Partially Defined Lotos Specifications and their Refinement Relations

Timo Karvi

To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public criticism in Auditorium XV, University Main Building, on December 9th, 2000, at 10 a.m.
Contact information

Postal address:
   Department of Computer Science
   P.O. Box 26 (Teollisuuskatu 23)
   FIN-00014 University of Helsinki
   Finland

Email address: postmaster@cs.Helsinki.FI (Internet)

URL: http://www.cs.Helsinki.FI/

Telephone: +358 0 708 51

Telefax: +358 0 708 44441

ISSN 1238-8645
ISBN 951-45-9604-8
Helsinki 2000
Yliopistopaino
Partially Defined Lotos Specifications and their Refinement Relations

Timo Karvi

Department of Computer Science
P.O.Box 26, FIN-00014 University of Helsinki, Finland

Helsinki, November 2000, 157 pages
ISSN 1238-8645, ISBN 951-45-9604-8

Abstract

This work is a study of how large Lotos specifications can be developed stepwise. First a specification is written using partially defined processes. After this, the partially defined processes are refined either into complete processes, or into partially but more specified processes. After some steps all the processes are totally defined and the system is complete. If the service description of the system is given, it is possible to compare partially defined systems with the service at every step using appropriate refinement relations. These relations detect whether a partially defined system can or cannot be completed into a correct totally defined system. Thus, design errors can be detected automatically at an early stage.

In this work, appropriate refinement relations are developed in decorated trace semantics or failure semantics. Especially, we concentrate on two particular failure semantics, the so-called BKO- and CFFD-semantics. These semantics are sufficiently complicated examples to illustrate the technique needed in transferring concepts from the bisimulation to failure semantics. However, the definitions can be adapted easily into other trace-based semantics.

We show that our refinement relations behave compositionally with respect to the Lotos operators and show how the relations can be computed algorithmically. We extend our relations so that it is also possible to refine actions in full Lotos specifications. Action refinement is realized, however, in a restricted form.

Partially defined specifications have other applications as well. We show how partially defined processes can be used in distributed detection of deadlocks.
Computing Reviews (1998) Categories and Subject Descriptions:

F.3.1 [Specifying and verifying and reasoning about programs]:
    logics of programs, mechanical verification, specification techniques

F.1.2 [Modes of computation]: concurrency

D.2.4 [Program verification]

C.2.2 [Network protocols]: protocol verification

General Terms: specification, Lotos, partially defined processes

Additional Key Words and Phrases: weak bisimulation equivalence, divergences, BKO-equivalence, CFFD-equivalence, action refinement, stepwise refinement, deadlock
Acknowledgements

I would like to thank my supervisor, professor Martti Tienari, for introducing me to this area of research. His comments have helped me to correct definitions and proofs and his insistence on clarity has strongly influenced on the outlook of this work. Especially, I evaluate his patience during the long period this work has been done.

My colleague Roope Kaivola has done some sharp observations and stimulating conjectures, which have been helpful in deeper understanding of the subject. He also has helped to design CFFD-algorithms.

The Department of Computer Science has offered me a stimulated working environment without which this work would not have been started or finished. Thanks for that to the whole staff.
## Contents

1 **Introduction**  
1.1 Specifying systems in Lotos .......................... 1  
1.2 The new definitions and results .......................... 3  
1.3 About notations ............................................ 5  

2 **Basic Concepts** ............................................. 7  
2.1 Partially defined Lotos processes ......................... 7  
2.2 Concept of refinement ..................................... 10  
2.3 Lotos operators and transition systems ................... 12  
2.4 Equivalences, preorders and graphs ........................ 14  
2.5 Notations .................................................. 17  

3 **Weak Bisimilarity and Related Refinement Relations** 19  
3.1 Strong bisimilarity ....................................... 19  
3.2 Weak bisimilarity ......................................... 20  
3.3 Bisimulation refinement ................................... 21  
3.4 Partial bisimulation refinement ........................... 26  
3.5 Other bisimulation based refinement relations .......... 28  

4 **Applications** ............................................... 29  
4.1 Compositional verification, a conceptual example ........ 30  
4.2 Stepwise design of protocols ............................. 34  
4.3 Distributed detection of deadlocks ........................ 39  

5 **Divergence, BKO- and CFFD-Relations** ................. 49  
5.1 Divergence bisimilarity and refinement relations .......... 49  
5.2 BKO-equivalence and refinement relations ................ 55
CONTENTS

5.3 CFFD-equivalence and refinement relations .......................... 64
5.4 Remarks ........................................................................ 68

6 Comparisons ..................................................................... 69
6.1 Equivalences ................................................................... 69
6.2 Refinement relations ...................................................... 75

7 Algorithmic Aspects .......................................................... 83
7.1 Bisimulation-based equivalences ....................................... 83
7.2 Bisimulation refinements ................................................ 85
7.3 BKO- and CFFD-equivalence ........................................... 87
7.4 BKO- and CFFD-refinement ............................................ 89
7.5 Practical implementation of CFFD ..................................... 95

