational equivalence*, based on an operational semantics. A more elegant version of this equivalence, called *bisimulation equivalence*, is defined in [Par81] and its application is elaborated in [Mil83]. An important construct that served to model nondeterminism in CCS is the internal action, denoted by τ. This action is hidden from its environment, and whenever a process is willing to engage in it, it is permitted to do so. This construct played an

important role in achieving the maximum expressive power of CCS with as few distinct primitive operators as possible.

Another school of thought is centered around the language CSP, which since its appearance [Hoa78] has played the role of a test language for many proposed theories of concurrency, by providing very interesting control structures. Following the algebraic approach advocated by Milner, the language changed radically [BHR84, Hoa85]. Binary communication, as in CCS, is replaced by multiway synchronization, and communication with values is replaced by uninterpreted actions. The new version of CSP, also referred to as TCSP (Theoretical CSP), consists of a set of combinators, together with recursive definition of processes. Its semantics is presented in terms of a denotational model called the *failure model* [BHR84, BR85], which, in turn, is used to justify equations over the set of combinators. Each possible failure of a process represents a finite piece of behaviour in which the process has engaged in a sequence of visible actions up to some moment, called a *trace*, and has since then refused to participate in some set of actions, called *refusal*. Therefore, instead of using τ to represent nondeterministic behaviours, nondeterminism manifests itself after a process has engaged into a trace and refuses some sets of actions. These refusals represent the possible consequences, for the next step, of the various nondeterministic decisions available to the process. A notion of *failure equivalence* is introduced, which identifies processes with the same failure set as denotation.

Comparative studies of both models, CCS and CSP, can be found in [Bro83, vG86]. In particular, the work of [Bro83] makes explicit the differences between the two systems, and shows that the semantic models underlying CCS and CSP are comparable.

During the period of this remarkable growth in the general understanding of the algebraic approach to the theory of communicating systems, the theory and practice of specification languages for data communication protocols and services (often called Formal Description Techniques or FDTs) has also been the object of much interest. The International Organization for Standardization (ISO), has been developing over the years a family of standardized data communications protocols, called OSI (Open Systems Interconnection). Therefore, it was necessary to provide OSI with an appropriate FDT, a precisely defined language for the formal description of the OSI architecture. A determining direction was taken by a decision to use CCS as the semantical basis for the control component (the component that deals with the description of process behaviours and interactions) of such an FDT. Since then this FDT has been referred to as LOTOS, the Language of Temporal Ordering Specifications. CSP has also influenced the design of LOTOS, especially in the construction of the parallel composition operator. By separating hiding from parallel composition, multiple processes can engage in a single action. This has led to the introduction of the so-called constraint-oriented specification style [VSvS88, Tur88].

The data type component of LOTOS is based on the algebraic specification language ACT ONE [EM85]. In this thesis, we will limit ourselves to describing the semantics of the control component, the so-called "pure LOTOS". Complete tutorials on LOTOS can be found in [Tur87, BB87, ISO88] and much additional information is given in [vEVD89].

1.2 LOTOS behaviours

LOTOS specifications describe distributed, concurrent systems via a hierarchy of process definitions. The following shows a typical structure of a process definition

> **process** "*PROCESS NAME*" [*"sequence of action names: gates"*] :=
> ⟨*behaviour expression*⟩
> **endproc**

As a convention we use boldface for reserved LOTOS keywords, italics for actions, and (italics) upper case names for processes. The reader should note that many syntactic details of LOTOS are omitted in the notation of this thesis to simplify our examples as much as possible.

A process is capable of performing internal, unobservable actions, and to interact with other processes, which form its environment, via its gates (observable actions). The main component of a process definition is the *behaviour expression* which is built from the constants and operators of LOTOS. A behaviour expression may include instantiations of other already defined processes. The most elementary components of a behaviour expression are the actions. They are indivisible and not subject to further investigations.

The nullary operator of LOTOS is denoted by **stop**, which is used to represent inaction or deadlock. Successful termination of a process is represented by **exit**, which offers a special symbol δ and then behaves like **stop**. The actions in a behaviour expression are action offers, and the actual actions which happen may be influenced by the environment of the expression. For example,

(*pass*; exit) [] (*fail*; stop)

will result in *pass* or *fail* depending on the initial action offered by the environment. The *choice operator* "[]"is used when alternative behaviours are allowed, and the *action prefix operator* ";" prefixes a behaviour expression with an action. For instance, if *pass* occurs, the process will terminate successfully, otherwise it may fail (*fail*) and die. However, a (good) student who is writing an exam does not really have the choice whether to pass or fail the exam; an unexpected event may lead him to a failure, in which case we write

(*pass*; exit) [] (i; *fail*; stop)

where i represents an *internal action*, and in this case it is not guaranteed that the environment can influence the occurrence of one the alternatives. The choice is made nondeterministically. The first alternative in the choice expression is the "normal" result expected by a good student and the second alternative is the devil's one. Now consider the following behaviour

(i; *pass*; exit) [] (i; *fail*; stop)

where the choice expression offers the same uncertainty. This means that no matter how hard a student works, passing or failing the exam is a matter of chance.

A woman who decides to get pregnant cannot choose between a baby girl or a baby boy. The choice is completely nondeterministic. This can be described in LOTOS as follows

(*pregnant*; *boy*; exit) [] (*pregnant*; *girl*; exit)

where identical actions are offered as alternatives, and in this case the environment cannot influence the consequence of the chosen alternative. Note that the expression

pregnant; (*boy*; exit [] *girl*; exit)

is deterministic; the pregnant women can decide on whether to have a boy or a girl. However, this is not realistic, and therefore, the two expressions above should not be regarded as equivalent by any means.

Another way of describing the choice between alternative behaviours is expressed by the *disabling operator* "[>", which allows the second named expression to interrupt the first named expression, and if this happens the future behaviour is that of the second expression only. If the first expression terminates successfully, it can no longer be interrupted. As an example consider our life cycle

(*birth*; *puberty*; *marriage*; stop) [> (*death*; exit)

However this allows *death* before *birth*, which can be argued to be nonsense. We can "correct" this using the *parallel composition operator* "|[...]|"

(*birth*; *puberty*; *marriage*; stop) [> (*death*; exit)
|[*birth*, *death*]|
(*birth*; *death*; exit)

This expression will synchronize on the birth and death events, since they are explicitly specified in the list of gates of the parallel operator, and will not allow *death* before *birth*. The second part of the expression can be viewed as a constraint on the action sequences of the first one. Successful termination of this process depends on the successful termination of both behaviour expressions; the special "exit" action δ is always synchronized. With an explicit empty list in "|[...]|", the parallel composition amounts to interleaving "|||", which allows behaviours to unfold completely independently in parallel, except on δ. If the list contains all gates, the behaviour expressions are obliged to synchronize on all observable actions (including δ), which is represented by the "||" operator.

Two behaviour expressions can be combined in sequence using the *enabling operator* "$\gg$". For example, the behaviour of many students may (partially) be described as follows

(*study*; *graduate*; exit) $\gg$ (*work*; *pay_tax*; stop)

Once the first expression exits, control is passed to the second expression and the occurrence of δ in this case is seen as an internal action. If the first behaviour expression does not terminate successfully, the second behaviour expression cannot be enabled.

The *hiding operator* in LOTOS transforms observable actions into i-actions, and so they become unobservable. This bears a similarity with the concealment operator of CSP, except that it introduces internal actions implicitly into a specification. CCS did not need a separate operator for hiding, since abstraction and parallel composition are integrated.

Process instantiation in LOTOS refers to an already defined process, where the associated list of gates can be used to rename the corresponding actions of the behaviour expression defining that process, without modifying its structure. Recursion is achieved by making a process refer to itself.

In [ISO88, BB87] LOTOS is formally defined by means of an operational semantics, which permits the derivation of the actions that a behaviour expression may perform from the structure of the expression. This is done by applying a labelled transition relation of the form $B \xrightarrow{x} B'$, which means that the behaviour expression B can engage in the action x and then behave like B' (another behaviour expression). This semantics consists of a set of axioms and inference rules that can be applied to a behaviour expression to build its *transition tree*, also called *action tree* or *synchronization tree*, where arcs are labelled by actions and nodes by behaviour expressions. Note that in this

model stop does not have any axiom or inference rules associated with it. We give now the axioms and inference rules for the "pure LOTOS" subset of LOTOS that is considered in this thesis. We use the following conventions

B_i $(i \geq 1)$	stands for a behaviour expression
a	stands for an observable action other than δ
a'	stands for any observable action (including δ)
x	stands for any action other than δ
x'	stands for an arbitrary action (including δ)
S	stands for a sequence of observable actions not containing δ
$\xrightarrow{i}$	stands for a transition involving the i-action

- Successful termination

  $$\textbf{exit} \xrightarrow{\delta} \textbf{stop}$$

- Action prefix

  $$x;\ B \xrightarrow{x} B$$

- Hiding

  $$B \xrightarrow{x'} B' \wedge x' \notin \{S\} \Rightarrow \textbf{hide}\ S\ \textbf{in}\ B \xrightarrow{x'} \textbf{hide}\ S\ \textbf{in}\ B'$$
  $$B \xrightarrow{a} B' \wedge a \in \{S\} \Rightarrow \textbf{hide}\ S\ \textbf{in}\ B \xrightarrow{i} \textbf{hide}\ S\ \textbf{in}\ B'$$

- Choice

  $$B_1 \xrightarrow{x'} B_1' \Rightarrow B_1 [] B_2 \xrightarrow{x'} B_1'$$
  $$B_2 \xrightarrow{x'} B_2' \Rightarrow B_1 [] B_2 \xrightarrow{x'} B_2'$$

- Parallel composition

  $$B_1 \xrightarrow{x} B_1' \wedge x \notin \{S\} \Rightarrow B_1 |[S]| B_2 \xrightarrow{x} B_1' |[S]| B_2$$
  $$B_2 \xrightarrow{x} B_2' \wedge x \notin \{S\} \Rightarrow B_1 |[S]| B_2 \xrightarrow{x} B_1 |[S]| B_2'$$
  $$B_1 \xrightarrow{a'} B_1' \wedge B_2 \xrightarrow{a'} B_2' \wedge a' \in \{S\} \cup \{\delta\}$$
  $$\Rightarrow B_1 |[S]| B_2 \xrightarrow{a'} B_1' |[S]| B_2'$$

- Enabling

$$B_1 \xrightarrow{x} B_1' \Rightarrow B_1 \gg B_2 \xrightarrow{x} B_1' \gg B_2$$
$$B_1 \xrightarrow{\delta} B_1' \Rightarrow B_1 \gg B_2 \xrightarrow{i} B_2$$

- Disabling

$$B_1 \xrightarrow{x} B_1' \Rightarrow B_1 [> B_2 \xrightarrow{x} B_1' [> B_2$$
$$B_1 \xrightarrow{\delta} B_1' \Rightarrow B_1 [> B_2 \xrightarrow{\delta} B_1'$$
$$B_2 \xrightarrow{x'} B_2' \Rightarrow B_1 [> B_2 \xrightarrow{x'} B_2'$$

The formal semantics of process instantiation will be described later. It is defined in terms of a renaming operation on processes.

1.3 Verification models of LOTOS

The operators of LOTOS were chosen in such a way that it has been possible to prove about them a rich set of algebraic laws, similar to those of CCS. Therefore, most proof methods associated with LOTOS are based on the concept of bisimulation equivalence. The best developed model (I know of) is LOTCAL [Bri88b], which identifies a small set of operators that suffices for the formal interpretation of LOTOS. It is closely related to the work of Milner. Other attempts to define alternative semantic models for LOTOS are reported in [vEVD89].

Another related work that is worth mentioning here is due to Brinksma et al. [BSS87, Bri88a]. This involved the extension of the theory of testing and failure equivalences for CCS and CSP to LOTOS, which led to the introduction of new notions such as the extension and conformance relations,

and the concept of canonical tester.

A number of tools were developed for the practical verification of LOTOS specifications. In particular, at least two interpreters are in existence today, [LOBF88, vEVD89] that allow LOTOS users to exercise a specification of a complex system at the design stage. This means that design errors can be revealed in an early stage of the software development cycle.

1.4 Motivation and organization

With our work we intend to introduce an alternative framework for the study of LOTOS specifications. Our approach is mainly denotational but having a strong connection with operational behaviours [Old86]. Such approach has been applied successfully to many nontrivial languages. The most relevant example to our work is that of CSP and TCSP. Recall that the motivation for seeking such models is to provide a useful mathematical framework for the analysis of programs, and for developing logical systems to prove their properties.

We first attempt to characterize LOTOS behaviours by means of their traces, and use this as a basis to develop a logical system to prove their "partial correctness", an idea inspired by A. Obaid in [Oba86]. This led us to study the trace theory of CSP and to extend it to LOTOS. The introduction of refusals strengthened significantly the model, and it was possible to give each LOTOS constructor a denotation based on its failure set.

In most cases, we only give an informal justification of the correspondence between our denotation of processes and their interpretations by means of the operational semantics. This is likely to be satisfactory, since such formal proofs turn out, in many cases, to be lengthy and to bear many similarities. Unfortunately, including proofs of even only the most important of our propositions would have led to an unbearably long (and not necessarily more enlightening) document.

Our work is closely related to CSP, and favours a proof theory based on failure equivalences. It is meant to be complementary with respect to the existing techniques, by offering an alternative theory for the validation of LOTOS specifications. A question that presents itself here is, should failure equivalence play an important role in LOTOS proof theory? Indeed, it has many nice properties, a rich theory, and an attractive methodology for verifying correctness of nontrivial protocols. If this is not convincing, bear with us and the remainder of the thesis may "change your mind".

Chapter 2 presents a trace theory that allows the expression of relations between compound processes as equations between the traces of their components. In Chapter 3 we apply this theory to characterize processes by means of their trace sets, and derive a Hoare-style proof system to prove the correctness of their trace specifications. Chapter 4 presents an extension of this trace model to cope with nondeterministic behaviours. This leads us to a revised version of trace specifications and the associated proof system, and to a failure semantics for LOTOS restricted to strongly convergent processes. We conclude the thesis in Chapter 5 with some remarks and suggestions for future work. In the Appendix we list most of the abbreviations, symbols, and notations used in this thesis.

Chapter 2

Theory of Traces

2.1 Introduction

A *trace* is a finite-length sequence of *symbols*. Symbols are denoted by identifiers and may represent atomic statements of a (concurrent) system, often referred to as actions or events. Each trace may then be interpreted as a sequence of (atomic) interactions that may take place between a process and its environment. A trace will be denoted as follows

- $\langle\rangle$ The empty trace containing no symbols.
- $\langle a \rangle$ A trace containing only the symbol a.
- $\langle a, b \rangle$ A trace containing two symbols, a followed by b.
- $\langle a \cdot t \rangle$ A trace containing the symbol a followed by the trace t. In other words, a and t are the *head* and *tail* of the trace, respectively.

This notation suggests that two traces are said to be *equal* if they are both empty or their heads and tails are equal.

Every trace is associated with some *alphabet*; a finite set of symbols. We interpret each symbol from the trace's alphabet as a possible interaction between a process and its environment. For each alphabet A, A^* denotes the set of all traces, including the empty trace $\langle\rangle$, which are formed from symbols in A. We will need to handle a special symbol denoted by δ, which is used in LOTOS to indicate the successful termination of a process which engages in it, via the exit process. Therefore this symbol can appear only at the end of a trace. We will thus let A_δ be the alphabet $A \cup \{\delta\}$ and A_δ^* be the set of all traces of symbols of A_δ, where δ may only occur at the end of a trace. Such traces are said to be *well-formed*. Note that the (well-applied) composition of well-formed traces using the functions and operations introduced in this chapter will always yield (a set of) well-formed traces.

In the remainder of this chapter we define the most important operators and composition functions on traces, and state their chief properties. This part has borrowed many ideas from [Hoa85, vdS85]. We shall omit the details of some of the proofs of the properties stated, when appropriate, and ignore the obvious ones. We will also use the following conventions

$a, b, c, d, \ldots$	stand for symbols
s, t, u, v, w	stand for traces
T, U, V	stand for sets of traces
A, B	stand for alphabets

2.2 Projection

The *projection* of a trace t on an alphabet A, denoted by $t\lfloor A$, can be obtained from t by omitting all symbols outside A. For example

$\langle a, b, c, d, a\rangle\lfloor\{a, b\} = \langle a, b, a\rangle$

This operator is defined as follows

1) $\langle\rangle\lfloor A = \langle\rangle$
2) $\langle a \cdot t\rangle\lfloor A = t\lfloor A$ if $a \notin A$
3) $\langle a \cdot t\rangle\lfloor A = \langle a \cdot t\lfloor A\rangle$ if $a \in A$

The following properties are consequences of this definition

P 2.1 $t\lfloor\{\} = \langle\rangle$

P 2.2 $t\lfloor A\lfloor B = t\lfloor(A \cap B) = t\lfloor B\lfloor A$

P 2.3 $t \in A^* \equiv t = t\lfloor A$

Next we extend the definition of projection to operate on sets of traces as well. It is defined by

$T\lfloor A = \{t\lfloor A | t \in T\}$

Projection of sets of traces is monotonic w.r.t. inclusion ordering ($\subseteq$) over sets

P 2.4 $U \subseteq V \Rightarrow U\lfloor A \subseteq V\lfloor A$

2.3 Length

The *length* of a trace t is denoted $|t|$. For example

$$|\langle a, b, c\rangle| = 3$$
$$|\langle\rangle| = 0$$

The number of occurrences of symbols from A in a trace t can be counted by $|t\lfloor A|$. Therefore, we define

$$t \downarrow A = |t\lfloor A|$$

If A contains a single symbol a, the number of its occurrences in the trace t is denoted $t \downarrow a$ (instead of $t \downarrow \{a\}$). The following property is an immediate consequence of the above definition and P 2.2

P 2.5 $(t\lfloor A) \downarrow B = t \downarrow (A \cap B) = (t\lfloor B) \downarrow A$

We also have

P 2.6 $t \downarrow (A \cup B) = t \downarrow A + t \downarrow B - t \downarrow (A \cap B)$

2.4 Concatenation

The *concatenation* operator, denoted by "$\frown$", constructs a trace by putting two traces, u and v, together in this order. For example

$$\langle a, b, c\rangle \frown \langle d, c\rangle = \langle a, b, c, d, c\rangle$$
$$\langle a, b\rangle \frown \langle\rangle = \langle a, b\rangle$$

It is defined as follows

1) $\langle\rangle \frown v = v \frown \langle\rangle = v$

2) $\langle a \cdot u\rangle \frown v = \langle a \cdot u \frown v\rangle$

Concatenation should always yield well-formed traces, and so it is undefined for traces associated with alphabets containing the special symbol δ in its first argument. Clearly it is associative (that is $(u \frown v) \frown t = u \frown (v \frown t)$) but not commutative ($u \frown v \neq v \frown u$), and has $\langle\rangle$ as its unit. It also satisfies the following properties

P 2.7 $(u \frown v)\lfloor A = (u\lfloor A) \frown (v\lfloor A)$

P 2.8 $|u \frown v| = |u| + |v|$

P 2.9 $(u \frown v) \downarrow A = (u \downarrow A) + (v \downarrow A)$

P 2.10 $(u \frown v) \in A_\delta^* \equiv u \in A^* \wedge v \in A_\delta^*$

We can now define t^n, the n^{th} trace power of t where n is a natural number, as n copies of t concatenated to each other. This can formally be defined as

3) $t^0 = \langle\rangle$

4) $t^{n+1} = t \frown t^n = t^n \frown t$

The following are obvious consequences of this definition and the above properties

P 2.11 $|t^n| = n \times |t|$

P 2.12 $t^n \downarrow A = n \times (t \downarrow A)$

We shall write t^* and t^+ to denote n or more copies of t concatenated to each other, where n is equal to 0 and 1 respectively.

2.5 Prefix

We write $u \preceq v$, to denote that u is an initial segment of v, often called *prefix*. This means that

$$|u| \leq |v|$$

and the two traces are identical in their first $|u|$ symbols. For example

$$\langle a, b\rangle \preceq \langle a, b, a, c\rangle$$

We call u a prefix of t if and only if

$$\exists v.t = u \frown v$$

If $v \neq \langle\rangle$ we can write $u \prec t$. The following properties are obvious consequences of this definition

P 2.13 $\langle\rangle \preceq t$ *(least element)*

P 2.14 $t \preceq t$ *(reflexive)*

P 2.15 $u \preceq v \wedge v \preceq u \Rightarrow u = v$ *(antisymmetric)*

P 2.16 $u \preceq v \wedge v \preceq t \Rightarrow u \preceq t$ *(transitive)*

Therefore $\preceq$ is a partial ordering relation. It follows that a function f that maps traces to traces is said to be *monotonic* if it preserves the ordering $\preceq$ ($f(u) \preceq f(v)$ whenever $u \preceq v$). For example projection is monotonic

P 2.17 $u \preceq v \Rightarrow (u\lfloor A) \preceq (v\lfloor A)$

Concatenation is monotonic in its second argument, keeping the first argument constant

P 2.18 $u \preceq v \Rightarrow (t \frown u) \preceq (t \frown v)$

The set of traces that contains all the prefixes of a trace t is called *prefix closure* of t, and is denoted by $pref(t)$. We have

$$pref(t) = \{u | u \preceq t\}$$

P 2.19 $u \preceq v \equiv pref(u) \subseteq pref(v)$

The prefix closure of a set of traces T is defined as

$$pref(T) = \bigcup_{t \in T} pref(t)$$

P 2.20 $T \subseteq pref(T)$

P 2.21 $\{\langle\rangle\} \subseteq pref(T)$

A set T is called *prefix-closed* if $T = pref(T)$. The $pref$ of sets of traces is monotonic for $\subseteq$-ordering, in that

P 2.22 $U \subseteq V \equiv pref(U) \subseteq pref(V)$

The last two properties describe the distribution of projection through prefix closures

P 2.23 $pref(T)\lfloor A = pref(T\lfloor A)$

P 2.24 *The projection of a prefix-closed set on any alphabet is prefix-closed.*

2.6 Containment

The *contiguous containment* operator, denoted by "*in*", is used to express the fact that a trace u is a contiguous subsequence of a trace v (not necessarily initial). We write u *in* v if and only if

$$\exists s, t. v = s \frown u \frown t$$

Note that

P 2.25 $u \preceq v \Rightarrow u \; in \; v$

If u is a subsequence of v, not necessarily a contiguous one, we write $u \sqsubseteq v$; this is defined as follows

$$u \sqsubseteq v \equiv (\exists u_1, \ldots, u_n, v_1, \ldots, v_{n+1} \cdot v = v_1 \frown u_1 \frown v_2 \frown \ldots \frown u_n \frown v_{n+1} \wedge u = u_1 \frown \ldots \frown u_n)$$

where $n \geq 1$. For example

$$\langle a, d \rangle \sqsubseteq \langle b, a, c, d, a \rangle$$

Clearly both relations are also partial orderings, and their least element is $\langle\rangle$. Moreover, projection is monotonic for these orderings, since it distributes through concatenation; they both satisfy property P 2.17. Also concatenation is monotonic in all its arguments for the $\sqsubseteq$-ordering

P 2.26 $u \sqsubseteq v \Rightarrow (t \frown u) \sqsubseteq (t \frown v)$
$u \sqsubseteq v \Rightarrow (u \frown t) \sqsubseteq (v \frown t)$

The $\sqsubseteq$-ordering is a generalization of the *in*-ordering, therefore

P 2.27 $u \; in \; v \Rightarrow u \sqsubseteq v$

The effect of projection can now be described

P 2.28 $t\lfloor A \sqsubseteq t$

2.7 Sequential composition

If the successful termination symbol δ does not occur at the end of a trace u (in which case it does not occur in u at all), the *sequential composition* of u and v, denoted $u \gg v$, is u. If δ does occur at the end of u, it is removed and the result is concatenated to v. We can give a formal definition of sequential composition on traces by means of the following rules

1) $u \gg v = u$ if $\neg(\langle\delta\rangle \; in \; u)$
2) $(u \frown \langle\delta\rangle) \gg v = u \frown v$

For example

$\langle a, b, c\rangle \gg \langle a\rangle = \langle a, b, c\rangle$
$\langle a, b, a, \delta\rangle \gg \langle b, d\rangle = \langle a, b, a, b, d\rangle$

This composition operator is monotonic in all its arguments

P 2.29 $u \preceq v \Rightarrow ((t \gg u) \preceq (t \gg v))$
$u \preceq v \Rightarrow ((u \gg t) \preceq (v \gg t))$

Proof. The proof of the first part is trivial. The second part follows from the fact that, if $u = (s \frown \langle\delta\rangle)$ then $u \gg t = s \frown t$ (by 2)), and $v \gg t = (s \frown t)$ since δ can only appear at the end of a trace and u is an initial subsequence of v containing δ; that is, $u = v$. On the other hand if $\neg(\langle\delta\rangle \; in \; u)$ then $u \gg t = u$ (by 1)), and $v \gg t$ is either v, or $(u \frown s \frown t)$ if $v = u \frown s \frown \langle\delta\rangle$. □

This property is also satisfied by the $\sqsubseteq$-ordering. Moreover, $\gg$ is clearly associative, strict (maps the empty trace to the empty trace) in its first argument, and has $\langle\delta\rangle$ as its unit.

P 2.30 $t \gg (u \gg v) = (t \gg u) \gg v$

P 2.31 $\langle\rangle \gg t = \langle\rangle$

P 2.32 $\langle\delta\rangle \gg t = t \gg \langle\delta\rangle = t$

2.8 Renaming

Let F be a function mapping symbols in an alphabet A to symbols in an alphabet B such that

$F(a) = \delta \equiv a = \delta$

We define the postfixed *renaming* operation over traces, as follows

1) $\langle\rangle[F] = \langle\rangle$

2) $\langle a \cdot t\rangle[F] = \langle F(a) \cdot t[F]\rangle$

Thus if $t \in A_\delta^*$ then $t[F] \in B_\delta^*$. For example

if $F : \{a, b, c, d\} \longrightarrow \{a, c, e, f\}$ given by

$F(a) = a$, $F(b) = c$, $F(c) = e$, $F(d) = f$

is a renaming, we then have $\langle a, b, c, d\rangle[F] = \langle a, c, e, f\rangle$

We shall use convenient abbreviations in writing renamings explicitly. Thus

$b_1/a_1, \ldots, b_n/a_n$

(where $a_1, \ldots, a_n$ are distinct symbols) stands for the renaming

$F : \{a_1, \ldots, a_n\} \longrightarrow \{b_1, \ldots, b_n\}$

given by

i) $F(a_i) = b_i$ if $a_i \in \{a_1, \ldots, a_n\}$

ii) $F(a) = a$ if $a \notin \{a_1, \ldots, a_n\}$

So in place of $t[F]$ above, we write $t[c/b, e/c, f/d]$. Identical renamings may be omitted in this notation.

The following properties are obvious consequences of the definition of renaming

P 2.33 $|t[F]| = |t|$

P 2.34 $(u \frown v)[F] = (u[F]) \frown (v[F])$

P 2.35 $(u \gg v)[F] = (u[F]) \gg (v[F])$

Renaming is also monotonic

P 2.36 $u \preceq v \Rightarrow u[F] \preceq v[F]$

The renaming function F can be many-to-one, therefore the following property is not generally true

$$(t\lfloor A)[F] = (t[F])\lfloor F(A)$$

where $F(A) = \{F(x) | x \in A\}$. A counterexample is given by the following renaming example

$$\begin{aligned}((a)\lfloor\{b\})[b/a] &= \langle\rangle[b/a] && \text{if } a \neq b\\ &= \langle\rangle && \text{by 1)}\\ &\neq \langle b\rangle\\ &= \langle b\rangle\lfloor\{b\}\\ &= ((a)[b/a])\lfloor\{b\} && \text{since } F(a) = b \text{ and } F(b) = b\end{aligned}$$

However the property is true if F is one-to-one (injection)

P 2.37 $(t\lfloor A)[F] = (t[F])\lfloor F(A)$ *if F is one-to-one.*

Proof. By induction on the length of t.

Base case. $(\langle\rangle\lfloor A)[F] = (\langle\rangle[F])\lfloor A = \langle\rangle$ by 1) and def of "$\lfloor$".

Induction hypothesis. $(t\lfloor A)[F] = (t[F])\lfloor F(A)$ if F is one-to-one.

Induction step.

If $a \in A$ then

$$\begin{aligned}(\langle a \cdot t\rangle\lfloor A)[F] &= \langle a \cdot t\lfloor A\rangle[F] && \text{by def of "}\lfloor\text{"}\\ &= \langle F(a) \cdot (t\lfloor A)[F]\rangle && \text{by 2)}\\ &= \langle F(a) \cdot (t[F])\lfloor F(A)\rangle && \text{by hypothesis}\\ &= \langle F(a) \cdot t[F]\rangle\lfloor F(A) && \text{by def of "}\lfloor\text{"}\\ &= (\langle a \cdot t\rangle[F])\lfloor F(A) && \text{by 2),}\\ & && \text{since } F \text{ is one-to-one.}\end{aligned}$$

If $a \notin A$ then

$$\begin{aligned}(\langle a \cdot t\rangle\lfloor A)[F] &= (t\lfloor A)[F] && \text{by def of "}\lfloor\text{"}\\ &= (t[F])\lfloor F(A) && \text{by hypothesis } \square\end{aligned}$$

P 2.38 $t \downarrow A = (t[F]) \downarrow F(A)$ *if F is one-to-one.*

Proof. Follows immediately from P 2.33 and P 2.37. □

2.9 Merging

Traces are very well suited to represent all possible communication patterns between a process and its environment. We would like to be able to express relations between compound processes as equations between the traces of their components. In this section we define a composition function, called *merging*, to describe the parallel composition of two or more processes. This includes their mutual communication embodied by the common actions specified in their "synchronization alphabet" and the successful termination action. Each communication requires the participation of both processes. The reader will recall that two LOTOS processes P and Q composed by means of the parallel composition operator $|[S]|$, where S is a sequence of actions will

1. mutually interleave for actions not in S.
2. mutually synchronize for the actions in S. This means that if P offers action $a \in \{S\}$, so must Q, and the result of their synchronization is a single offer of a.
3. mutually synchronize on the "exit" action δ.

If at any point a required synchronization is not possible, a deadlock occurs. This leads to the following definition. Let

$$\mathcal{M}(v_1, A, v_2) = \{t | v_i \sqsubseteq t \wedge v_i \lfloor A_\delta = t \lfloor A_\delta \wedge |t| = \textstyle\sum_i |v_i| - (v_i \downarrow A_\delta)\}$$

where $i = 1, 2$. Furthermore, for a set of traces T, trace $t \in T$ is said to be "*longest*" in T if $|t'| \leq |t|$ for any $t' \in T$. The merging of two traces u_1 and u_2 in the order imposed by a synchronization alphabet A, denoted by

$u_1|A|u_2$, is the set of longest traces t such that there exists two prefixes v_1 and v_2 of u_1 and u_2, respectively, for which $t \in \mathcal{M}(v_1, A, v_2)$. We illustrate this definition with some examples as follows

$$\langle a, b, \delta\rangle|\{a\}|\langle a, b\rangle = \mathcal{M}(\langle a, b\rangle, \{a\}, \langle a, b\rangle) = \{\langle a, b, b\rangle\}$$

since

$$\mathcal{M}(t, \{a\}, \langle\rangle) = \mathcal{M}(\langle\rangle, \{a\}, t) = \{\langle\rangle\} \text{ where } t \preceq \langle a, b, \delta\rangle$$
$$\mathcal{M}(\langle a\rangle, \{a\}, \langle a\rangle) = \{\langle a\rangle\}$$
$$\mathcal{M}(\langle a, b\rangle, \{a\}, \langle a\rangle) = \mathcal{M}(\langle a\rangle, \{a\}, \langle a, b\rangle) = \{\langle a, b\rangle\} \text{ etc.}$$

Similarly, we can show that

$$\langle a, b, a\rangle|\{b\}|\langle b, c\rangle = \{\langle a, b, a, c\rangle, \langle a, b, c, a\rangle\}$$
$$\langle a, b, \delta\rangle|\{\}|\langle c, \delta\rangle = \{\langle a, b, c, \delta\rangle, \langle a, c, b, \delta\rangle, \langle c, a, b, \delta\rangle\}$$
$$\langle a, b\rangle|\{a, b\}|\langle b, a\rangle = \{\langle\rangle\}$$
$$\langle a, b, c\rangle|\{b, d\}|\langle d, e\rangle = \{\langle a\rangle\}$$

Notice that in the case of two traces with an empty synchronization alphabet, merging does not amount to interleaving as it is defined in [Hoa85] or shuffle as it is called in [Gin66]. They still have to "synchronize" on the special "exit" symbol δ. This ensures that the merging of well-formed traces yields a set of well-formed traces. Also note that merging may not allow (in some cases) the combination of all symbols from u_1 and u_2 into the resulting trace t; it may reach a *deadlock* before it terminates. Deadlock may even occur at the very beginning if the heads of u_1 and u_2 are in A and are not equal. In case of deadlock, no further symbols from u_1 and u_2 can be combined, in which case we can write $deadlock(u_1, A, u_2)$. Therefore we can define

$$deadlock(u_1, A, u_2) \equiv \forall t \in u_1|A|u_2.\neg(u_1 \sqsubseteq t \wedge u_2 \sqsubseteq t)$$

An alternative definition of deadlock can be formalized using projection

$$deadlock(u_1, A, u_2) \equiv \forall t \in u_1|A|u_2.t\lfloor A_\delta \prec u_1\lfloor A_\delta \vee t\lfloor A_\delta \prec u_2\lfloor A_\delta$$

The first, fourth and fifth examples above are examples of deadlock. Note that in the first line the deadlock results by the fact that the traces cannot synchronize on δ. The following property is a consequence of this definition

P 2.39 $t \in u_1|A|u_2 \Rightarrow \exists v_1, v_2.v_i \preceq u_i \wedge t \in v_1|A|v_2 \wedge \neg deadlock(v_1, A, v_2)$

The following two rules give a method for computing whether a given trace is a possible merging of a pair of traces u and v w.r.t. to a synchronization alphabet A or not. Let u_{head}, v_{head}, and u_{tail}, v_{tail} denote the heads and tails of u and v respectively

1) $\langle\rangle \in u|A|v \equiv (u = \langle\rangle \wedge v = \langle\rangle)$
$\vee\ (u \neq \langle\rangle \wedge v = \langle\rangle \wedge u_{head} \in A_\delta)$
$\vee\ (v \neq \langle\rangle \wedge u = \langle\rangle \wedge v_{head} \in A_\delta)$
$\vee\ (u \neq \langle\rangle \wedge v \neq \langle\rangle \wedge u_{head} \neq v_{head}$
$\wedge\ u_{head} \in A_\delta \wedge v_{head} \in A_\delta)$

2) $\langle a \cdot t\rangle \in u|A|v \equiv (u \neq \langle\rangle \wedge a \notin A_\delta \wedge u_{head} = a \wedge t \in u_{tail}|A|v)$
$\vee\ (v \neq \langle\rangle \wedge a \notin A_\delta \wedge v_{head} = a \wedge t \in u|A|v_{tail})$
$\vee\ (u \neq \langle\rangle \wedge v \neq \langle\rangle \wedge a \in A_\delta$
$\wedge\ u_{head} = v_{head} = a \wedge t \in u_{tail}|A|v_{tail})$

The following properties are obvious consequences of the definition of merging

P 2.40 $u_1|A|u_2 = u_2|A|u_1$ *(merging is commutative)*

P 2.41 $u_1 \in B_\delta^* \wedge u_2 \in C_\delta^* \Rightarrow u_1|A|u_2 \subseteq (B \cup C)_\delta^*$

P 2.42 $t \in u_1|A|u_2 \Rightarrow |t| \leq |u_1| + |u_2|$

P 2.43 $t \in u_1|A|u_2 \wedge \neg deadlock(u_1, A, u_2) \Rightarrow |t| = |u_1| + |u_2| - (u_1 \downarrow A_\delta)$

P 2.44 $t \in u_1|A|u_2 \wedge \neg deadlock(u_1, A, u_2) \Rightarrow t \downarrow A_\delta = u_i \downarrow A_\delta$
for $i = 1, 2$.