8 Abstract versus concrete notions of refinement ..................... 97
8.1 An Introduction .............................................................. 97
8.2 The conjecture in the bisimulation semantics ...................... 97
  8.2.1 The counterexamples ................................................ 97
  8.2.2 The modified conjecture in the bisimulation semantics .... 99
8.3 The conjecture in the CFFD-semantics ............................... 101
  8.3.1 The plan of the proof in CFFD-semantics ................. 101
  8.3.2 The proof of the structure conjecture in CFFD-semantics . 102

9 Action refinement in full Lotos ........................................... 111
9.1 About full Lotos ............................................................ 111
9.2 Partially defined actions ............................................... 113
9.3 Generalized BKO-refinement ......................................... 116
9.4 About modified bisimulation refinement .......................... 118
9.5 Implementation of BKO- and CFFD-refinement relations with action refinement ........................................... 119

10 Bounded Retransmission Protocol .................................... 121
10.1 Description of the protocol .......................................... 121
  10.1.1 Informal protocol specification ................................ 121
  10.1.2 Informal service specification .................................. 123
10.2 Lotos description of BRP service .................................. 123


CONTENTS

10.2.1 Data structures ........................................... 123
10.2.2 BRP service ................................................ 126
10.3 Design of BRP ................................................ 127
   10.3.1 The first step .......................................... 127
   10.3.2 The second step ....................................... 128
   10.3.3 The third step ......................................... 132
   10.3.4 The fourth step ....................................... 135
   10.3.5 The fifth step ......................................... 137
10.4 Derivation of BRP using action refinement ................. 143
   10.4.1 First step ............................................. 143
   10.4.2 Second step ........................................... 145
   10.4.3 Third step ............................................ 147

11 Conclusions and Further Research ............................. 149
Chapter 1

Introduction

1.1 Specifying systems in Lotos

Lotos is a specification language for systems consisting of communicating processes. Its semantics has been formally defined and this makes it possible to analyse and verify formally or algorithmically systems written in Lotos. Because Lotos is meant for the specification of real systems, it is fairly complex. Especially its data part, i.e. the description technique of the data sent between processes, is complicated and it is impossible to master it completely algorithmically. The new extended version of Lotos, E-Lotos, is still more complex and it resembles programming languages.

A usual way to analyse and verify a Lotos specification is to first generate a labeled transition system or a so called reachability graph. In the reachability graph we can locate deadlocks and other deficiencies such as livelocks, unspecified or unexecutable interactions and other kinds of dubious behaviour. Other types of errors are also possible. In fact, there are so many different potential error situations that it is impossible to enlist them all explicitly beforehand and then to check their presence or absence automatically one by one. So instead of a direct analysis, we need a different approach. We can either minimize the reachability graph and study this smaller graph, or we can compare it with a simpler and correct system. Both approaches demand a new concept, a process equivalence.

Process equivalences are reflexive, symmetric and transitive relations between processes. Using an equivalence we can identify similar processes with each other. Of course, an equivalence abstracts details away to a certain extent, but when chosen carefully, it can keep track of those properties we are interested in. For example, most of the known equivalences preserve deadlocks. Process equivalences facilitate the analysis of distributed systems, because usually every process has a minimal, often unique, counterpart with respect to an equivalence. This minimal process is equivalent with the original one and usually considerably smaller. So we can study the smaller counterpart instead of the original process.

Very often, in practice, the original system is even so large that we cannot build
its reachability graph and thus we can’t construct its minimal counterpart directly. But we are often able to minimize the components, combine them and minimize the possibly smaller system to achieve the minimal process. An essential condition in this method is that our equivalence ≈ is congruent with respect to the applied operators. If \( C \) is the set of possible processes and if we were using an operator \( f : C \times C \rightarrow C \) when combining smaller systems, the requirement that the relation \( \approx \) is a congruence means that for every \( P_1, P_2 \in C \), if \( P_1 \approx P'_1 \) and \( P_2 \approx P'_2 \), then \( f(P_1, P_2) \approx f(P'_1, P'_2) \). However, this approach does not guarantee success, because it might be that after minimizing the components, the resulting combined system is still too large. But if carefully planned and realized, the compositional construction brings bigger systems within reach of our analysis better than without intermediate minimizations.

A system usually interacts with other systems or users. It is not necessary for a system to know the internal structure of other systems, but it is essential to know the interfaces accurately. The term service is used to describe the interface a process gives to other systems via its interactions. Often the service can be described easily and, when formally specified as a process, it may already be minimal. Now it is possible to test if the original process, when restricted in some way, is equivalent to its service process. If this is the case, we can conclude that the system operates according to our requirements. Thus equivalence relations give us better prospects to verify the correctness of large systems. There are many equivalences and choosing an appropriate one for an application is not always easy. An equivalence should at the same time be intuitive and as weak as possible, but still preserve the properties a designer is interested in. Even if we had good equivalences something is still missing.

Large specifications are often constructed by a method known as stepwise refinement. First we have a coarse body, which is enlarged and refined by adding new parts and details to it. It is clear that different versions are not equivalent, because newer specifications have gone through essential modifications. Anyway, it would be useful to be able to compare different versions in a systematic way. Especially later versions should be compatible in one form or another with earlier ones. We need relations, which are looser than equivalences. These looser relations are not symmetric and sometimes even the transitivity property must be dropped. These relations are called preorders, partial orders, implementation relations or refinement relations depending on their properties and the context. We adopt the last term. If \( \sqsubseteq \) is a refinement relation, then we interpret the formula \( P \sqsubseteq R \) by saying that \( P \) is more abstract than \( R \) or that \( R \) is more specified than \( P \). In other words, \( R \) is a refinement of \( P \). A refinement relation is usually closely related to an equivalence. This follows from the fact that the intersection \( \sqsubseteq \cap \sqsubseteq^{-1} \) is an equivalence and often it is closely related to the equivalence we are interested in.

In order to be useful, a refinement relation \( \sqsubseteq \) should have the following properties. It should be compatible with the equivalence \( \approx \) we are using in the sense that if \( P_1 \sqsubseteq P_2 \) and \( P'_1 \approx P_1, P'_2 \approx P_2 \), then \( P'_1 \sqsubseteq P'_2 \). Furthermore, it should be congruent with the operators, i.e. if \( f : C \times C \rightarrow C \) is an operator and \( P_1 \sqsubseteq P'_1, P_2 \sqsubseteq P'_2 \), then \( f(P_1, P_2) \sqsubseteq f(P'_1, P'_2) \). A refinement relation should preserve the same essential
properties as the corresponding equivalence, for example deadlocks and livelocks, divergences or traces. These are preserved in the sense that if $P_1$ has a certain property, such as $P_1$ can be deadlocked, and $P_1 \sqsubseteq P_2$, then $P_2$ also has the same property, i.e. $P_2$ can deadlock, too.

We have various possibilities to define processes in the context of equivalences and refinement relations. Firstly, we can define processes completely and compare them with the help of refinement relations and equivalences. Secondly, we can develop processes by stepwise refinement and point out explicitly, which parts in the process are defined incompletely. In the latter alternative the idea is that among completely defined processes a refinement relation agrees with the corresponding equivalence. This property seems to be valuable and we adopt the latter alternative when defining processes. Of course we have to modify the definitions of equivalences, too.

This thesis studies equivalences and refinement relations that are suitable for the stepwise refinement of Lotos specifications. We start with partially defined basic Lotos specifications, i.e. Lotos specifications without data part. We modify certain existing failure equivalences so that they can be applied to this situation and compare them with the modified bisimulation equivalences. In this approach only states can be refined. In full Lotos it is, however, desirable to refine also the data. In general, this is a difficult demand, if the semantics of the parallel operator is based on interleaving. Fortunately, the data refinement in full Lotos can be done in spite of the interleaving semantics. The solution is not perfect but hopefully useful in practice. Data refinement also shows some differences between bisimulation and failure equivalences.

We apply the techniques in various places. The first applications are small and they are done either manually or using Basic Lotos. At the end we examine a bigger example, Bounded Retransmission Protocol. Its data part is moderately complex and thus it gives a good example about applying refinements in realistic full Lotos cases. First we apply only state refinement. After designing the protocol in this way, we start all over again and refine the data, as well.

1.2 The new definitions and results

The starting point in this thesis has been the bisimulation and partial bisimulation refinements introduced in [Cleaveland and Steffen 90]. I have proved the congruence properties of these relations with respect to the basic Lotos operators. It turned out that it was possible to follow a similar proof in [Walker 90]. However, Walker considered CCS instead of Lotos and his relations were not exactly ours so that it was necessary to make changes and additions.

I have defined the divergence, BKO- and CFFD-refinements. The first was an untrivial modification of the bisimulation refinement, but the two others demanded more consideration. Again, the congruence properties with respect to Lotos were proved. In the case of CFFD-refinement, it was possible to follow the congruence
proof in [Valmari and Tienari 95] with some additions.

After the chapter five we have a lot of different equivalences and refinement relations. The first task to make order of this chaos is to compare these relations with each other. For the most part the comparisons follow directly from the definitions. The only exception is the relationship between CFFD-refinements and bisimulation-based refinements. I have given a fairly detailed and hence lengthy but not very difficult proof relating CFFD-refinements with other refinement relations.

In chapter 7, I have shown how the BKO- and CFFD-refinements can be computed algorithmically. It turned out that it is possible to apply the same ideas as was used computing the CFFD-equivalence in the article [Valmari and Tienari 91], but in the case of refinement relations the algorithm is essentially more complicated than in the case of equivalences.

A conjecture about the structure and interpretation of refinement relations was formulated by Roope Kaivola. Its closer investigation reveals interesting differences between the bisimulation and failure-based semantics. These observations are new.