P 2.45 $t \in u_1|\{\}|u_2 \Rightarrow |t| = |u_1| + |u_2| - (u_1 \downarrow \delta + u_2 \downarrow \delta - 1)$

The next property describes the prefix closure of merging in terms of its operands

P 2.46 $pref(u_1|A|u_2) = \{t | \exists v_1, v_2 . v_i \preceq u_i \wedge t \in \mathcal{M}(v_1, A, v_2)\}$

Proof. We only give an informal proof of this property. If $t \in u_1|A|u_2$, then t is the longest trace formed by the prefixes v_i of u_i, for $i = 1, 2$, such that $t \in \mathcal{M}(v_1, A, v_2)$ (definition of merging). Therefore, for every prefix of v_i there must be a prefix of t for which this condition is satisfied, and vice-versa. Hence the prefix closure of t is the set of traces formed by the mergings of the prefixes of u_1 and u_2. □

An obvious consequence of this property is that merging preserves the inclusion ordering of the prefix closures of the merged traces

P 2.47 $v_1 \preceq u_1 \wedge v_2 \preceq u_2 \Rightarrow pref(v_1|A|v_2) \subseteq pref(u_1|A|u_2)$

Next we describe the distribution of projection through merging

P 2.48 $t \in \mathcal{M}(v_1, A, v_2) \Rightarrow t\lfloor B \in \mathcal{M}(v_1\lfloor B, A, v_2\lfloor B)$

Proof.

$t \in \mathcal{M}(v_1, A, v_2)$

$\Rightarrow$ [by the definition of $\mathcal{M}$, and since $\lfloor$ is monotonic for $\sqsubseteq$ and $=$]

$v_i\lfloor B \sqsubseteq t\lfloor B \wedge v_i\lfloor A_\delta\lfloor B = t\lfloor A_\delta\lfloor B \wedge |t\lfloor B| = \sum_i |v_i\lfloor B| - v_i \downarrow (A_\delta \cap B)$

$\equiv$ [P 2.2, P 2.5]

$v_i\lfloor B \sqsubseteq t\lfloor B \wedge v_i\lfloor B\lfloor A_\delta = t\lfloor B\lfloor A_\delta \wedge |t\lfloor B| = \sum_i |v_i\lfloor B| - (v_1\lfloor B) \downarrow A_\delta$

$\equiv$ [definition of $\mathcal{M}$]

$t\lfloor B \in \mathcal{M}(v_1\lfloor B, A, v_2\lfloor B)$ □

P 2.49 $(u_1|A|u_2)\lfloor B \subseteq u_1\lfloor B|A|u_2\lfloor B$

Proof. It suffices to prove that this relation holds for their prefix closures. For every trace s

$s \in pref(u_1|A|u_2)\lfloor B$

$\equiv$ [definition of $\lfloor\rfloor$]

$\exists t.t \in pref(u_1|A|u_2) \wedge s = t\lfloor B$

$\equiv$ [P 2.46]

$\exists t, v_1, v_2.v_i \preceq u_i \wedge t \in \mathcal{M}(v_1, A, v_2) \wedge s = t\lfloor B$

$\Rightarrow$ [$\lfloor$ is monotonic, and P 2.48]

$\exists t, v_1, v_2.v_i\lfloor B \preceq u_i\lfloor B \wedge t\lfloor B \in \mathcal{M}(v_1\lfloor B, A, v_2\lfloor B) \wedge s = t\lfloor B$

$\Rightarrow$ [obviously]

$\exists w_1, w_2.w_i \preceq u_i\lfloor B \wedge s \in \mathcal{M}(w_1, A, w_2)$

$\equiv$ [P 2.46]

$s \in pref(u_1\lfloor B|A|u_2\lfloor B)$ □

Consider the following example

$(\langle a, b, a\rangle|\{b\}|\langle b, c\rangle)\lfloor\{a, c\} = \{\langle a, a, c\rangle, \langle a, c, a\rangle\}$

If the projection is applied on the operands, some of the symbols of the

synchronization alphabet might be omitted, and in this case merging would yield a larger set of traces

$$\langle a,b,a\rangle \lfloor \{a,c\} | \{b\} | \langle b,c\rangle \lfloor \{a,c\} = \{\langle a,a,c\rangle, \langle a,c,a\rangle, \langle c,a,a\rangle\}$$

If no such symbols are omitted, projection would distribute through merging as follows

P 2.50 $(u_1|A|u_2)\lfloor B = u_1\lfloor B|A|u_2\lfloor B \qquad \text{if } A \subseteq B$

For example

$$\begin{aligned}(\langle a,b,a\rangle|\{b\}|\langle b,c\rangle)\lfloor\{a,b\} &= \langle a,b,a\rangle\lfloor\{a,b\}|\{b\}|\langle b,c\rangle\lfloor\{a,b\} \\ &= \{\langle a,b,a\rangle\}\end{aligned}$$

It follows that

P 2.51 $t \in u_1|A|u_2 \Rightarrow t \downarrow B \leq (u_1 \downarrow B + u_2 \downarrow B)$

P 2.52 $t \in u_1|A|u_2 \wedge \neg deadlock(u_1, A, u_2)$
$\Rightarrow t \downarrow B = u_1 \downarrow B + u_2 \downarrow B - u_1 \downarrow (A \cap B)$

Finally, we extend the definition of merging to operate on sets of traces. We have

$$U|A|V = \{t | \exists u, v . u \in U \wedge v \in V \wedge t \in u|A|v\}$$

This definition enjoys a number of interesting properties. It provides a means for describing the associativity of merging

P 2.53 $(U|A|V)|A|T = U|A|(V|A|T)$

However, it is not in general true that

$$(U|A|V)|B|T = U|A|(V|B|T)$$

As a counterexample let

$U = \{\langle a \rangle\}$, $V = \{\langle a \rangle\}$, $T = \{\langle b \rangle\}$, $A = \{b\}$, and $B = \{a\}$

Merging is monotonic for the $\subseteq$-ordering of sets of traces

P 2.54 $T \subseteq U \Rightarrow T|A|V \subseteq U|A|V$

The last two properties are related to merging and prefix closure.

P 2.55 *The merging of prefix-closed sets of traces is prefix-closed.*

Proof. Let $i = 1, 2$ and U_1, U_2 be two prefix-closed sets of traces. Choose s and t such that

$s \frown t \in U_1|A|U_2$

$\equiv$ [definition of merging]

$\exists u_i . u_i \in U_i \wedge s \frown t \in u_1|A|u_2$

$\Rightarrow$ [definition of prefix closure]

$\exists u_i . u_i \in U_i \wedge s \in pref(u_1|A|u_2)$

$\equiv$ [P 2.46]

$\exists v_i, u_i . u_i \in U_i \wedge v_i \preceq u_i \wedge s \in \mathcal{M}(v_1, A, v_2)$

$\Rightarrow$ [definition of merging]

$\exists v_i, u_i . u_i \in U_i \wedge v_i \preceq u_i \wedge s \in v_1|A|v_2$

$\Rightarrow$ [U_i is prefix-closed]

$\exists v_i . v_i \in U_i \wedge s \in v_1|A|v_2$

$\equiv$ [definition of merging]

$s \in U_1|A|U_2$

which proves that $U_1|A|U_2$ is prefix-closed. □

P 2.56 $pref(U|A|V) = pref(U)|A|pref(V)$

Proof. From P 2.20 we have

$$U \subseteq pref(U) \wedge V \subseteq pref(V)$$

$\Rightarrow$ [P 2.54]

$$U|A|V \subseteq pref(U)|A|V$$
$$\wedge pref(U)|A|V \subseteq pref(U)|A|pref(V)$$

$\Rightarrow$ [$\subseteq$ is transitive]

$$U|A|V \subseteq pref(U)|A|pref(V)$$

$\Rightarrow$ [$pref$ is monotonic for $\subseteq$]

$$pref(U|A|V) \subseteq pref(pref(U)|A|pref(V))$$

$\equiv$ [P 2.55]

$$pref(U|A|V) \subseteq pref(U)|A|pref(V)$$

On the other hand

$$t \in pref(U)|A|pref(V)$$

$\equiv$ [definition of merging]

$$\exists u, v.u \in pref(U) \wedge v \in pref(V) \wedge t \in u|A|v$$

$\Rightarrow$ [P 2.20]

$$\exists u, v.u \in pref(U) \wedge v \in pref(V) \wedge t \in pref(u|A|v)$$

$\Rightarrow$ [P 2.47, and definition of $pref$]

$$\exists u, v.u \in U \wedge v \in V \wedge t \in pref(u|A|v)$$

$\equiv$ [definition of merging and $pref$]

$$t \in pref(U|A|V)$$

which proves that $pref(U)|A|pref(V) \subseteq pref(U|A|V)$ □

2.10 Disruption

The sequential composition uses δ as a glue which sticks two traces u and v together. We define another composition function, called *disruption*, that does not need the glue to stick a prefix of u and v together, and if the glue occurs, v cannot stick. It is denoted by $u[> v$, and can be defined as follows

$$u[> v = \{t \frown s | t \preceq u \wedge (if\ \neg(\langle\delta\rangle\ in\ t)\ then\ s = v\ else\ s = \langle\rangle)\}$$

For example

$$\langle a, b, \delta\rangle[> \langle a\rangle = \{\langle a\rangle, \langle a, a\rangle, \langle a, b, a\rangle, \langle a, b, \delta\rangle\}$$

As we shall see, this operator is useful to describe a situation where a sequence of actions can be nondeterministically interrupted by another sequence. Obviously, if the first sequence exits, disruption is no longer possible.

Disruption is not symmetric. However it enjoys a number of simple properties, including

P 2.57 $t \in u[> v \Rightarrow |t| \leq |u| + |v|$

P 2.58 $u \preceq v \Rightarrow u[> t \subseteq v[> t$

P 2.59 $\langle\rangle[> t = \{t\}$

P 2.60 $t[> \langle\rangle = pref(t)$

Its effect on a singleton trace containing δ in its first argument, is obvious

P 2.61 $\langle\delta\rangle[> t = \{\langle\delta\rangle, t\}$

The next property describes the distribution of projection through disruption

P 2.62 $(u[> v)\lfloor B_\delta = u\lfloor B_\delta [> v\lfloor B_\delta$

It follows that

P 2.63 $t \in u[> v \Rightarrow t \downarrow B_\delta \leq (u \downarrow B_\delta + v \downarrow B_\delta)$

We conclude this section by extending the definition of disruption to operate on sets of traces and listing the deriving properties. We have

$U[> V = \{u[> v | u \in U \wedge v \in V\}$

P 2.64 $T[> (U[> V) = (T[> U)[> V$ *(associativity)*

P 2.65 $S \subseteq U \wedge T \subseteq V \Rightarrow S[> T \subseteq U[> V$ *([> is monotonic for $\subseteq$)*

P 2.66 $pref(U[> V) = pref(U)[> pref(V)$
(pref distributes through [>)

P 2.67 *Let $T = U[> V$ where U and V are prefix-closed, then T is prefix-closed.*

Chapter 3

Trace Characterization of Processes

3.1 Introduction

The *environment* of a process P consists of a set of processes with which P can potentially interact, together with an unspecified, possibly human, observer process who watches P and records every action name as it occurs. Therefore, whenever P interacts with the observer, it performs an observable action. If P can perform two observable actions simultaneously, the observer would have to record one of them first, and then the other. Hence, the order in which he records them should not matter. With the aid of such an observer process, we will be provided eventually with the observable action sequences that P can perform. These sequences represent the traces of P which can be

considered to serve as a means of characterizing processes.

3.2 Traces of processes

A process definition describes the behaviour pattern of a process, by defining the sequences of observable actions that may occur (be observed), over a finite set of actions. The latter is the *process alphabet*, denoted $\alpha(P)$ for a given process P (this notion will be defined precisely on page 45). The behaviour of a process up to some moment in time can then be recorded as a finite-length sequence of observable actions in which the process has participated, which we called a *trace*. In other words, a trace of the behaviour of a process P is a finite sequence of observable actions (in which P has engaged up to some moment in time) recorded by an observer with which P interacts.

Nothing can be observed or recorded before a process has engaged in an observable action. This is represented by the empty trace $\langle\rangle$, which every process has as its shortest possible trace. Therefore the complete set of all possible traces, often called the *trace set*, of any process contains at least the empty trace. Also note that a process may reach a point where it cannot offer anything to the environment; it may reach a deadlock state or engage in an unbounded sequence of unseen actions. Once again nothing can be observed and only the empty trace can be recorded.

Example 3.1 The only trace of the inactive process **stop** is $\langle\rangle$, so its trace set is

$$\{\langle\rangle\}$$ □

Example 3.2 The process **exit** performs the successful termination action δ and then behaves like **stop**, so its trace set is

$\{\langle\rangle, \langle\delta\rangle\}$ □

The traces of the behaviour of a process have finite length. However the trace set of a process can be infinite (i.e. recursive processes).

Example 3.3 Consider a process that describes the externally observable behaviour of a one slot simplex buffer (SB) between two points (as defined in the [ISO88]-tutorial), which can accept only one input at a time, and then perform an output. Its behaviour is defined as

process SB [*input*, *output*] :=
    input; *output*; SB [*input*, *output*]
endproc

and its trace set is

$\{\ \langle\rangle, \langle input\rangle, \langle input, output\rangle, \langle input, output, input\rangle, \ldots\ \}$

which shows the traces of a partial execution of the process SB. □

Not every action, in which a process is ready to engage at a given moment in time, will actually occur: the choice may depend on the environment in which the process is placed, or on the nondeterministic behaviour of the process. Thus not every possible trace of a given process will actually be recorded every time the process is "executed".

Example 3.4 The process SB (Example 3.3) is deterministic, in that no communication can be refused on either its input or output channels. It

behaves as a reliable channel, which is bound to output (eventually) whatever was input. Consider the introduction of the process, unreliable simplex buffer (SB'), which accepts an input, and then nondeterministically either immediately outputs what was input or loses it

```
process SB' [input, output] :=
        input; ( output; SB' [input, output]
                 []
                 i; SB' [input, output] )
endproc
```

Its trace set is

$\{ \langle\rangle, \langle input\rangle, \langle input, output\rangle, \langle input, input\rangle, \ldots \}$

Only one of the two traces of length two can actually occur in an actual execution. The choice between them is nondeterministic. Note that i (the unobservable action) models the loss of input, which cannot be observed or recorded in the trace model. □

3.3 From labelled transitions to traces

The operational semantics of LOTOS [ISO88, BB87] enables us to derive actions, in which a process (behaviour expression) can engage, from the structure of the expression itself. More precisely, a behaviour expression B can perform an action x and then behave like B' (another behaviour expression). What we derive here are labelled transitions denoted $B \xrightarrow{x} B'$. We shall write $B \xrightarrow{s} B'$, where s denote a sequence of actions (not necessarily observable)

$x_1 \ldots x_n$ $(n \geq 0)$, to mean that for some B_i $(0 \leq i \leq n)$

$$B = B_0 \stackrel{x_1}{\Rightarrow} B_1 \stackrel{x_2}{\Rightarrow} B_2 \cdots \stackrel{x_n}{\Rightarrow} B_n = B'$$

In particular for $n = 0$ we have $B \stackrel{\epsilon}{\longrightarrow} B$, where ϵ is the empty sequence. Now consider the result(s) of the execution of a trace $\langle a_1, a_2, \ldots, a_n \rangle$ on B. The result may be any B' for which $B \stackrel{s}{\longrightarrow} B'$, where $s = i^{k_0} a_1 i^{k_1} a_2 i^{k_2} \ldots a_n i^{k_n}$ $(k_i \geq 0)$ and i^k denotes a sequence of k i-actions, that is, an arbitrary number of internal events may occur before, among and after the a_i. This leads to the definition of the *trace relation* '$\Longrightarrow$', whose purpose is to abstract from the invisible actions that are derived by the transition relation when applied to a behaviour expression.