I have introduced a method to refine actions in full Lotos in chapter 9. In many specifications complicated and lengthy data definitions are needed. In refining a specification stepwise, it is not natural to define all data items completely already in the first step. My method to refine data works in practice as its application to the Bounded Retransmission Protocol shows. This idea seems to be valuable in practice although it is not theoretically perfect.

The first two applications in chapter 4 have been taken from the literature whereas the distributed detection of deadlocks is new. I have analysed the small example in 4.3 semi-automatically. It means that partial state graphs have first been generated. After this, I have examined the graphs manually. This process could be fully automated.

The large example of Bounded Retransmission Protocol has been taken from [Mateescu 96]. However, this presentation did not apply stepwise refinement whereas my approach was based on it.

I have implemented algorithms to compute BKO- and CFFD-refinement relations. The implementation of BKO-relation also works in the case of partial actions, but CFFD-implementation only works in the case of partially defined states. The programs are prototypes, written in Ada95, and they have been used in a Linux environment.
1.3 About notations

I have used traditional mathematical conventions and symbols:

- *Set* can contain only one copy of an element.
- Subset relation $A \subseteq B$ allows the possibility $A = B$.
- *Set difference* is written $A \setminus B$ instead of $A - B$.
- *Relation* in a set $A$ is a subset $R \subseteq A \times A$. Generally, a relation between sets $A_1, A_2, \ldots, A_n$, $n \geq 2$, is a subset $R \subseteq A_1 \times A_2 \times \cdots \times A_n$.
- I have followed the usual mathematical typography in mathematical formulas. To take an example, if the name of a function or predicate is a natural word or an abbreviation like $sin$, $log$, $Div$, then this name is written in mathematical formulas using the ordinary roman type instead of mathematical type $\textit{sin}$, $\textit{log}$, $\textit{Div}$. It seems that these good conventions, widely adopted in mathematics, are not universally followed in computer science.
Chapter 2

Basic Concepts

2.1 Partially defined Lotos processes

We study the theory and applications of partially defined Lotos processes in the spirit of the papers [Cleaveland and Steffen 90] and [Celikkan and Cleaveland 95]. Such processes are constructed with the help of the process $\text{undef}$. In what follows, we think that $\text{undef}$ is part of the Lotos language like $\text{stop}$ and $\text{exit}$. Its semantics is practically the same as the semantics of $\text{stop}$, i.e. no transition is possible from $\text{undef}$. But in order to be able to separate $\text{stop}$ and $\text{undef}$ formally, we define that $\text{undef} \xrightarrow{\eta} \text{undef}$. Here $\eta$ is a reserved action like $\delta$ in normal Lotos. It represents the fact that some actions might be possible from the state $\text{undef}$. The process $\text{undef}$ can be combined with other processes using the Lotos operators action prefix, choice, hiding, parallel composition, sequential composition and disabling. The operational semantics of these operators are defined in the table 2.1. We assume that the basic properties of these operators are familiar to the reader. (see for instance [Bolognesi and Brinksma 87]).

We take the liberty to use different symbols compared to standard Lotos. The internal action in diagrams is here ‘$?$’ instead of ‘$?'$. In hiding and parallel composition, the original Lotos conventions are somewhat restrictive, because Lotos does not use symbols as parameters; we use set symbols to represent the parameter list in these operations. So we write $P|B/Q$ and $h_B(P)$, where $B$ is a set of actions. With these modifications we try to improve the readability of the theory. However, sometimes we use the original symbols in examples.

As it is well known, every Lotos behavioural expression corresponds to a labeled transition system and vice versa. Thus, we can speak about the states of a Lotos process or expression. For theoretical considerations, we introduce a predicate $\uparrow$ which indicates, if the initial state of a Lotos process is partially defined or not. If $P$ is a process with a partially defined initial state, we write $P \uparrow$. In practice, this means that new transitions can be added to the initial state. If the initial state of $P$ is totally defined, we write $P \downarrow$. Formally we define
<table>
<thead>
<tr>
<th>Process</th>
<th>Assumptions</th>
<th>Possible transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>udef</td>
<td></td>
<td>udef $\xrightarrow{\eta} udef$</td>
</tr>
<tr>
<td>stop</td>
<td></td>
<td>no transitions</td>
</tr>
<tr>
<td>exit</td>
<td></td>
<td>exit $\xrightarrow{\delta} stop$</td>
</tr>
<tr>
<td>i;B</td>
<td></td>
<td>i; B $\xrightarrow{i} B$</td>
</tr>
<tr>
<td>$\alpha;B$</td>
<td></td>
<td>$\alpha;B$ $\xrightarrow{\alpha} B'$</td>
</tr>
<tr>
<td><strong>hide</strong> $g_1, \ldots, g_n$ in $B$</td>
<td>$B\xrightarrow{\alpha} B'$, $\alpha \notin {g_1, \ldots, g_n}$</td>
<td>hide $g_1, \ldots, g_n$ in $B$ $\xrightarrow{\alpha}$ hide $g_1, \ldots, g_n$ in $B'$</td>
</tr>
<tr>
<td></td>
<td>$B\xrightarrow{\alpha} B'$, $\alpha \in {g_1, \ldots, g_n}$</td>
<td>hide $g_1, \ldots, g_n$ in $B$ $\xrightarrow{i}$ hide $g_1, \ldots, g_n$ in $B'$</td>
</tr>
<tr>
<td>$B[h_1/g_1, \ldots, h_n/g_n]$</td>
<td>$B\xrightarrow{\alpha} B'$</td>
<td>$B[h_1/g_1, \ldots, h_n/g_n] \xrightarrow{h_i}$ $B[h_1/g_1, \ldots, h_n/g_n]$</td>
</tr>
<tr>
<td></td>
<td>$B\xrightarrow{\alpha} B'$, $\alpha \notin {g_1, \ldots, g_n}$</td>
<td>$B[h_1/g_1, \ldots, h_n/g_n] \xrightarrow{\alpha}$ $B[h_1/g_1, \ldots, h_n/g_n]$</td>
</tr>
<tr>
<td>$P[h_1, \ldots, h_n]$</td>
<td>$B_p$ is the definition of $P$ and $P[h_1, \ldots, h_n] \xrightarrow{\alpha} B'$</td>
<td>$P[h_1, \ldots, h_n] \xrightarrow{\alpha} B'$</td>
</tr>
<tr>
<td>$B_1 \parallel B_2$</td>
<td>$B_1 \xrightarrow{\alpha} B'_1$</td>
<td>$B_1 \parallel B_2 \xrightarrow{\alpha} B'_1$</td>
</tr>
<tr>
<td></td>
<td>$B_2 \xrightarrow{\alpha} B'_2$</td>
<td>$B_1 \parallel B_2 \xrightarrow{\alpha} B'_2$</td>
</tr>
<tr>
<td>$B_1</td>
<td></td>
<td>g_1, \ldots, g_n$</td>
</tr>
<tr>
<td></td>
<td>$B_2 \xrightarrow{\alpha} B'_2$, $\alpha \notin {g_1, \ldots, g_n, \delta}$</td>
<td>$B_1</td>
</tr>
<tr>
<td></td>
<td>$B_1 \xrightarrow{\alpha} B'_1$, $B_2 \xrightarrow{\alpha} B'_2$, $\alpha \in {g_1, \ldots, g_n, \delta}$</td>
<td>$B_1</td>
</tr>
<tr>
<td>$B_1 &gt;&gt; B_2$</td>
<td>$B_1 \xrightarrow{\alpha} B'_1$, $\alpha \neq \delta$</td>
<td>$B_1 &gt;&gt; B_2 \xrightarrow{\alpha} B'_1 &gt;&gt; B_2$</td>
</tr>
<tr>
<td></td>
<td>$B_1 \xrightarrow{\delta} B'_1$</td>
<td>$B_1 &gt;&gt; B_2 \xrightarrow{\delta} B'_2$</td>
</tr>
<tr>
<td>$B_1</td>
<td>&gt; B_2$</td>
<td>$B_1 \xrightarrow{\alpha} B'_1$</td>
</tr>
<tr>
<td></td>
<td>$B_1 \xrightarrow{\delta} B'_1$</td>
<td>$B_1</td>
</tr>
<tr>
<td></td>
<td>$B_2 \xrightarrow{\alpha} B'_2$</td>
<td>$B_1</td>
</tr>
</tbody>
</table>

Figure 2.1: Semantics of Lotos operators
2.1. PARTIALLY DEFINED LOTOS PROCESSES

- stop↓, exit↓;
- if a is an action, B a set of actions, τ, δ, η ⊈ B, and P and Q are processes then
  1. (a.P) ↓;
  2. if P ↓ and Q ↓, then (P + Q) ↓, (P|> Q) ↓, (P|B|Q) ↓;
  3. if P ↓, then (P >> Q) ↓ and h_B(P) ↓.

These rules define when the initial state of a process is totally defined. If the initial state is not totally defined, it is partially defined.

A behavioural expression written in basic Lotos with udef corresponds with a partially defined labeled transition system.

**Definition 2.1.1** A partially defined labeled transition system P is a tuple (S, A, T, ↑, q_0), where

- S is a countable set of states;
- A is a countable set of actions; it can contain the special elements τ, the internal (invisible) action, and δ, the successful termination; on the other hand, η ⊈ A;
- T ⊂ S × A × S is a transition relation. We denote by T_a the set T \cap (S × \{a\} × S) for every a ∈ A; it is supposed that T_a ≠ ∅ for every a ∈ A;
- ↑ ⊂ S is a partiality predicate;
- q_0 ∈ S is the initial state; we demand that every state in S is reachable from the initial state q_0 when the system is interpreted as a graph.