Definition 3.1 (Trace Relation) *Let t denote a trace $\langle a_1, a_2, \ldots, a_n \rangle$ then we have $B \stackrel{t}{\Longrightarrow} B'$ whenever there exists a sequence $i^{k_0} a_1 i^{k_1} a_2 i^{k_2} \ldots a_n i^{k_n}$ $(k_i \geq 0)$, denoted by s, such that $B \stackrel{s}{\longrightarrow} B'$.*

This implies that $B \stackrel{\langle\rangle}{\Longrightarrow} B'$ whenever $B \stackrel{i^k}{\longrightarrow} B'$ $(k \geq 0)$, and that, for any B, $B \stackrel{\langle\rangle}{\Longrightarrow} B$ (in this case $k = 0$).

If $B \stackrel{t}{\Longrightarrow} B'$, we say that B' is an element of B/t (B after t). In other words if t is a trace of B then B/t is a set of behaviour expressions whose elements behave the same as B from the time after B has been engaged in all the actions recorded in the trace t.

Definition 3.2 *Let B be a behaviour expression and t a possible trace of B, then $B/t = \{B' | B \stackrel{t}{\Longrightarrow} B'\}$.*

If t is not a possible trace of B then B/t is undefined. The set $B/\langle\rangle$ contains B, together with, possibly, other behaviour expressions that do not behave

like B, since B may engage in an arbitrary number of internal events to reach a point where it does not accept what it originally did. As an example, consider the following behaviour expression

B = (i; stop [] a; stop)

for which $B/\langle\rangle$ contains stop (after engaging into the i-action), together with B, which is ready to accept a. In general, the behaviour of a process after engaging into a trace t is not unique for a given B and t, as B/t is a set of behaviours. Another example is given by the behaviour expression B such that

B = (a; stop) [] (a; b; stop)

Clearly $B/\langle a\rangle$ contains stop together with a behaviour expression that is ready to accept b. This operator differs from the one in [Hoa85], which yields a single process as opposite to a set of processes.

At this point, we can define Milner's *observation equivalence* $\approx$ [Mil80] by means of the / operator. The behaviour of two processes will be distinguishable if their possible derivations, w.r.t. / operator, differ.

Definition 3.3 *Let $n \geq 0$, we define*

1) $\forall B_1, B_2.B_1 \approx_0 B_2$

2) $B_1 \approx_{n+1} B_2 \equiv \forall t.$

$$B_1' \in B_1/t \Rightarrow (\exists B_2' \in B_2/t.B_1' \approx_n B_2') \wedge$$
$$B_2' \in B_2/t \Rightarrow (\exists B_1' \in B_1/t.B_1' \approx_n B_2')$$

3) $B_1 \approx B_2 \equiv \forall n.B_1 \approx_n B_2$

Note that the sequence of equivalence relations $\{\approx_n \mid n \geq 0\}$ on processes form a decreasing chain of finer and finer relations.

An interesting survey on observational equivalence verification algorithms of finite state processes can be found in [BS87].

A behaviour expression B may engage in an observable action on the very first step or after performing an arbitrary number of internal events. We will call the set of such actions the *initials* of B, denoted $initials(B)$.

Definition 3.4 *Let S be a set of behaviour expressions, B a behaviour expression, and a an observable action. We define the* initials set *by means of the following two rules*

1) $initials(B) = \{a \in \alpha(B) | \exists B'.B \stackrel{\langle a \rangle}{\Longrightarrow} B'\}$

2) $initials(S) = \bigcup_{B \in S} initials(B)$

The choice of which of these actions, if any, will actually occur depends (at least in part) on environmental factors. Notice that stop cannot offer anything to the environment, nor it can perform internal actions, and so its initials set is empty. Also note, if a is an initial of B ($a \in initials(B)$) then $\langle a \rangle$ must be a possible trace of B, and vice-versa.

Example 3.5 Consider the following behaviour expression

$B =$ (a; stop) [] (i; (b; stop [] i; c; stop))

It is clear that B may perform the action a on its very first step, or b after engaging into one i-action, or c after engaging into two i-actions. Therefore

$initials(B) = \{a, b, c\}$

Since $B/\langle a \rangle$ is a singleton set containing stop, it follows that

$initials(B/\langle a \rangle) = \{\}$ □

The trace set of a behaviour expression B, denoted $traces(B)$, consists of the action sequences derived by means of the trace relation.

Definition 3.5 *Let S be a set of behaviour expressions, B a behaviour expression. We define the* trace set *by means of the following two rules*

1) $traces(B) = \{t | \exists B'.B \stackrel{t}{\Longrightarrow} B'\}$

2) $traces(S) = \bigcup_{B \in S} traces(B)$

The trace set of B contains the empty trace, because $\langle\rangle$ is a trace of the behaviour of every process up to the moment that it engages in its very first action. Also, every nonempty trace in $traces(B)$ begins with an action a, that is an initial of B, and its tail must be a possible trace of a process in $B/\langle a \rangle$. This implies that

P 3.1 $\langle\rangle \in traces(B)$

P 3.2 $traces(B) \subseteq \alpha(B)^*$

Furthermore, $\langle a \rangle \in traces(B)$ whenever $\langle a \cdot t \rangle \in traces(B)$. That is, if $\langle a \cdot t \rangle$ is a trace of a process up to some moment, then $\langle a \rangle$ must have been a trace of that process up to some earlier moment. This leads to the following property

P 3.3 $u \frown v \in traces(B) \Rightarrow u \in traces(B)$

Therefore

P 3.4 *The trace set of every process is prefix-closed.*

The trace set of B/t, provided t is a possible trace of B, can be defined as follows

P 3.5 $traces(B/t) = \{u | t \frown u \in traces(B)\}$ *if* $t \in traces(B)$

Finally, the initials of B can be defined in terms of its trace set

P 3.6 $initials(B) = \{a | \langle a \rangle \in traces(B)\}$

Notice that there is a close relationship between the traces of a process and its action tree. For any node in the tree, the trace t of the behaviour of a process up to the time when it reaches that node (or state) is just the sequence of observable actions encountered on the path leading from the root of the tree to that node. For instance, we may imagine that B and the elements of B/t denote simply tree nodes, or states, rather than behaviour expressions.

Two processes are said to be *trace-equivalent* if and only if they have the same trace set. More formally

Definition 3.6 *Two behaviour expressions* B_1 *and* B_2 *are* trace-equivalent *if and only if* $traces(B_1) = traces(B_2)$.

This simple equivalence relation coincides with Milner's first equivalence $\approx_1$ (Definition 3.3) [Mil80].

3.4 Trace set semantics

In this section, we present a *trace set semantics* for LOTOS which consists of a set of rules and axioms, presented in a denotational style. this semantics provides a means to systematically derive the set of all possible traces (trace set) that a behaviour expression may perform from the structure of the expression itself. More precisely, given an expression B, we use the function $traces(B)$ (definition 3.5) to yield that set. The result is the interpretation of a LOTOS text in terms of a trace set. A similar approach was used to develop the trace set semantics of CSP [Hoa85]. It can be argued that modelling the behaviour of a process in terms of traces does not fully determine all possible observable aspects of its behaviour. This point will be addressed in the next chapter. However, we are primarily interested in determining the range of possible future behaviours of a process by giving possible positive information about what might happen.

In the definitions below, when a set of traces T for a given behaviour is defined as a function of sets of traces $T_1, \ldots, T_n$ of other behaviours, T will be undefined whenever any of $T_1, \ldots, T_n$ is undefined.

3.4.1 Inaction and successful termination

As mentioned in Example 3.1, **stop** has only one trace

T 3.1 $traces(\textbf{stop}) = \{\langle\rangle\}$

Hence we regard stop as a dead process which is unable to perform any action or to interact with any other process. It was also mentioned in Example 3.2 that the first and the only action of exit is successful termination, so it has only two traces

T 3.2 $traces(\text{exit}) = \{\langle\rangle, \langle\delta\rangle\}$

3.4.2 Action prefix and hiding

The action prefix behaviour expression $(a;\ B)$, where a is an observable action, is capable of performing a and then behaving like B. Because the environment may not participate in action a, it must have the empty trace, and every nonempty trace must have a as its head and a possible trace of B as its tail

T 3.3 $traces(a;\ B) = \{\langle\rangle\} \cup \{\langle a \cdot t\rangle | t \in traces(B)\}$

If B is prefixed with an internal action i, then nothing can be observed, or recorded, until B engages into an observable action

T 3.4 $traces(\text{i};\ B) = traces(B)$ *if B does not diverge.*

We say that B *diverges* if

$$\forall k \geq 0 . \exists B' . B \xrightarrow{i^k} B'$$

This means that B can progress invisibly by engaging in an unbounded sequence of internal actions, and refuse to respond to the requests of its environment, in which case its trace set is undefined. Another kind of behaviour

expressions that can have a similar behaviour is one that involves hiding. The latter provides a means to transform observable actions of a process into unobservable ones; it introduces i-actions implicitly in a behaviour expression. Therefore a behaviour expression B may diverge immediately on hiding actions in A, where A is a sequence of observable actions, in which case we shall write $diverges(\{B\}, \{A\})$. Thus we can define

$$diverges(S, \{A\}) \equiv \forall n. \exists t \in traces(S) \cap \{A\}^*. |t| > n$$

where S is a set of behaviour expressions. The trace set of (hide A in B) can be obtained from the trace set of B by simply removing all occurrences of the actions in A from its elements

T 3.5 $traces(\text{hide } A \text{ in } B) = \{t \lfloor (\alpha(B) - \{A\}) | t \in traces(B)\}$
if $\forall t \in traces(B). \neg diverges(B/t, \{A\})$

However, this is undefined if

$$\exists t \in traces(B). diverges(B/t, \{A\})$$

The last two rules are restricted to the case where the process does not diverge. We do not regard this as a serious limitation, since divergence is never an intentional result in the attempted definition of a process.

At this point we are able to define precisely the alphabets of compound behaviours in terms of the alphabets of their components. This is expressed by the following rules

1) $\alpha(\mathbf{stop}) = \{\}$
2) $\alpha(\mathbf{exit}) = \{\delta\}$
3) $\alpha(a;\ B) = \{a\} \cup \alpha(B)$
4) $\alpha(\mathbf{i};\ B) = \alpha(B)$ if B does not diverge.

5) $\alpha(\text{hide } A \text{ in } B) = \alpha(B) - \{A\}$
if $\forall t \in traces(B).\neg diverges(B/t, \{A\})$

6) If "process $P[a'_1, \ldots, a'_n] := B_p$ endproc" is a process definition and F is a renaming given by $a_1/a'_1, \ldots, a_n/a'_n$
then $\alpha(P[a_1, \ldots, a_n]) = F(\alpha(B_p))$

7) $\alpha(B_1 * B_2) = \alpha(B_1) \cup \alpha(B_2)$

where $*$ stands for $[]$, $|[A]|$, $\gg$, and $[>$. Note that whenever an alphabet of a behaviour expression is defined in terms of alphabets of other behaviour expressions, the former is not defined if any of the latter is not.

3.4.3 Process instantiation and recursion

A *process instantiation* "$P\ [a_1, \ldots, a_n]$", where "P" is a *process identifier*, and $[a_1, \ldots, a_n]$ is a list of observable actions (called *actual gates*), refers to a process definition that must exist somewhere in the specification, whose behaviour is described in terms of a list of actions $[a'_1, \ldots, a'_n]$ (called *formal gates*), which corresponds to the process alphabet. The behaviour of instantiation "$P\ [a_1, \ldots, a_n]$" is obtained from the corresponding process definition behaviour by renaming a'_i to become a_i, for $i = 1, \ldots, n$. Therefore

T 3.6 *If* "process $P\ [a'_1, \ldots, a'_n] := B_p$ endproc" *is a process definition then* $traces(P\ [a_1, \ldots, a_n]) = \{t[a_1/a'_1, \ldots, a_n/a'_n] | t \in traces(B_p)\}$

This rule can be applied to discover the trace set of a recursively *guardedly well-defined* (or g.w.d. for short) process [Mil80], since recursion in LOTOS is achieved by process instantiation. We say that process P *invokes* process Q if it either contains an instantiation of Q, or it contains an instan-

tiation of a process that invokes Q. We say that process P is *recursive* if it invokes itself. Such a process is said to be g.w.d. or simply guarded if it is capable of performing at least one action (not necessarily observable) before invoking itself. We say that an instantiation of P is *unguarded* if it occurs without an enclosing guard. Therefore the behaviour of a recursive process P that is not g.w.d. may entail an infinite sequence of instantiations of P, in which case P diverges without even performing internal actions, and so its trace set is undefined.

3.4.4 Choice

The behaviour expression $(B_1 [] B_2)$ denotes a process which behaves either like B_1 or like B_2, The choice of which one would be selected can be controlled by the environment, provided that this control can be exercised on the very first action. Therefore, every trace of $(B_1 [] B_2)$ must be a trace of B_1 or a trace of B_2, hence

T 3.7 $traces(B_1 [] B_2) = traces(B_1) \cup traces(B_2)$

3.4.5 Parallel composition

The behaviour expression $(B_1 |[A]| B_2)$, where A is a sequence of observable actions, describes a composition in which B_1 and B_2 must synchronize on the actions in A. It is able to perform any action not in A that either component is ready to perform, or any action in A or δ that both components are ready

to perform. Therefore, every trace of $(B_1|[A]|B_2)$ depends on the traces of B_1 and B_2, and on A, as expressed by the following rules

1) If $B_1' \in B_1/t$ and $t\lfloor\{A\}_\delta = \langle\rangle$ then $B_1'|[A]|B_2 \in (B_1|[A]|B_2)/t$
2) If $B_2' \in B_2/t$ and $t\lfloor\{A\}_\delta = \langle\rangle$ then $B_1|[A]|B_2' \in (B_1|[A]|B_2)/t$
3) If $B_1' \in B_1/t$ and $B_2' \in B_2/t$ and $t\lfloor\{A\}_\delta = t$
   then $B_1'|[A]|B_2' \in (B_1|[A]|B_2)/t$

These rules are directly derived from the inference rules of the parallel composition operator in LOTOS. They say essentially that every trace t of a parallel composition expression is formed by sub-sequences containing actions not in $\{A\}_\delta$ that either component expression can perform at some moment in time, and sub-sequences containing only actions of $\{A\}_\delta$ that both components can perform at some other moment in time. Clearly, if one of the components reaches a state where it can only perform a sequence of actions of $\{A\}_\delta$, it is forced to wait until its "partner"(the other component) is ready to perform the same sequence. Therefore, if t_i is a possible trace of a component expression B_i $(i = 1, 2)$ whose actions are all recorded in t, we have

$$t_i \sqsubseteq t \wedge t_i\lfloor\{A\}_\delta = t\lfloor\{A\}_\delta$$

Also, the length of t depends on the length of the t_i's in that every action recorded in either one of the t_i's and not in $\{A\}_\delta$, or every action of $\{A\}_\delta$ recorded in both t_i's, is also recorded in t. Since the trace set of every process is prefix-closed (P 3.4), it follows that

$$\begin{aligned}
& t \in traces(B_1|[A]|B_2) \\
\equiv\ & \text{[rules 1), 2), and 3) above]} \\
& \exists t_1, t_2 . t_i \in traces(B_i) \wedge t \in \mathcal{M}(t_1, \{A\}, t_2) \\
\equiv\ & \text{[definition of merging]} \\
& \exists t_1, t_2 . t_i \in traces(B_i) \wedge t \in t_1|\{A\}|t_2
\end{aligned}$$

Therefore, the trace set of a parallel composition expression can be described by means of merging of its components trace sets

T 3.8 $traces(B_1|[A]|B_2) = traces(B_1)|\{A\}|traces(B_2)$

When A is empty, the expression given above is written $(B_1|||B_2)$, to describe pure interleaving, in which case the components B_1 and B_2 don't have to synchronize, except on the successful termination action. Therefore

T 3.9 $traces(B_1|||B_2) = traces(B_1)|\{\}|traces(B_2)$

When A includes all actions that are in the alphabets of both components, we write $(B_1||B_2)$, to describe the composition of dependent processes which have to synchronize on every action. It follows that

T 3.10 $traces(B_1||B_2) = traces(B_1)|\alpha(B_1) \cup \alpha(B_2)|traces(B_2)$

3.4.6 Sequential composition

The behaviour expression $(B_1 \gg B_2)$ describes a composition, often referred to as the enabling operator, that enables the execution of B_2 if B_1 terminates successfully by performing the successful termination action via the exit process. The occurrence of this transition is hidden; δ is transformed into the internal action i. Therefore, every trace of $(B_1 \gg B_2)$ must be the result of the sequential composition (over traces) of a pair of possible traces of B_1 and B_2, and conversely

T 3.11 $traces(B_1 \gg B_2) = \{u \gg v | u \in traces(B_1) \wedge v \in traces(B_2)\}$
if B_2 does not diverge.