We write p → a q instead of (p, a, q) ∈ T and p ↑ if p ∈ ↑. If p ∉ ↑, then we use the notation p ↓. If p ↓ for all p ∈ S, then P is totally defined or specified, otherwise P is partially specified.

We don't allow η-transitions in transition systems. Instead of η, the predicate ↑ shows, if a state is partially defined. It seems to be conceptually easier to use η in Lotos processes and ↑ in transitions systems. Later we give rules which show how to generate a transition system from a Lotos expression.

Let ALTS (augmented labeled transition systems) be the set of partially defined labeled transition systems. Note that in our terminology totally defined labeled transition systems also belong to ALTS. We assume that every action symbol is a finite string constructed using a fixed finite alphabet. In addition, it is assumed that there is a countable superset of states and the state set in an arbitrary transition system is a subset of this superset. By these assumptions ALTS is countable and well defined as a set. We do not define this superset explicitly and feel free to use whatever names seem to be practical in concrete situations. The same is true for action names, but it is sometimes convenient to have a symbol for the set of all actions. Let A be the set of all possible action symbols in all possible transition systems. If (S, A, T, ↑, q_0) is a labeled transition system, then A ⊂ A.
The special action $\delta$ plays an important role in sequential composition of processes and in disabling. In order to achieve consistency with Lotos expressions, we have to demand that every transition system is $\delta$-consistent, that is, if $(s, \delta, s') \in T_\delta$ then $s' \downarrow$ and there are no $a$ and no $s''$ such that $(s', a, s'') \in T_a$. This means that after executing a $\delta$-transition in a given labeled transition system no more executable transitions can be reached.

If $P = (S, A, T, \uparrow, q_0) \in \text{ALTS}$ and $q \in S$, then we can consider $P' = (S, A, T, \uparrow, q)$ as another transition system with the initial state $q$. Of course, we should also modify the sets $S$, $A$ and $T$, if there are unreachable states in the new system. But usually we are interested only in the executable transitions and don't bother to remove the superfluous states explicitly. We use the notation $P \xrightarrow{a} P'$, when $(q_0, a, q) \in T$.

2.2 Concept of refinement

We now have enough concepts to define informally, what we mean by refinement in practice. Theoretically, refinements can be more complex, because we consider processes modulo an equivalence. But the practical refinement technique we describe in this section can be applied with respect to every equivalence and refinement relation we define in this thesis.

The idea to refine programs or algorithms is not new. However, our method has very little in common, for example, with Dijkstra's refinement method ([Dijkstra 76], [Dijkstra and Scholten 89]) or its modern versions ([Back 78], [Back and Wright 98]). Closer to our approach is the practice to apply preorders or implementation relations to verify programs. A good example is the article [Hoare, Jifeng 91], which uses CSP and refinement relations to develop a verified compiler. The method used in their article differs essentially from ours anyway, because partially defined states or processes are not used.

Consider first an algebraic expression with one or more udef. A refinement of this expression is any expression, which has been constructed by replacing one or more udef's with another process. These other processes may be totally defined or they may contain udef.

Consider next a labeled transition system with partially defined states. A refinement of this is a transition system which is constructed by adding transitions from partially defined states. After adding transitions, the state can still be partially defined or it can be changed into totally defined. It is also possible that a partially defined state is changed into totally defined without adding new transitions. The new transitions can lead to new, totally or partially defined states, or they can lead to already existing states.

These two ways to refine are, as a matter of fact, complementary representations of the same thing. More precisely, a Lotos expression with udef corresponds with a partially defined labeled transition system as follows. First we generate a labeled transition system in a normal way. Every state in this transition system corre-
2.2. CONCEPT OF REFINEMENT

responds with a subexpression in the original Lotos expression. We remove all the \( \eta \)-transitions and mark the corresponding states as partially defined. For example, the expression

\[ a; b; \text{stop} \parallel [a] \parallel a; c; \text{udef} \]

gives rise to the transition system

\[ a; b; \text{stop} \parallel [a] \parallel a; c; \text{udef} \]

\[ \downarrow \]

\[ b; \text{stop} \parallel [a] \parallel c; \text{udef} \]

\[ a \]

\[ b \]

\[ c \]

\[ \text{stop} \parallel [a] \parallel \text{udef} \]

\[ \downarrow \]

\[ \text{stop} \parallel [a] \parallel \text{udef} \]

Consider also the process

\[ a; \text{udef} \parallel b; \text{udef}. \]

It generates the transition system

\[ a; \text{udef} \parallel b; \text{udef} \]

\[ \downarrow \]

\[ a \]

\[ b \]

\[ \text{udef} \]

\[ \text{udef} \]

\[ \text{udef} \]

Under the normal Lotos semantics this is same as

\[ a; \text{udef} \parallel b; \text{udef} \]

\[ \text{udef}^a \rightarrow \text{udef} \]