This rule is also restricted to the case where the second argument of the sequential composition operation does not diverge, since the latter introduces implicitly unobservable actions into a behaviour expression, and so it is potentially divergent. As an example consider the following behaviour

$B = \mathbf{exit} \gg B$

where B invokes itself in the second part of the expression, and, in this case, it engages into an unbounded sequence of internal actions. Obviously, if B_1 diverges, then so does $(B_1 \gg B_2)$. However, this is not mentioned in the rule above, because, in this case, the sequential composition of B_1 and B_2 does not, specifically, yield such behaviour.

3.4.7 Disruption

The behaviour expression $(B_1[> B_2)$ describes another kind of composition, often referred to as the disabling operator, that allows the occurrence of an initial action of B_2 at each point of the execution of B_1, in which case control is transferred to B_2, leaving B_1. Once B_1 has terminated successfully, B_2 can no longer disrupt B_1. Hence every trace of $(B_1[> B_2)$ is either a trace of B_1 with δ at the end or a trace constructed from a pair of possible traces of B_1 and B_2, respectively, by simply putting them together in this order. More formally, since the trace set of every process is prefix-closed we can write

T 3.12 $traces(B_1[> B_2) = traces(B_1)[> traces(B_2)$

Note that since the trace set of B_2 contains at least the empty trace, the trace set of $B_1[> B_2$ will contain at least the elements of the trace set of B_1.

3.5 More on traces of processes

Tracing the behaviour of a formally described process, by examining some or all of its traces, is a common practice when trying to get a feel of the process's behaviour. This can be achieved using an automatic trace generator, which implements the trace rules. Such a tool was developed for CSP [Kou87] which translates the latter to Prolog and generates all traces up to termination or recursion. However, such tools can only generate a subset of the traces of a process with an infinite behaviour, which is the case of most interesting processes. Hence, it is always desirable, for design and verification purposes, to formulate and prove general statements about the trace set of a process in terms of its alphabet using some of the operators on traces, introduced in Chapter 2. Once again, this can be achieved with the aid of our trace rules.

Our analysis of recursive processes will be helped by defining a means of approximating the behaviour of processes. We adapt the following definition from [Hoa85].

Definition 3.7 *If P is a non-divergent process and n a natural number, we define $(P \uparrow n)$ as a process which behaves like P for its first n observable actions, and then becomes* **stop**.

It follows that

T 3.13 $traces(P \uparrow n) = \{t | t \in traces(P) \wedge |t| \leq n\}$

The following properties are obvious consequences

P 3.7 $traces(P \uparrow 0) = traces(\mathbf{stop}) = \{\langle\rangle\}$

P 3.8 $traces(P \uparrow m) \subseteq traces(P \uparrow n) \qquad if\ m \leq n$

P 3.9 $traces(P) = \bigcup_{n \geq 0} traces(P \uparrow n)$

The last rule will be useful in the analysis of recursive constructions. More specifically, we will apply these identities to carry proofs by induction on recursively defined processes that do not diverge. Note that such processes are obviously guarded. Moreover, it was shown in [Hoa85] that an equation defining a g.w.d. process in this way has a unique solution.

Example 3.6 Let us consider the process simplex buffer SB (Example 3.3 on page 36) and apply the rules to formally define its trace set. We want to prove that

$traces(SB) = \bigcup_{n \geq 0} \{t | t \preceq \langle input,\ output \rangle^n\}$

This corresponds exactly to the intuitive result stated, informally, in Example 3.3. By P 3.9 it is sufficient to prove that

$\forall n \geq 0. traces(SB \uparrow n) = \{t | t \preceq \langle input,\ output \rangle^n\}$

This can be done by induction on n.

Proof.

Base case.

$$
\begin{aligned}
& traces(SB \uparrow 0) \\
= \quad & \text{[P 3.7, T 3.1]} \\
& traces(\mathbf{stop}) = \{\langle\rangle\} \\
= \quad & \text{[definition of } n^{th} \text{ power of a trace]} \\
& traces(\{t | t \preceq \langle input,\ output\rangle^0\}
\end{aligned}
$$

Induction hypothesis.

$$traces(SB[input,\ output]\uparrow n) = \{t | t \preceq \langle input, output\rangle^n\}$$

Induction step.

$$
\begin{aligned}
& traces(input;\ output;\ SB[input,\ output]\uparrow n) \\
= \quad & \text{[T 3.3 twice]} \\
& \{\langle\rangle, \langle input\rangle\} \\
& \cup\{\langle input,\ output\rangle \frown t | t \in traces(SB[input,\ output]\uparrow n)\} \\
= \quad & \text{[induction hypothesis]} \\
& \{\langle\rangle, \langle input\rangle\} \cup \{\langle input,\ output\rangle \frown t | t \preceq \langle input,\ output\rangle^n\} \\
= \quad & \text{[definition of } n^{th} \text{ power of a trace, and } \preceq\text{]} \\
& \{t | t \preceq \langle input,\ output\rangle^{n+1}\}
\end{aligned}
$$

□

As the trace rules are defined by induction on the structure of the behaviour expression, we adopt a bottom-up approach to discover the trace set of a given process. For instance the trace set of compound processes can be formulated using the corresponding compound trace sets.

Example 3.7 Consider the definition of a Two Slot Buffer (TSB) using two one slot simplex buffers (as defined in [ISO88]-tutorial), which can accept up to two inputs at a time, and perform one or two outputs depending on what was input. Its behaviour can formally be described as

process TSB [*input*, *output*] :=
  hide *mid* **in**
    SB [*input*, *mid*]
    |[*mid*]|
    SB [*mid*, *output*]
endproc

Intuitively, traces generated by this process are sequences of the symbols *input* and *output* where no more than two consecutive inputs can occur, and where for every prefix the number of *outputs* cannot exceed the number of preceding inputs. This characterization will be (partially) proven later (Example 3.9). Here, we would like to limit ourselves to a simpler characterization to illustrate the concepts we have introduced. Using the results obtained in the previous example, and taking

$T = traces(SB[input; output])$

we can apply the process instantiation rule to write

$$\begin{aligned} traces(SB[input, mid]) &= \{t[input/input, mid/output] | t \in T\} \\ &= \bigcup_{n \geq 0} \{t[input/input, mid/output] | \\ &\qquad t \preceq \langle input,\ output \rangle^n\} \\ &= \{t | t \preceq \langle input,\ mid \rangle^*\} \end{aligned}$$

and similarly,

$traces(SB[mid, output]) = \{t | t \preceq \langle mid,\ output \rangle^*\}$

Therefore, by applying the parallel composition rule (T 3.8 and definition of merging)

$traces(SB[input, mid]|[mid]|SB[mid, output])$
$= \{t | \exists u, v. u \preceq \langle input,\ mid \rangle^* \wedge v \preceq \langle mid,\ output \rangle^* \wedge t \in u|\{mid\}|v\}$

By applying the hiding rule, we obtain

$traces(TSB) = \{\ t\lfloor\{input, output\}\ |\exists u, v. u \preceq \langle input,\ mid \rangle^*$

$$\wedge v \preceq \langle mid,\ output\rangle^* \wedge t \in u|\{mid\}|v\}$$

One can easily accept that this corresponds to the informal behaviour requirement stated above. □

The concept of traces, as a method of characterizing processes, is useful because it allows to interpret the behaviour of a given process as a set of traces, which is suggestive of the process behaviour in a less abstract way than the formal process description. Moreover, they are necessary in the design and validation activities of communicating processes. However, traces are, by themselves, insufficient to model processes, since they do not provide negative information about what might not happen when trying to determine the future of a process's behaviour. The next chapter suggests a model which overcomes this problem.

3.6 Trace specifications

The required behaviour of a process can be described in terms of some observable aspects of its behaviour. The most relevant observation, that we have mentioned, is the trace of actions that occur at some moment in time. This description is then a (trace) predicate S containing a free variable t that stands for an arbitrary trace of the process being defined with, possibly, some other free variables. For example the one slot simplex buffer SB (Example 3.3 on page 36) must satisfy the following trace specification. For every trace t of SB

$$0 \leq t \downarrow input - t \downarrow output \leq 1$$

Also the trace specification of the one slot unreliable simplex buffer SB' (Ex-

ample 3.4 on page 36) can be described as follows

$t \downarrow output \leq t \downarrow input \wedge \neg(\langle output, output \rangle \; in \; t)$

A trace specification describes the characteristics of a process's trace set. Therefore, there is a need for a notion of a process P satisfying its (trace) specification S. In this context, P can be viewed as an implementation of S, since it gives a more structured and detailed description of the system specified in S.

3.6.1 Satisfaction relation

The fact that a process P meets a behaviour requirement S, can be modelled as a binary relation "$\models$". We write $\models$ in an infix manner and $P \models S$ may be read "P *satisfies* S", which means that S is a property that is true for every possible observation of the behaviour of P; or, more formally

$\forall t.t \in traces(P) \Rightarrow S(t)$

In other words, S is true if its variable t takes values observed form the process P. Following Hoare, we will sometimes write $S(t)$ to indicate that a specification S contains a free variable t, whenever there is a need to show how t may be substituted by some more elaborated expression.

The satisfaction relation $\models$ is defined by means of the same rules governing Hoare's sat relation [Hoa85], as follows

1) $P \models true$
2) If $P \models R$ and $P \models S$ then $P \models (R \wedge S)$
3) If $P \models R$ or $P \models S$ then $P \models (R \vee S)$
4) If $\forall n.(P \models S(n))$ then $P \models (\forall n.S(n))$
5) If $\exists n.(P \models S(n))$ then $P \models (\exists n.S(n))$

6) If $P \models R$ and $R \Rightarrow S$ then $P \models S$

Rule 1) states that every process satisfies the predicate *true*; which places no constraints on the behaviour of a process. Note that S does not necessarily "describe" P as indicated for example by this rule. If a process satisfies two different specifications, it also satisfies their conjunction, by rule 2). The latter generalizes to infinite conjunctions in rule 4), where $S(n)$ denotes a specification containing the variable n. Similarly rules 3) and 5) relate $\models$ to disjunction. Rule 6) says that a specification S that is logically implied by another specification R, which is satisfied by a process P, is also satisfied by P. That is so, because every observation described by S is also described by R.

The rules given above provide the most general properties of the satisfaction relation. They apply to all kinds of processes and all kinds of specifications, and are based on the structure of the specification. Our objective is to design a framework for the verification of statements about the behaviour of LOTOS processes, which requires additional rules based on their structure. Such rules should permit proof of the correctness of a compound process to be constructed from a proof of correctness of its parts. We shall give these rules in the next section and refer to them as the *proof rules*. This leads to a Hoare-style proof system for LOTOS.

3.6.2 Proof rules

The proof rules permit the use of formal reasoning to ensure that a LOTOS behaviour expression B meets its specification S. They also provide other means of characterizing processes, in that the behaviour of a process can

be modelled by logical assertions on its traces and, possibly, some other observable aspects of that.

The reader should beware that, in the list of rules given below, different occurrences of the symbol t in the same rule may refer to different traces. Also recall that, as stated at the beginning of section 3.6.1, the variable t is always supposed to be universally quantified in such a context. This shorthand is due to Hoare [Hoa85].

As mentioned before, the process **stop** is completely inactive, and so the only possible observation of its behaviour is the empty trace

S 3.1 $\mathbf{stop} \models (t = \langle\rangle)$

The **exit** process performs the successful termination action δ and then transforms into **stop**

S 3.2 $\mathbf{exit} \models (t = \langle\rangle \vee t = \langle\delta\rangle)$

Every trace of the expression $(a;\ B)$ is either empty or has a as its head and its tail is a possible trace of B. Therefore, the trace specification of B must describe its tail

S 3.3 *If* $B \models S(t)$
then $(a;\ B) \models (t = \langle\rangle \vee (t = \langle a \cdot t'\rangle \wedge S(t')))$

All rules below assume that the behaviour expressions involved in the following properties do not diverge.

The trace specification of B must describe every trace of (i; B), since the i-action is not observable

S 3.4 *If* $B \models S$
then (i; B) $\models S$

The proof rule for hiding is also restricted to the case where the behaviour expression B does not diverge immediately on hiding actions in a sequence A

S 3.5 *If* $B \models (\neg diverge(\{B\}, \{A\}) \wedge S(t))$
then (hide A in B) $\models (\exists u.t = u\lfloor(\alpha(B) - \{A\}) \wedge S(u))$

Every trace of a process instantiation is the result of renaming a trace described by the trace specification of the behaviour expression defining that process

S 3.6 *Let* "process P $[a'_1, \ldots, a'_n] := B_p$ endproc" *be a process definition*
If $B_p \models S(t)$
then P $[a_1, \ldots, a_n] \models (\exists u \in traces(B_p).t = u[a_1/a'_1, \ldots, a_n/a'_n] \wedge S(u))$

Any observation of the behaviour $(B_1 [] B_2)$ must be a possible observation of either B_1 or B_2, and so it must be described by at least one of their trace specifications

S 3.7 *If* $B_1 \models R$
and $B_2 \models S$
then $(B_1 [] B_2) \models (R \vee S)$

The following describes the proof rules associated with the parallel composition, sequential composition, and disruption. This description is based on the fact that every trace of such compound processes can be described as equations between the traces of their components

S 3.8 *If* $B_1 \models R(t)$
and $B_2 \models S(t)$
then a) $(B_1|[A]|B_2) \models (\exists u, v.t \in u|\{A\}|v \wedge R(u) \wedge S(v))$
b) $(B_1 \gg B_2) \models (\exists u, v.t = u \gg v \wedge R(u) \wedge S(v))$
c) $(B_1[> B_2) \models (\exists u, v.t \in u[> v \wedge R(u) \wedge S(v))$

Next, we give the rules governing the / and $\uparrow$ operators on processes. Every trace specification of a behaviour expression B describes the trace $s \frown t$, whenever t is a possible trace of (B/s) and s a possible trace of B, since $s \frown t$ is also a trace of B

S 3.9 *If* $B \models S(t)$
then $\forall B' \in B/s.(B' \models S(s \frown t))$

If n is a natural number, every trace of $(B \uparrow n)$ is also a trace of B, and therefore must be described by any trace specification which B satisfies

S 3.10 $B \models S(t) \equiv \forall n \geq 0.(B \uparrow n) \models S(t) \wedge |t| \leq n$

This is because the set of all traces of B is prefix-closed, thus the left-hand-side asserts a property of all such traces.

We end this section with some examples.

Example 3.8 We want to prove that

$$SB[input, output] \models 0 \leq (t \downarrow input - t \downarrow output) \leq 1$$

where SB is the process describing the simplex buffer (Example 3.3 on page 36). In other words, at any given time there can be at most one undelivered message. By S 3.10 it suffices to prove that

$$\forall n \geq 0. SB[input, output]\uparrow n \models 0 \leq (t \downarrow input - t \downarrow output) \leq 1 \wedge |t| \leq n$$

Since SB does not diverge, we can achieve this by induction on n. Let

$$B = SB[input, output]$$

Thus $SB[input, output] = (input; output; B)$

Proof.

Base case. By P 3.7 and S 3.1, we have

$$B \uparrow 0 \models t = \langle\rangle \wedge |t| \leq 0$$

$\Rightarrow$ $[\langle\rangle \downarrow input = \langle\rangle \downarrow output = 0$, and $|\langle\rangle| = 0]$

$$B \uparrow 0 \models 0 \leq (t \downarrow input - t \downarrow output) \leq 1 \wedge |t| \leq 0$$

Induction hypothesis.

$$B \uparrow n \models 0 \leq (t \downarrow input - t \downarrow output) \leq 1 \wedge |t| \leq n$$

Induction step. By S 3.3 and induction hypothesis, we have

$$(output; B \uparrow n) \models -1 \leq (t \downarrow input - t \downarrow output) \leq 0 \wedge |t| \leq n + 1$$

$\Rightarrow$ [S 3.3]

$$(input; output; B \uparrow n) \models 0 \leq (t \downarrow input - t \downarrow output) \leq 1 \wedge |t| \leq n + 2$$

$\Rightarrow$ [S 3.10]

$$(input; output; B \uparrow n) \uparrow n + 1 \models 0 \leq (t \downarrow input - t \downarrow output) \leq 1 \wedge |t| \leq n + 1$$

$\Rightarrow$ [definition of $\uparrow$]

$$(input; output; B) \uparrow n + 1 \models 0 \leq (t \downarrow input - t \downarrow output) \leq 1$$

$$\wedge |t| \leq n+1 \qquad \square$$

Example 3.9 Consider the two slot buffer TSB (Example 3.7 on page 53), we wish to prove that

$$TSB[input, output] \models 0 \leq (t \downarrow input - t \downarrow output) \leq 2$$

In other words, at any given time there can be at most two undelivered messages. Let

$$A = \{input, output\}$$
$$B' = SB[input, mid] |[mid]| SB[mid, output]$$
$$B = \text{hide } mid \text{ in } B'$$

Also note that $\alpha(B') = \{input, output, mid\}$

Proof. First Note that by S 3.6 and proof of Example 3.8, we have

$$SB[input, mid] \models \exists u.t = u[input/input, mid/output] \wedge 0(\leq u \downarrow input - u \downarrow output) \leq 1$$

$\Rightarrow$ [P 2.38, since the renaming is one-to-one]

$$SB[input, mid] \models 0 \leq (t \downarrow input - t \downarrow mid) \leq 1$$

Similarly, we can show that

$$SB[mid, output] \models 0 \leq (t \downarrow mid - t \downarrow output) \leq 1$$

By S 3.8-*a*) it follows that

$$B' \models \exists u, v.t \in u|\{mid\}|v \wedge 0 \leq (u \downarrow input - u \downarrow mid) \leq 1$$
$$\wedge 0 \leq (v \downarrow mid - v \downarrow output) \leq 1$$

$\Rightarrow$ [P 3.4, P 2.39, P 2.44, and P 2.52]

$$B' \models \exists u, v.t \in u|\{mid\}|v$$
$$\wedge 0 \leq (u \downarrow input - u \downarrow mid + v \downarrow mid - v \downarrow output) \leq 2$$
$$\wedge t \downarrow A = (u \downarrow A + v \downarrow A) \wedge (t \downarrow mid = u \downarrow mid = v \downarrow mid)$$

$\Rightarrow$ [$u \downarrow output = v \downarrow input = 0$]

$$B' \models 0 \leq (t \downarrow input - t \downarrow output) \leq 2$$

$\Rightarrow$ [S 3.5, since clearly $traces(B') \cap \{mid\}^* = \{\langle\rangle\}$]
$B \models \exists u.t = u \lfloor A \wedge 0 \leq (u \downarrow input - u \downarrow output) \leq 2$
$\Rightarrow$ [$(u \lfloor A) \downarrow a = u \downarrow a$ if $a \in A$]
$B \models 0 \leq (t \downarrow input - t \downarrow output) \leq 2$ □

Example 3.10 In this example we would like to consider the behaviour of an unbounded reliable buffer, that will be referred to as UB, which has two channels, one input and one output. It is bound to output (eventually) whatever was input, and the number of inputs performed is unbounded. It can be described in LOTOS using the one slot buffer SB (Example 3.3). This example is due to E. Brinksma.

```
process UB[input, output] :=
        hide mid in
            (input; (UB[input, mid]
                    |[mid]|
                    SB[output, mid]))
endproc
```

Note that UB re-instantiates itself at each input. Each such re-instantiation will cause creation of a new copy of process SB, which performs the outputs. The number of outputs ready to be performed cannot exceed the number of inputs that have been performed. We would like to prove that

$UB[input, output] \models t \downarrow output \leq t \downarrow input$

Once again, by S 3.10 it suffices to prove that

$\forall n \geq 0.UB[input, output]\uparrow n \models t \downarrow output \leq t \downarrow input \wedge |t| \leq n$

and since UB does not diverge, we can achieve this by induction on n. Let

$A = \{input, output\}$

$B' = UB[input, output]$

$B = UB[input, mid]|[mid]|SB[output, mid]$

Proof. We will ignore some of the details of this proof.