This does not correspond with our intuition. In the original process definition there are two points where the refinement can start from and the two refinements can be different. On the other hand, in the latter diagram there is only one possible point of refinement. We draw the diagrams in the former way. We could define formally that every \text{udef} in a Lotos expression is indexed with a unique natural number. This guarantees that different \text{udef}-processes are not identified. We don’t use indexes in diagrams, because we draw them so that \text{udef}-processes have already been separated. Furthermore, it is not necessary to use indexes in the forthcoming definitions of equivalences and refinement relations, because the definitions are constructed so that they do not depend on this detail. We return to this question later, when we define the weak bisimulation equivalence.
Every partially defined transition system can be transformed to a corresponding Lotos expression. Consider for example the following transition systems:

\[
\begin{align*}
\rightarrow & \quad P_1 \uparrow \overset{a}{\rightarrow} P_2 \overset{a}{\rightarrow} P_3 \uparrow, \\
\rightarrow & \quad Q_1 \overset{a}{\rightarrow} Q_2 \uparrow \overset{a}{\rightarrow} Q_3, \\
\rightarrow & \quad R_1 \overset{a}{\rightarrow} R_2 \uparrow \overset{a}{\rightarrow} R_3 \overset{a}{\rightarrow} R_4 \uparrow.
\end{align*}
\]

These correspond with the Lotos expressions

\[
\begin{align*}
P & = (a; a; \text{undef}) \parallel \text{undef}, \\
Q & = a; (a; \text{stop}) \parallel \text{undef}, \\
R & = a; ((a; a; \text{undef}) \parallel \text{undef}).
\end{align*}
\]

The usual transition system, for example, generated from the first expression would be

\[
\begin{tikzpicture}
    \node (a) [state, initial] {a; \text{undef}};
    \node (eta) [state, below of=a] {\text{eta}};
    \node (eta1) [state, below of=eta, anchor=west] {\text{eta}};
    \draw[->, bend left] (a) to node[auto,swap] {$a$} (eta);
    \draw[->, bend right] (a) to node[auto] {$a$} (eta1);
    \end{tikzpicture}
\]

if we also draw the $\eta$-transitions.

### 2.3 Lotos operators and transition systems

The Lotos operators are also operators between transition systems, i.e., they are mappings on ALTS or on $\text{ALTS} \times \text{ALTS}$ to ALTS. The usual operational semantics of the operators combined with the rules for the partiality predicate determines directly the resulting transition systems. There is one subtle point concerning process instantiation in Lotos expressions: if we want a complete correspondence between Lotos expressions and transition systems, we should define something equivalent to process instantiation with respect to transition systems. By studying the definition of process instantiation, it can be seen that renaming is a suitable equivalent concept in transition systems. If $P$ is a transition system and $f : A \setminus \{\tau\} \rightarrow A \setminus \{\tau\}$ is a renaming function, then the notation $P[f]$, renaming means the transition system which is like $P$, but the actions $a \neq \tau$ in $P$ are replaced by the actions $f(a)$. If $f(a) \neq a$ only for finitely many actions $a_1, \ldots, a_n$, then we can also use the notation $P[f(a_1)/a_1, \ldots, f(a_n)/a_n].$

The following diagrams depict how the operators transform transition systems.
1. Prefix.

\[ a . \begin{tikzpicture} \node (P) at (0,0) {}; \end{tikzpicture} = \begin{tikzpicture} \node (P) at (0,0) {}; \end{tikzpicture} \]

2. Choice.

\[ \begin{tikzpicture} \node (P) at (0,0) {}; \end{tikzpicture} \begin{tikzpicture} \node (Q) at (1,0) {}; \end{tikzpicture} = \begin{tikzpicture} \node (P) at (0,0) {}; \end{tikzpicture} \begin{tikzpicture} \node (Q) at (1,0) {}; \end{tikzpicture} \]


\[ \begin{tikzpicture} \node (P) at (0,0) {}; \end{tikzpicture} \begin{tikzpicture} \node (Q) at (1,0) {}; \end{tikzpicture} = \begin{tikzpicture} \node (P) at (0,0) {}; \end{tikzpicture} \begin{tikzpicture} \node (Q) at (1,0) {}; \end{tikzpicture} \]

4. Disabling.

\[ \begin{tikzpicture} \node (P) at (0,0) {}; \end{tikzpicture} \begin{tikzpicture} \node (Q) at (1,0) {}; \end{tikzpicture} = \begin{tikzpicture} \node (P) at (0,0) {}; \end{tikzpicture} \begin{tikzpicture} \node (Q) at (1,0) {}; \end{tikzpicture} \]

5. Hiding has no influence on states.

6. Renaming has no influence on states.
7. Parallel composition generates the well-known reachability graph from the component processes. The transitions follow from the operational semantics of the parallel operator.

In a Lotos specification there is always a finite number of actions (or gates, if we are using full Lotos). But if we are using full Lotos and the corresponding transition system is infinite, then the transition system may contain an infinite number of different actions. Thus when dealing with transition systems, the parameter $B$ in the hiding operator $h_B$ and parallel operator $-|B|$ may be an infinite set. The same is true for renaming functions, i.e. $f(a) \neq a$ can happen for infinite many $a \in A$.