Base case. Obvious.

Induction hypothesis.

$$B' \uparrow n \models t \downarrow output \leq t \downarrow input \wedge |t| \leq n$$

Induction step.

Using similar arguments to those used in the previous examples, we show

$$B \uparrow n \models t \downarrow output \leq (t \downarrow input + 1) \wedge |t| \leq n$$

$\Rightarrow$ [S 3.3]

$$(input;\ B \uparrow n) \models t \downarrow output \leq t \downarrow input \wedge |t| \leq n + 1$$

$\Rightarrow$ [S 3.5, since clearly $traces(B) \cap \{mid\}^* = \{\langle\rangle\}$]

$$\textbf{hide } mid \textbf{ in } (input;\ B \uparrow n) \models \exists u.t = u\lfloor A \wedge u \downarrow output \leq u \downarrow input \wedge |u| \leq n + 1$$

$\Rightarrow$ [$(u\lfloor A) \downarrow a = u \downarrow a$ if $a \in A$, and $(u\lfloor A) \downarrow a = 0$ if $a \notin A$]

$$\textbf{hide } mid \textbf{ in } (input;\ B \uparrow n) \models t \downarrow output \leq t \downarrow input \wedge |t| \leq n + 1$$

$\Rightarrow$ [S 3.10, and definition of $\uparrow$]

$$(\textbf{hide } mid \textbf{ in } (input;\ B)) \uparrow n + 1 \models t \downarrow output \leq t \downarrow input \wedge |t| \leq n + 1$$

□

One should note that the unbounded buffer, described above, and an unbounded unreliable buffer UB' , that may lose inputs, satisfy the same trace specification, since their traces t enjoy the property

$$t \downarrow output \leq t \downarrow input$$

The LOTOS description of UB' is similar to that of UB except that we should replace the instantiation of SB by an instantiation of SB'', given by

process SB'' [*output*, *input*] :=
output; *input*; SB'' [*output*, *input*]
[]
i; *input*; SB'' [*output*, *input*]
endproc

The proof that UB' satisfies the above property can be achieved by proving, by induction, that

$SB''[output, input] \models t \downarrow output \leq t \downarrow input + 1$

However, these two processes cannot be regarded as equivalent, even though we cannot distinguish their trace sets. This shows that in general, trace specifications cannot model all observable aspects of a process, and therefore they need to be enriched, so the $\models$ relation can provide means to distinguish such processes. Such means will be discussed in the next chapter.

Chapter 4

Refusals and Failures of Processes

4.1 Introduction

In LOTOS, a process is defined in terms of its behaviour vis-à-vis its environment. Frequently, there will be a choice between several different actions. For instance, whenever there is more than one action possible, if the choice between them is determined externally by the environment, the process is said to be "deterministic". More concisely, a deterministic process may be said to accept each of its traces.

On the other hand, a process may have a range of possible behaviours, but its environment cannot influence or even observe the selection between

the alternatives. Such a process is said to be nondeterministic. The latter may, potentially, accept one of its possible traces, but may also nondeterministically refuse to accept it. In this context, traces can only be used to partially model nondeterministic processes behaviours. Therefore, as already shown in the example at the end of the previous chapter, we cannot rely upon a trace model to fully characterize processes; we need a finer model which supports the representation of all possible externally observable aspects of a process's behaviour, and this includes *refusals*.

4.2 Refusals

The choice of which of the initials of a process P, if any, will actually occur, depends (at least in part) on the environment in which P is placed. Therefore, if X is a set of actions which are initially offered by that environment, the action that actually occurs must be in the intersection $(X \cap \mathit{initials}(P))$. If this intersection is empty, P has reached a deadlock state, and X is a *refusal* of P. However, if P is nondeterministic and the intersection is not empty, it might be possible for P to perform an internal action to reach a new state P' where the intersection $(X \cap \mathit{initials}(P'))$ is empty. In this case, we say that X is a possible refusal of P. A simple example is given by the behaviour expression B such that

B= (i; stop [] a; stop)

Even though it is possible for B to perform a on its very first step, B can, nondeterministically, choose to refuse it and behave like stop.

Unfortunately, P cannot perform actions that are not in its alphabet.

Therefore, any set of actions that is not a subset of the alphabet of P is a refusal of P. However, we define the set of all possible refusals of P, denoted $refusals(P)$, w.r.t. the alphabet of the environment of P, which we take to include the alphabet of P.

The concept of refusals was first introduced in [BHR84] from which we adapt the following definition and properties.

Definition 4.1 *Let S be a set of behaviour expressions, B a behaviour expression, and $\alpha(E)$ the alphabet of an environment E in which B is placed. We define the set of B's and S's refusals by means of the following rules*

1) $refusals(B) = \{X \subseteq \alpha(E) | \exists B'.B \stackrel{()}{\Longrightarrow} B' \wedge X \cap initials(B') = \{\}\}$

2) $refusals(S) = \bigcup_{B \in S} refusals(B)$

This implies that the empty set is a possible refusal of every process

P 4.1 $\{\} \in refusals(B)$

which means that B deadlocks when its environment refuses to cooperate. If B refuses a non-empty set, it can also refuse any subset of that set

P 4.2 $(X \in refusals(B) \wedge Y \subseteq X) \Rightarrow Y \in refusals(B)$

This property provides means for representing the refusal set of B by only listing the $\subseteq$-maximal subsets that are in refusals of B. This is what will normally be done in this thesis. Furthermore, B can refuse a set of actions that it cannot perform, together with any set of actions that it can refuse

P 4.3 $(X \in refusals(B) \wedge Y \subseteq (\alpha(B) - initials(B)))$
$\Rightarrow (X \cup Y) \in refusals(B)$

Note that, although the refusal model makes a single set of "possible" refusals (sets of actions that the process can perform) and "necessary" refusals (sets of actions that the process cannot perform), it is always possible to identify necessary refusals by noting that they are not in the initial set of the process. This reasoning is used often in testing theory [Bri88a].

The refusals of a process consists of sets of observable actions. This leads to a proper treatment of nondeterminism. A behaviour expression B can refuse the set of actions that it cannot perform together with any set of actions that it can refuse. Thus a formal distinction can be made between deterministic and nondeterministic behaviours. B is *deterministic* if and only if

$$\forall t \in traces(B).X \in refusals(B/t) \equiv (X \cap initials(B/t) = \{\})$$

A process is said to be *nondeterministic* if it does not satisfy this property; that is, at some point it may refuse an action in which it can engage, even though the environment is ready for it. A simple example of nondeterminism is represented by the following expression

$$B_1 = (a; \mathbf{stop} \,[]\, i; b; \mathbf{stop})$$

for which

$$initials(B_1) = \{a, b\} \text{ and } refusals(B_1) = \{\{\}, \{a\}\}$$

Thus a can both be accepted and refused after $\langle\rangle$. Another example is given by the following behaviour expression

$$B_2 = (a; b; \mathbf{stop} \,[]\, a; c; \mathbf{stop})$$

where if the alphabet of the environment of B_2 is the same as the alphabet of B_2

$refusals(B_2) = \{\{b, c\}\}$

It also describes a nondeterministic behaviour, since

$\{b\} \in (refusals(B_2/\langle a \rangle) \cap initials(B_2/\langle a \rangle))$

and so does $\{c\}$. While the expression given by

$B_3 = a;$ (b; **stop** [] c; **stop**)

is deterministic, in that it satisfies the above property. Notice that B_2 and B_3 have the same trace and refusal sets.

4.3 Refusals of compound processes

The following rules define the refusals of LOTOS constructions. Using similar arguments to those mentioned in section 3.4, we will restrict the rules for the operators, that potentially yield divergent behaviours, to the case where the relevant operands do not diverge. Once again it must be understood that if the refusal set of one of the operands is undefined, then the refusal set of the compound expression is also undefined, provided that the latter is defined as a function of the operands refusal sets. Let $\alpha(E)$ denote the alphabet of an environment E, which interacts with any of the indicated processes. Clearly, **stop** refuses everything

R 4.1 $refusals(\textbf{stop}) = \{X | X \subseteq \alpha(E)\}$

The process **exit** refuses every set that does not contain the successful termination action δ

R 4.2 $refusals(\textbf{exit}) = \{X | X \subseteq (\alpha(E) - \{\delta\})\}$

The behaviour expression $(a;\ B)$ refuses every set that does not contain the observable action a

R 4.3 $refusals(a;\ B) = \{X | X \subseteq (\alpha(E) - \{a\})\}$

It is clear from the definition of refusals that the refusal set of (i; B) is the same as the refusal set of B, provided that B does not diverge

R 4.4 $refusals(\mathrm{i};\ B) = refusals(B)$ *if B does not diverge.*

Hiding a sequence of actions A in an expression B results in the refusal of a set X only when B can refuse the whole set $X \cup \{A\}$; X together with all the hidden actions

R 4.5 $refusals(\text{hide } A \text{ in } B)$
$= \{X | (X \cup \{A\}) \in refusals(B/t) \text{ where } t \in traces(B) \cap \{A\}^*\}$
$if \forall t \in traces(B).\neg diverges(B/t, \{A\})$

For example, if

$B =$ hide a in (a; stop [] b; stop)

then

$refusals(B) = \{\{b\}\}$

where in this context $\alpha(E) = \alpha(B) = \{b\}$. Note that a is hidden in B, and so it is not in its alphabet.

The rule for process instantiation is clearly simple

R 4.6 *If* "process $P\ [a'_1,\ldots,a'_n] := B_p$ endproc" *is a process definition and* F *is a renaming given by* $a_1/a'_1,\ldots,a_n/a'_n$
then $refusals(P\ [a_1,\ldots,a_n]) = \{F(X) | X \in refusals(B_p)\}$

However, if a process P is recursive and not g.w.d. then its refusal set is undefined.

If $\exists B'.B \xrightarrow{i} B'$, we say that B is *i-prefixed*, and write $i\text{-}prefixed(B)$. The rule for [] is complicated by the need to handle all cases of nondeterminism introduced by the composition of i-prefixed expressions with this operator. If both operands are i-prefixed, the refusal of every one of them is also a possible refusal of the compound expression B. If only one operand is i-prefixed, then B would refuse whatever this operand can refuse. When neither operand is i-prefixed, if both of them can refuse a set of actions X, so can B. However, if one of them cannot refuse X, then neither can B. Therefore

R 4.7 *If* $i\text{-}prefixed(B_j)$ $(j = 1,2)$, *and* $\neg i\text{-}prefixed(B_k)$ $(k = 3,4)$
then a) $refusals(B_1[]B_2) = refusals(B_1) \cup refusals(B_2)$
b) $refusals(B_2[]B_3) = refusals(B_3[]B_2) = refusals(B_2)$
c) $refusals(B_3[]B_4) = refusals(B_3) \cap refusals(B_4)$

For example, let $\alpha(E) = \{a,b\}$ and
$B_1 = (a;\ \mathbf{stop})$ and $B_2 = (b;\ \mathbf{stop})$
then
$refusals(\mathrm{i};\ B_1\ []\ \mathrm{i};\ B_2) = refusals(B_1) \cup refusals(B_2) = \{\{a\},\{b\}\}$
$refusals(\mathrm{i};\ B_1\ []\ B_2) = refusals(B_1) = \{\{b\}\}$
$refusals(B_1[]B_2) = refusals(B_1) \cap refusals(B_2) = \{\}$

It seems appropriate at this point, to precisely define i-prefixed expressions based on their structures. Obviously, the processes stop, exit, and any expression prefixed by an observable action are not i-prefixed. However,

1) $i\text{-}prefixed(\mathrm{i};\ B) \equiv true$

2) $i\text{-}prefixed(\mathrm{hide}\ A\ \mathrm{in}\ B) \equiv (initials(B) \cap \{A\} \neq \{\}) \vee i\text{-}prefixed(B)$

3) $i\text{-}prefixed(B_1 \gg B_2) \equiv i\text{-}prefixed(B_1) \vee \delta \in initials(B_1)$

4) $i\text{-}pref(B_1 * B_2) \equiv i\text{-}prefixed(B_1) \vee i\text{-}prefixed(B_2)$

where $*$ stands for [], $|[A]|$, and [>. Note that a process instantiation is i-prefixed if the behaviour expression defining that process is i-prefixed. If P is recursive and not g.w.d. then $i\text{-}prefixed(P)$ is undefined. $i\text{-}prefixed(B)$ is also undefined whenever it is undefined for any of the component expressions of B.

The parallel composition expression $(B_1|[A]|B_2)$ can refuse all actions refused by both components, B_1 and B_2, together with the actions of A_δ that either component can refuse

R 4.8 $refusals(B_1|[A]|B_2)$
$$= \{(X \cap Y) \cup ((X \cup Y) \cap \{A\}_\delta) | X \in refusals(B_1) \wedge Y \in refusals(B_2)\}$$

When the sequence of actions A is empty, this composition amounts to interleaving (|||). Therefore

R 4.9 $refusals(B_1|||B_2)$
$$= \{(X \cap Y) \cup ((X \cup Y) \cap \{\delta\}) | X \in refusals(B_1) \wedge Y \in refusals(B_2)\}$$

When A includes all actions that are in the alphabets of both components, this composition amounts to full synchronization (||). Therefore

R 4.10 $refusals(B_1||B_2) = \{X \cup Y | X \in refusals(B_1) \wedge Y \in refusals(B_2)\}$

If B_1 can refuse the set $X \cup \{\delta\}$, that is it can refuse X and cannot immediately terminate successfully, then the combination $(B_1 \gg B_2)$ can refuse X, since the latter is a refusal of B_1. But if B_1 can immediately terminate successfully, then any refusal of B_2 is also a refusal of $(B_1 \gg B_2)$, since termination enables B_2, and the passing of control is seen as an internal action i.

R 4.11 $refusals(B_1 \gg B_2) = \{X | X \cup \{\delta\} \in refusals(B_1)\}$
$\cup \{X | \delta \in initials(B_1) \wedge X \in refusals(B_2)\}$
if B_2 does not diverge.

If the rule appears to be awkwardly stated, consider the example

(i; a; **stop** [] **exit**) $\gg$ b; **stop**

the refusal set of which is

$\{\{b\}\} \cup \{\{a, \delta\}\} = \{\{b\}, \{a, \delta\}\}$

Also consider

(i; a; **stop** [] **exit**) $\gg$ **exit**

which can only refuse $\{a, b\}$. Note therefore that

$refusals((a;$ **stop** [] **exit**$) \gg b;$ **stop**$)$
$= refusals(a;$ **stop** [] i; $b;$ **stop**$) = \{\{a, \delta\}\}$

as expected. In all previous examples let $\alpha(E) = \{a, b, \delta\}$.

The rule for the disabling operator depends on whether its second argument is i-prefixed or not. If B_2 is not i-prefixed, then $(B_1[> B_2)$ can refuse a set of actions X whenever B_1 and B_2 can. However, if B_2 is i-prefixed, $(B_1[> B_2)$ can only refuse the refusals of B_2, no matter what B_1 can refuse.

This is so, because B_1 may or may not be interrupted by an initial of B_2, until B_1 terminates successfully. More formally

R 4.12 $refusals(B_1[> B_2)$
$= refusals(B_1) \cap refusals(B_2)$ *if* $\neg i\text{-}prefixed(B_2)$
$= refusals(B_2)$ *otherwise.*

For example, let $\alpha(E) = \{a, b\}$
$B_1 = (a;\ \mathbf{stop})$ and $B_2 = (b;\ \mathbf{stop})$
then
$refusals(B_1[> B_2) = refusals(B_1) \cap refusals(B_2) = \{\}$
$refusals(B_1[>\ \mathrm{i};\ B_2) = refusals(B_2) = \{\{a\}\}$

4.4 Trace specifications revisited

We can conclude from our previous discussion on refusals, that refusal sets represent an important (indirectly) observable aspect of the behaviour of a process. Therefore, we can enrich our trace specifications to express the behaviour requirements of processes by describing the characteristics of their refusal sets as well as their traces. This description is then a predicate containing two free variables t and r, that stand for an arbitrary trace and refusal set, respectively, of the process being defined, together with, possibly, some other variables. We revise the formal meaning of the satisfaction relation
$P \models S(t, r)$
to become
$\forall t, r.t \in traces(P) \wedge r \in refusals(P/t) \Rightarrow S(t, r)$

For example, the one slot simplex buffer SB (Example 3.3 on page 36) satisfies the following behaviour requirement

$$SB[input, output] \models (0 \leq t \downarrow input - t \downarrow output \leq 1) \wedge$$
$$(\text{if } t \downarrow input = t \downarrow output$$
$$\text{then } input \notin r$$
$$\text{else } output \notin r)$$

The behaviour requirement of the one slot unreliable simplex buffer SB' (Example 3.4 on page 36) can be precisely defined

$$SB'[input, output] \models (t \downarrow output \leq t \downarrow input) \wedge$$
$$\neg(\langle output, output \rangle \text{ in } t) \wedge (input \notin r)$$

Moreover the behaviour requirement of the unbounded buffer UB (Example 3.10 on page 63) can be expressed as

$$UB[input, output] \models (t \downarrow output \leq t \downarrow input) \wedge$$
$$(\text{if } t \downarrow input = t \downarrow output$$
$$\text{then } input \notin r$$
$$\text{else } output \notin r)$$

The introduction of refusals in the trace specifications of processes provides a means to distinguish the specification of the unbounded reliable buffer UB from one that describes the behaviour of an unreliable one (see page 64), since the latter can be expressed as

$$(t \downarrow output \leq t \downarrow input) \wedge input \notin r$$

The difference is that UB can never refuse to output what was input, while the unreliable buffer can.

We shall now revise our proof rules. Once again, a specification will be written as S, $S(t)$, or $S(t, r)$, according to the context. Sometimes, we will use a single underline character (i.e. $S(t, \_)$) to denote a variable that is not relevant to the context.

The process stop refuses everything, including every subset of the alphabet of the environment in which this process is placed. Therefore, every refusal of stop is a subset of this alphabet. This is consistent with the fact that every process refuses every subset of actions that is not in its alphabet, and the alphabet of stop is empty.

Sr 4.1 stop $\models t = \langle\rangle \wedge r \subseteq \alpha(E)$

The process exit refuses every set not containing the successful termination action δ, after which it behaves like stop. Therefore, its rule need to be strengthened

Sr 4.2 exit $\models ((t = \langle\rangle \wedge \delta \notin r) \vee (t = \langle\delta\rangle \wedge r \subseteq \alpha(E)))$

Similarly, the rule for the expression $(a;\ B)$ has to mention that initially, when $t = \langle\rangle$, the (observable) action a cannot be refused

Sr 4.3 *If* $B \models S(t)$
then $(a;\ B) \models ((t = \langle\rangle \wedge a \notin r) \vee (t = \langle a \cdot t'\rangle \wedge S(t')))$

In all rules below, it is to be understood that the processes involved do not diverge.

The rule for the expression (i; B) need not to be changed, since every refusal of (i; B) is also a refusal of B. Therefore

Sr 4.4 *If* $B \models S$
then (i; B) $\models S$

The rule for hiding needs to be changed

Sr 4.5 *If* $B \models (\neg diverge(\{B\},\{A\}) \wedge S(t,r))$
then (hide A in B) $\models (\exists u.t = u\lfloor(\alpha(B) - \{A\}) \wedge S(u, r \cup \{A\}))$

That is so, because a behaviour expression involving hiding can only interact with its environment when it reaches a state in which it cannot engage in any further hidden actions.

The rule for process instantiation needs a simple adaptation

Sr 4.6 *Let* "process $P\ [a'_1,\ldots,a'_n] := B_p$ endproc" *be a process definition and* F *be a renaming given by* $a_1/a'_1,\ldots,a_n/a'_n$
If $B_p \models S(t,r)$
then $P\ [a_1,\ldots,a_n] \models (\exists u \in traces(B_p).t = u[F]$
$\wedge \exists X \in refusals(B_p/u).r = F(X) \wedge S(u,X))$

We proceed by revising the rule for [] to extend it to refusals. If both operands are i-prefixed, the previous rule is still valid, since in this case every possible observation of the behaviour of $(B_1[]B_2)$ will be a possible observation for B_1 or B_2. However, if only B_1 is i-prefixed, initially, when $t = \langle\rangle$, $(B_1[]B_2)$ cannot refuse a set of actions X unless it is refused by B_1. Furthermore, if neither B_1 nor B_2 is i-prefixed, $(B_1[]B_2)$ cannot refuse X unless it is refused by both B_1 and B_2, whenever $t = \langle\rangle$. Therefore

Sr 4.7 *If* $i\text{-}prefixed(B_j)$ $(j = 1,2)$, *and* $\neg i\text{-}prefixed(B_k)$ $(k = 3,4)$
and $B_j \models R_j$ *and* $B_k \models S_k$
then a) $(B_1[]B_2) \models (R_1 \vee R_2)$

$b)\ (B_2[]B_3) \models (if\ t = \langle\rangle\ then\ R_2\ else\ (R_2 \vee S_3))$
$c)\ (B_3[]B_4) \models (if\ t = \langle\rangle\ then\ (S_3 \wedge S_4)\ else\ (S_3 \vee S_4))$

Note that the right-hand-side of rule *b*) applies also to $(B_3[]B_2)$.

Next, we strengthen the rule for parallel composition

Sr 4.8 *If* $B_1 \models R(t,r)$
and $B_2 \models S(t,r)$
then $(B_1|[A]|B_2) \models (\exists u,v,X,Y.t \in u|\{A\}|v \wedge$
$r = (X \cap Y) \cup ((X \cup Y) \cap A_\delta) \wedge R(u,X) \wedge S(v,Y))$

The rule for sequential composition is slightly more complicated than it was

Sr 4.9 *If* $B_1 \models R(t,r)$
and $B_2 \models S(t,r)$
then $(B_1 \gg B_2) \models (\exists u,v.t = u \gg v \wedge (if\ \langle\delta\rangle\ in\ u\ then\ (R(u,\_) \wedge$
$S(v,r))\ else\ (R(u,r \cup \{\delta\}) \wedge S(v,\_))))$

The rule for [> needs a similar adaptation, except that we need to distinguish two cases. If its second operand is not i-prefixed, then every refusal of the compound expression is either a possible refusal of both operands or a possible refusal of the second operand if the first one was already interrupted. Otherwise, every refusal of the compound expression is a refusal of the second operand before or after disruption. Therefore

Sr 4.10 *If* $B_1 \models R(t,r)$
and $B_2 \models S(t,r)$

$$\begin{aligned} \textit{then } (B_1[> B_2) \models (\exists u, v.t \in u[> v \wedge (\textit{if } v = \langle\rangle \textit{ then } R(u,r) \\ \textit{else } R(u,\_)) \wedge S(v,r)) \end{aligned}$$

$$\begin{aligned} &\textit{if } \neg \textit{i-prefixed}(B_2) \\ &\models (\exists u, v.t \in u[> v \wedge R(u,\_) \wedge S(v,r)) \\ &\textit{otherwise.} \end{aligned}$$

The rules for / and ↑ need not to be changed, since they don't relate to refusal sets.

Example 4.1 Consider the simple example that describes pregnancy given by

$$B = B_1 [] B_2$$

where

$$B_1 = \textit{pregnant}; \textit{boy}; \textbf{stop}$$

$$B_2 = \textit{pregnant}; \textit{girl}; \textbf{stop}$$

Clearly by Sr 4.3

$$B_1 \models t = \langle \textit{pregnant} \rangle \Rightarrow \textit{boy} \notin r$$

$$B_2 \models t = \langle \textit{pregnant} \rangle \Rightarrow \textit{girl} \notin r$$

Therefore, by Sr 4.7-c)

$$B \models t = \langle \textit{pregnant} \rangle \Rightarrow (\textit{boy} \notin r \vee \textit{girl} \notin r)$$

which describes exactly the nondeterministic choice made before pregnancy. Note that the behaviour B' given by

$$B' = \textit{pregnant}; (\textbf{i}; \textit{boy}; \textbf{stop} [] \textbf{i}; \textit{girl}; \textbf{stop})$$

satisfies the same behaviour requirement as B, even though B and B' are distinguishable by Milner's observation equivalence ($\approx$, see page 39). □

4.5 Failures

In this section we introduce the notion of *failures* of a process P [BHR84], which takes into account what P may refuse after engaging in an arbitrary trace of its behaviour. In other words the concept of failure makes it possible to combine the information about traces and about refusals. This permits a clear distinction between processes, by observing their behaviour in a finite environment, which cannot be distinguished by experiments. We say that (t, X) is a failure of P if and only if t is a trace of P, and after engaging in the actions of t, P can refuse the finite set of actions X. The set of all failures of P is denoted by $failures(P)$, and defined as follows

Definition 4.2 *Let B be a behaviour expression. We define*

$$failures(B) = \{(t, X) | t \in traces(B) \wedge X \in refusals(B/t)\}$$

From this definition it follows that the set $F = failures(B)$ satisfy the properties

P 4.4 $(\langle\rangle, \{\}) \in F$

P 4.5 $((t, X) \in F \wedge Y \subseteq X) \Rightarrow (t, Y) \in F$

P 4.6 $((t, X) \in F \wedge (t \frown \langle a \rangle, \{\}) \notin F) \Rightarrow (t, X \cup \{a\}) \in F$

Clearly the failures of a process provide more information about the behaviour of that process than its traces or its refusals. This is justified by the fact that traces and refusals can be defined in terms of failures

$$traces(B) = \{t | (t, \{\}) \in failures(B)\}$$

since $\forall t.\{\} \in refusals(B/t)$, and

$$refusals(B) = \{X | (\langle\rangle, X) \in failures(B)\}$$

It is not also surprising that we can define the set of initials in terms of failures

$$initials(B) = \{a | (\langle a \rangle, \{\}) \in failures(B)\}$$

We can now reformulate the property of deterministic processes in terms of failures. B is deterministic if and only if for all traces, no initials can be refused, i.e.

$$(t, X) \in failures(B) \equiv X \cap initials(B/t) = \{\}$$

The formulation of a sufficient condition for proof of absence of deadlock, in a process B, is now possible

$$nodeadlock(B) \equiv (t, \alpha(B)) \notin failures(B)$$

which means that B cannot engage into a trace t after which it refuses every action in its alphabet. Unfortunately, this is also a sufficient condition for non-divergence.

The failures of a process B_1 are the only externally observable aspects of its behaviour. Thus a process B_2 that fails only in circumstances in which B_1 can fail, is regarded as a *reduction* of the behaviour of B_2. This approach is widely used in reasoning about an implementation and its specification [BSS87]. One may want to establish that an implementation is an acceptable reduction of its specification, rather than proving the equivalence of the two. This leads to the definition of the reduction relation.

Definition 4.3 *Let B_1 and B_2 be two behaviour expressions. We have*

$$B_2 \text{ red } B_1 \equiv failures(B_2) \subseteq failures(B_1)$$

In words, "B_2 **red** B_1" means that B_2 can only engage into traces that are possible for B_1, and only refuse sets of actions that can be refused by B_1.

Example 4.2 Let

$B_1 = (a;\ b;\ \mathbf{stop}\ []\ a;\ c;\ \mathbf{stop})$

$B_2 = a;\ (b;\ \mathbf{stop}\ []\ c;\ \mathbf{stop})$

$A = \{a, b, c\}$

Then B_2 **red** B_1 since

$$failures(B_1) = \{(\langle\rangle, \{b,c\}), (\langle a\rangle, \{a,c\}), (\langle a\rangle, \{a,b\}), (\langle a,b\rangle, A), (\langle a,c\rangle, A)\}$$

$$failures(B_2) = \{(\langle\rangle, \{b,c\}), (\langle a\rangle, \{a\}), (\langle a,b\rangle, A), (\langle a,c\rangle, A)\}$$

and $failures(B_2) \subseteq failures(B_1)$ □

Note that we do not need to write the complete failure set, since if $(t, X \cup Y) \in failures(B)$, then $(t, X) \in failures(B)$, and in this case (t, X) need not to be mentioned (it is implicit). Also note from the example above that B_2 is deterministic and B_1 is nondeterministic, in which case we say that B_2 is more deterministic than B_1. Therefore, **red** corresponds exactly to the nondeterministic order on processes. This "implementation relation" was introduced for CSP in [BHR84], and for CCS in [DH84].

Clearly, **red** is a partial ordering. However, contrary to CSP, it is not monotonic for all LOTOS operators. For example, [] is not monotonic for this ordering, since

($\mathbf{i}$; B_1) **red** B_1

always holds, but it is not in general true that

($\mathbf{i}$; B_1 [] B_2) **red** (B_1 [] B_2)

As a counterexample let

$B_1 = (a;\ \mathbf{stop})$ and $B_2 = (b;\ \mathbf{stop})$

Therefore, the general fixed point theory, which implies in the case of a complete partial order (for which all operators are continuous) that any fixed point equation has a unique minimal and maximal solution, cannot be applied in LOTOS by means of red. This is also the case for CCS, because no complete partial order, that is continuous for all its operators, can be found. The implications of this fact is that failures of recursive behaviour expressions cannot be computed, at least by fixed point induction.

The reduction relation induces equivalences between behaviours of processes.

Definition 4.4 *Let B_1 and B_2 be two behaviour expressions. We define*

$$B_1 \sim B_2 \equiv B_1 \text{ red } B_2 \wedge B_2 \text{ red } B_1$$

This equivalence ($\sim$) is referred to as *failure equivalence* in [BHR84], since it doesn't distinguish two processes that fail in exactly the same circumstances. It follows that

P 4.7 $B_1 \sim B_2 \equiv failures(B_1) = failures(B_2)$

Following [DH84], if $(B_1 \sim B_2)$, we say that B_1 is *testing equivalent* with B_2. The name testing equivalence is justified by the fact that this relation identifies exactly those processes that cannot be distinguished by testing. Moreover, it was used in [Bri88a] as a basis to support practical testing of processes for conformance to their specification (in the case of strongly convergent processes). In this context, it is called *testing equivalence*. The latter paper shows how tests can be derived from a specification by constructing

its failure tree, an alternative representation of failure sets, which is then transformed to give the failure tree of the corresponding tester.

It was noticed in [Bro83] that testing equivalence implies trace equivalence ($\approx_1$, see Definition 3.6 on page 42) and is implied by Milner's second equivalence relation ($\approx_2$, see Definition 3.3 on page 39). Recall that

1) $B_1 \approx_1 B_2 \equiv traces(B_1) = traces(B_2)$

2) $B_1 \approx_2 B_2 \equiv \forall t, T.(\exists B_1' \in B_1/t \wedge traces(B_1') = T$
$\equiv \exists B_2' \in B_2/t \wedge traces(B_2') = T)$

Clearly, if

$B_1 = a$; stop and $B_2 = (a$; stop [] i; stop)

then $B_1 \approx_1 B_2$ but $B_1 \not\sim B_2$. However B_1 red B_2. Moreover, if

$B_1 = (a; b$; stop [] $a; c$; stop)

$B_2 = (\mathrm{i}; a; b$; stop [] i; $a; c$; stop)

then $B_1 \not\approx_2 B_2$ but $B_1 \sim B_2$.

Therefore, testing equivalence is a weaker relation than observation equivalence (Definition 3.3 on page 39), since it makes more identifications between processes.

We end this section by listing a set of rules and axioms that define the failures of compound processes in terms of the failures of their components. They are obtained by combining the rules that define traces with the rules that define refusals. Obviously, if the trace set or refusal set of a process is undefined, so is its failure set. The result is a failures semantics for LOTOS, restricted to non-divergent behaviours, since failure sets describe all observable aspects of the behaviour of a process, and the following axioms and rules provide a means for interpreting LOTOS text as a failure set. Let E denote the environment which interacts with any of the indicated processes.

F 4.1 $failures(\mathbf{stop}) = \{(\langle\rangle, X) | X \subseteq \alpha(E)\}$

F 4.2 $failures(\mathbf{exit}) = \{(\langle\rangle, X) | X \subseteq (\alpha(E) - \{\delta\})\} \cup \{(\langle\delta\rangle, X) | X \subseteq \alpha(E)\}$

F 4.3 $failures(a;\ B) = \{(\langle\rangle, X) | X \subseteq (\alpha(E) - \{a\})\}$
$\cup \{(\langle a \cdot t\rangle, X) | (t, X) \in failures(B)\}$

F 4.4 $failures(\mathbf{i};\ B) = failures(B)$ *if B does not diverge.*

F 4.5 $failures(\mathbf{hide}\ A\ \mathbf{in}\ B)$
$= \{(t \lfloor (\alpha(B) - \{A\}), X) | (t, X \cup \{A\}) \in failures(B)\}$
if $\forall t \in traces(B).\neg diverges(B/t, \{A\})$

F 4.6 *If* "**process** $P\ [a'_1, \ldots, a'_n] := B_p$ **endproc**" *is a process definition and F is a renaming given by* $a_1/a'_1, \ldots, a_n/a'_n$
then $failures(P\ [a_1, \ldots, a_n]) = \{(t[F], F(X)) | (t, X) \in failures(B_p)\}$

F 4.7 *If* $i\text{-}prefixed(B_j)$ $(j = 1, 2)$, *and* $\neg i\text{-}prefixed(B_k)$ $(k = 3, 4)$
then a) $failures(B_1 [] B_2) = failures(B_1) \cup failures(B_2)$
b) $failures(B_2 [] B_3) = failures(B_3 [] B_2)$
$= \{(t, X) | (t, X) \in failures(B_2) \vee$
$(t \neq \langle\rangle \wedge (t, X) \in failures(B_2) \cup failures(B_3))\}$
c) $failures(B_3 [] B_4)$
$= \{(t, X) | (t, X) \in (failures(B_3) \cap failures(B_4)) \vee$
$(t \neq \langle\rangle \wedge (t, X) \in failures(B_3) \cup failures(B_4))\}$

F 4.8 $failures(B_1 |[A]| B_2)$
$= \{(t, (X \cap Y) \cup ((X \cup Y) \cap \{A\}_\delta)) | \exists u, v. t \in u |\{A\}| v$

$$\wedge(u,X) \in failures(B_1) \wedge (v,Y) \in failures(B_2)\}$$

F 4.9 $failures(B_1|||B_2)$
$$= \{(t,(X \cap Y) \cup ((X \cup Y) \cap \{\delta\}))|\exists u,v.t \in u|\{\}|v$$
$$\wedge(u,X) \in failures(B_1) \wedge (v,Y) \in failures(B_2)\}$$

F 4.10 $failures(B_1||B_2)$
$$= \{(t,X \cup Y)|\exists u,v.t \in u|\alpha(B_1) \cup \alpha(B_2)|v$$
$$\wedge(u,X) \in failures(B_1) \wedge (v,Y) \in failures(B_2)\}$$

F 4.11 $failures(B_1 \gg B_2) = \{(t,X)|(t,X \cup \{\delta\}) \in failures(B_1)\}$
$$\cup\{(u \frown v,X)|u \frown \langle\delta\rangle \in traces(B_1) \wedge (v,X) \in failures(B_2)\}$$

if B_2 does not diverge.

F 4.12 $failures(B_1[> B_2) = \{(t,X)|\exists u,v.t \in u[> v \wedge$
$$(if\ v = \langle\rangle\ then\ (u,X) \in failures(B_1)\ else\ u \in traces(B_1)) \wedge (v,X) \in failures(B_2)\}$$
if $\neg i\text{-}prefixed(B_2)$
$$= \{(t,X)|\exists u,v.t \in u[> v \wedge u \in traces(B_1) \wedge (v,X) \in failures(B_2)\}$$
otherwise.

Example 4.3 Let's consider the behaviour

$B = B_1[]B_2$

where

$B_1 = (a;$ stop) and $B_2 = (i;$ stop)

and $\alpha(E) = \{a\}$. Then by F 4.7-*b*)

$$failures(B) = \{(t, X) | (t, X) \in failures(B_2) \vee (t \neq \langle\rangle \wedge (t, X) \in failures(B_2) \cup failures(B_1))\}$$

Now by F 4.1, F 4.3, and F 4.4

$$failures(B_2) = failures(\mathbf{stop}) = \{(\langle\rangle, \{a\})\}$$

$$failures(B_1) = \{(\langle\rangle, \{\}), (\langle a\rangle, \{a\})\}$$

Therefore

$$failures(B) = \{(\langle\rangle, \{a\}), (\langle a\rangle, \{a\})\}$$ □

Chapter 5

Conclusions

We began by attempting to use traces as a means to characterize LOTOS processes. To this end, we adapted the existent trace theory and defined new composition functions on traces. Possibly the most interesting one is the "merging", which enjoys many pleasing properties. This has led to a trace set semantics for LOTOS presented in denotational style, the trace sets of compound processes being built up from the traces of their components. These semantics are suitable for reasoning about communication sequences but unable to cope with nondeterminism. Therefore, following the lead of TCSP [BHR84], we adapted the failure model to serve as a mathematical model for LOTOS. This is a direct extension of the trace model given by associating to every process a so-called refusal set. The result is an alternative semantics for LOTOS, basically denotational but having strong connections with operational behaviours [Old86]. The LOTOS behaviour expressions, however, are interpreted as failure sets. This is different from the direct

definition of [ISO88]-operators in that the latter are expressed in operational terms by inference rules. One advantage of our approach is a better link between LOTOS and CSP; it also provides valuable information for defining a mapping from LOTOS to CSP. In [DH87] an alternative version of CCS, called TCCS (Testing CCS), which does not use internal actions to model nondeterminism, is proposed. Instead, the choice operator of CCS is replaced by the internal and external choice operators of TCSP. Therefore, the purely nondeterministic part of TCCS coincides with that of TCSP, and it was possible to define a mapping from CCS to TCCS.

Another advantage of our approach is that it provides a useful mathematical framework for the analysis of LOTOS specifications and for developing logical systems to prove their properties. This is illustrated by the proof rules that permit a proof of correctness of a compound process to be constructed from a proof of the correctness of its parts. This differs from the form of modal logic introduced in [HM85] where processes are characterized by the properties they enjoy, because the calculus governing the associated satisfaction relation is based on the structure of the properties rather than on the structure of the processes. The definition of our proof rules is based on Hoare's sat relation [Hoa85].

Two kinds of properties, *safety* and *liveness*, are important for parallel systems. The trace model is well suited for the treatment of safety properties, since it provides possible positive information about what might happen. The treatment of liveness properties requires a model that supports reasoning about what must happen by giving possible negative information, that is refusals or failures. It would be interesting to show how the proof techniques of this thesis can be applied to verify safety and liveness properties of systems

described in LOTOS.

In [Bri88a] a theory of test derivation on the basis of testing equivalences is formulated. It uses failures trees, an alternative representation of failure sets, to model processes. This theory is applicable to formal description methods that allow a semantic interpretation of specifications in terms of labelled transition systems with internal actions. It should be possible to formulate a method for the derivation of testers from LOTOS specifications based on the syntactical approach explored in [Wez88, Wez89], using the framework presented in this thesis. More specifically, testers of compound processes can be constructed from the testers of their components by applying simple transformations on their failure sets from which we derive the expressions they model. Recently, an alternative testing theory for LOTOS has been proposed [Lan89]. It gives rise to several new implementation relations, and it defines a testing language that has a construct for the detection of deadlock. It would be interesting to see how our work can be extended to handle such a construct and to discriminate processes that cannot be discriminated by testing equivalence, as is the case in this testing theory.

The model presented in this thesis is very well suited to reasoning about the problems associated with deadlock. However, it doesn't answer the question related to the interpretation of divergent behaviours by means of their traces and failures. Such behaviours have been dealt with in different ways. In [BHR84, BR85] divergence is equated with a special process CHAOS that makes any distinction of the subsequent process behaviour impossible. The set of all traces after which a process behaves chaotically is called *divergences*. In CSP the divergences of compound processes are determined by the divergences and traces of their components. The adaptation of this method to

LOTOS is not straightforward, since divergence in LOTOS does not amount to CHAOS, and is modelled by an infinite sequence of i-actions (a concept that doesn't exist in CSP). Another relevant treatment of divergence is due to [BKO86], where a process semantics that combines fair abstraction from internal process activity with the failure semantics is introduced. Divergence is modelled by a special process pronounced "delay", an idea that was inspired by Milner's delay operator in SCCS [Mil83]. This differs from the original failure semantics [BHR84, BR85] in that it continues to record failure pairs even after a divergence is encountered. It is not yet clear which methods of modelling divergence are likely to be more successful. Therefore, for this reason and for the sake of clarity of exposition, we decided to defer the treatment of divergences to a later stage of research.

It seems likely that, in many cases, in order to cope with value expressions in LOTOS specifications, we need to adapt the CSP notion [Hoa85] of input messages along a communication channel. In this context, a communication is seen as an event that is described by a pair $a.v$, where a is the name of the channel on which the communication takes place and v is the value of the message passed. However, this remains to be elaborated.

An alternative formulation of our work could have been based on the model of acceptance semantics introduced in [DH84, Hen85, Hen88]. This is quite similar to the failure semantics except that refusal sets are replaced with acceptance sets. Both models share their most important characteristic, namely a sound mathematical basis for reasoning about specifications, verifications, testing, and implementations. However, much more (exciting) research is required to narrow the gap between theory and practice.

Appendix A

Glossary of Symbols

A.1 Abbreviations

CCS	Calculus of Communicating Systems
CSP	Communicating Sequential Processes
FDT	Formal Description Techniques
g.w.d.	guardedly well-defined
ISO	International Organization for Standardization
JACM	Journal of the Association of Computing Machinery
LNCS	Lecture Notes in Computer Science
LOTCAL	A basic Calculus for LOTOS processes
LOTOS	Language of Temporal Ordering Specifications
OSI	Open Systems Interconnection
SCCS	Synchronous CCS

TCS	Theoretical Computer Science
TCCS	Testing CCS
TCSP	Theoretical CSP
w.r.t.	with respect to

A.2 Frequently used symbols

$=$	equals
$\neq$	is distinct from
$\{\}$	the empty set
$\{x \mid P(x)\}$	the set of all x such that $P(x)$
$F : A \longrightarrow B$	F is a function mapping members of A to members of B
$F(x)$	that member of B to which F maps x (in A)
$F(X)$	the image of a set X under F
$\in$	is a member of
$\notin$	is not a member of
$\subseteq$	set containment
$\cup$	set union
$\cap$	set intersection
$\wedge$	conjunction
$\vee$	disjunction
$\neg$	negation
$\Rightarrow$	implies
$\equiv$	if and only if
$\forall$	universal quantifier
$\exists$	existential quantifier

$\leq$	less or equals
$\geq$	greater or equals
$\square$	end of an example or proof

A.3 Traces

$\langle\rangle$	the empty trace
$\langle a \cdot t \rangle$	the trace with a followed by the trace t
A^*	set of all traces with elements in A
A^*_δ	set of all traces in $(A \cup \{\delta\})^*$ with δ at the end
$\lfloor$	projection
$\|t\|$	the length of the trace t
$t \downarrow A$	the count of symbols of A in t
$t \downarrow a$	the count of symbol a in t
$\frown$	concatenation
t^n	n^{th} trace power of t
t^*	0 or more copies of t concatenated to each other
t^+	1 or more copies of t concatenated to each other
$\preceq$	prefix relation
$\prec$	proper prefix relation
$pref(T)$	prefix closure of T
in	contiguous containment
$\sqsubseteq$	non-contiguous containment
$\gg$	sequential composition over traces
$t[F]$	the result of applying the renaming function given by F to t
a/b	defines a renaming F such that $F(b) = a$

$\|A\|$	merging w.r.t. to the set A
[>	disruption over traces

A.4 Behaviours

stop	inaction
exit	successful termination
δ	successful termination action generated by exit
i	internal action
;	action prefix
hide A in B	hiding the actions of A in B
$P[\ldots]$	an instantiation of a process P
[]	choice
$\|[A]\|$	parallel composition
$\|\|\|$	interleaving
$\|\|$	parallel composition with full synchronization
$\gg$	sequential composition (also called enabling)
[>	disruption (also called disabling)
/	after (see page 38)
$\uparrow$	approximation (see page 51)
$\alpha(B)$	the alphabet of a behaviour B (see page 45)
$diverges(S, A)$	a behaviour in S diverges after hiding the actions in A (see page 45)
$i\text{-}prefixed(B)$	B can perform an i-action (see page 73)
$\xrightarrow{x}$	a transition involving a single action x
$\xrightarrow{s}$	a transition involving a sequence of actions denoted by s

$\Longrightarrow$	trace relation
$\approx_i$	i^{th} Milner's observation equivalence
$\approx$	Milner's observation equivalence
$\sim$	testing or failure equivalence
red	reduction relation
$\models$	satisfaction relation

Bibliography

[BBS7] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[BHR84] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential. *JACM*, 31(3):560–599, 1984.

[BKO86] J.A. Bergstra, J.W. Klop, and E.-R. Olderog. Failures without CHAOS: A new process semantics for fair abstraction. Technical Report CS-R8625, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1986.

[BRS5] S.D. Brookes and A.W. Roscoe. An improved failures model for communicating sequential processes. In S.D. Brookes, A.W. Roscoe, and G. Winskel, editors, *NSF-SERC Seminar on Concurrency*. LNCS volume 197, Springer-Verlag, 1985.

[Bri88a] E. Brinksma. A theory for the derivation of tests. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing, and Verification VIII*, pages 63–74. North-Holland, 1988.

[Bri88b] Ed Brinksma. *On the Design of Extended LOTOS*. PhD thesis, University of Twente, The Netherlands, 1988.

[Bro83] S.D. Brookes. On the relationship of CCS and CSP. In G. Goos and J. Hartmanis, editors, *Automata, Languages and Programming, 10th Colloquium.* LNCS volume 154, Springer-Verlag, 1983.

[BS87] T. Bolognesi and S.A. Smolka. Fundamental results for the verification of observational equivalence: a survey. In H. Rudin and C. West, editors, *Protocol Specification, Testing, and Verification VII.* North-Holland, 1987.

[BSS87] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS specifications, their implementations and their tests. In G.v. Bochmann and B. Sarikaya, editors, *Protocol Specification, Testing, and Verification VI*, pages 349–360. North-Holland, 1987.

[DH84] R. DeNicola and M. Hennessy. Testing equivalences for processes. *TCS*, 34:83–133, 1984.

[DH87] R. DeNicola and M. Hennesy. CCS without τ's. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development.* LNCS volume 249, Springer-Verlag, 1987.

[EM85] B. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications.* Sringer-Verlag, 1985.

[Gin66] Seymour Ginsburg. *The Mathematical Theory of Context-free Languages.* McGraw-Hill, 1966.

[Hen85] M. Hennessy. Acceptance trees. *JACM*, 32(4):896–928, 1985.

[Hen88] Matthew Hennessy. *Algebraic Theory of Processes.* Foundations of Computing Series. The MIT Press, 1988.

[HM85] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *JACM*, 32(1):137–161, 1985.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), 1978.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-

Hall International, 1985.

[ISO88] ISO, IS 8807. *Information processing systems - Open systems interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behaviour*, May 1988.

[Kou87] D.G. Kourie. The design and use of a prolog trace generator for CSP. *Software-Practice and Experience*, 17:423–438, 1987.

[Lan89] Rom Langerak. A testing theory for LOTOS using deadlock detection. In E. Brinksma, G. Scollo, and C. Vissers, editors, *Protocol Specification, Testing, and Verification IX*. North-Holland, 1989. To appear.

[LOBF88] L. Logrippo, A. Obaid, J.P. Briand, and M.C. Fehri. An interpreter for LOTOS, a specification language for distributed systems. *Software-Practice and Experience*, 18:365–385, 1988.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Sringer-Verlag, 1980.

[Mil83] Robin Milner. Calculi for synchrony and asynchrony. *TCS*, 25(3):267–310, 1983.

[Oba86] A. Obaid. A behavioural environment for the specification, design, and validation of protocols. A Ph.D. Candidacy paper, Electrical Engineering Department, University of Ottawa, November 1986.

[Old86] Ernst-Rudiger Olderog. TCSP: Theory of Communicating Sequential Processes. In *Petri Nets: Applications and Relationships to Other Models of Concurrency, Proceedings of an advanced course*. LNCS volume 255, Springer-Verlag, 1986.

[Par81] David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science, 5th GI-Conference*. LNCS volume 104, Springer-Verlag, 1981.

[Tur87] Ken Turner. The formal specification language LOTOS. A Course for Users, University of Stirling, August 1987.

[Tur88] Ken Turner. Constraint-oriented style in LOTOS. In *Proceedings of British Computing Society, Workshop on Formal Methods in Standards*, Dicot, April 1988.

[vdS85] Jan L.A. van de Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science*. Sringer-Verlag, 1985.

[vEVD89] P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors. *The Formal Description Technique LOTOS*. North-Holland, 1989.

[vG86] R.J. van Glabbeek. Notes on CCS and CSP. Technical Report CS-R8624, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1986.

[VSvS88] Chris A. Vissers, Giuseppe Scollo, and Marten van Sinderen. Architecture and specification style in formal descriptions of distributed systems. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing, and Verification VIII*. North-Holland, 1988.

[Wez88] Clazien D. Wezeman. The 'Co-op method', a method for compositional derivation of canonical testers. Master's thesis, University of Twente, The Netherlands, 1988.

[Wez89] Clazien D. Wezeman. The Co-op method for compositional derivation of conformance testers. In E. Brinksma, G. Scollo, and C. Vissers, editors, *Protocol Specification, Testing, and Verification IX*. North-Holland, 1989. To appear.