## 2.4 Equivalences, preorders and graphs

Let $C$ be a set. A reflexive, symmetric and transitive relation $\approx \subseteq C \times C$ is an equivalence. If $x \in C$, then the set

$$[x]_\approx = \{y \in C \mid (x, y) \in \approx\}$$

is the equivalence class of the element $x$. When the equivalence is known from the context we write $[x]$ instead of $[x]_\approx$. We also write $x \approx y$ instead of $(x, y) \in \approx$.

A collection $\mathcal{H}$ of subsets of $C$ is a partition, if

**H1.** every $B \in \mathcal{H}$ is non-empty,

**H2.** $\bigcup_{B \in \mathcal{H}} B = C$,

**H3.** $B_1 \cap B_2 = \emptyset$ for every $B_1, B_2 \in \mathcal{H}, B_1 \neq B_2$.

**Proposition 2.4.1** If $\approx \subseteq C \times C$ is an equivalence, then the set of equivalence classes, $C/\approx$, is a partition of $C$. If $x$ and $y \in C$, then $x \approx y$ if and only if $x$ and $y$ belong to the same equivalence class.

Let $\mathcal{H}$ be a partition of $C$. If a relation $R_\mathcal{H} \subseteq C \times C$ is defined by demanding that $(x, y) \in R_\mathcal{H}$ if and only if $x$ and $y$ belong to the same set in the partition $\mathcal{H}$, then $R_\mathcal{H}$ is an equivalence relation.

Furthermore, if $\mathcal{H} = C/R$, where $R$ is an equivalence relation, then $R_\mathcal{H} = R$.

**Proof.** See introductory textbooks on algebra. $\Box$

The proposition shows that we can deal with equivalences using either relations or partitions. Which one is chosen depends on the context. Typically, relations are used in theoretical considerations whereas algorithms are based on partitions.
2.4. EQUIVALENCES, PREORDERS AND GRAPHS

Definition 2.4.1 Let $C$ be a set and $f : C \to C$, $g : C \times C \to C$ mappings. An equivalence relation $\approx$ on $C$ is a congruence with respect to $f$ and $g$, if $x \approx y$ and $z \approx v$ imply $f(x) \approx f(y)$ and $g(x, z) \approx g(y, v)$. In this case we also say that $f$ and $g$ are compositional with respect to $\approx$.

Proposition 2.4.2 An equivalence relation $\approx$ on $C$ is congruence with respect to $f : C \to C$ and $g : C \times C \to C$, if and only if $f$ and $g$ induce mappings

$$
\tilde{f} : C/\approx \longrightarrow C/\approx, \quad \tilde{f}([P]) = [f(P)],
$$

$$
\tilde{g} : C/\approx \times C/\approx \longrightarrow C/\approx, \quad \tilde{g}([P_1], [P_2]) = [g(P_1, P_2)].
$$

Proof. The functions $\tilde{f}$ and $\tilde{g}$ are well-defined only if their values are independent of the chosen representative in an equivalence class. If $\approx$ is a congruence and $x \in C$, then $f(x) = f(y)$ for all $y \in [x]$. In other words, $\tilde{f}$ is well-defined. Similarly, if $g$ is a congruence, it is well-defined.

If $\tilde{f}$ and $\tilde{g}$ are well-defined, then for all $y \in [x]$ and $v \in [z]$ both $[f(x)] = [f(y)]$ and $[g(x, z)] = [g(y, v)]$. But this means that $f(x) \approx f(y)$ and $g(x, z) \approx g(y, v)$. So $f$ and $g$ are compositional with respect to $\approx$. □

Proposition 2.4.3 If $g$ is commutative, i.e. $g(x, y) = g(y, x)$ for all $x, y \in C$, and if $x \approx y$ implies $g(x, z) \approx g(y, z)$ for all $z \in C$ then $g$ is a congruence with respect to $\approx$.

Proof. Let $x \approx y$ and $z \approx v$ and suppose that $g$ is commutative and the condition in the proposition is valid. Then

$$
g(x, z) \approx g(y, z) = g(z, y) \approx g(v, y) = g(y, v).
$$

□

Definition 2.4.2 A preorder is a reflexive and transitive relation.

Proposition 2.4.4 If $\preceq$ is a preorder, then $\preceq \cap \preceq^{-1}$ is an equivalence.

Proof. Follows directly from the definitions. □

Proposition 2.4.5 If $E_i, i \in I$, is an arbitrary collection of equivalences and $R_j, j \in J$, is any collection of preorders, then $\bigcap_{i \in I} E_i$ is an equivalence and $\bigcap_{j \in J} R_j$ is a preorder.

Proof. Straightforward. □

It should be noted that an arbitrary union of equivalences or the complement of an equivalence relation are not necessarily equivalences.
Proposition 2.4.6 If a preorder $\sqsubseteq$ is congruent with respect to an operator $f : C \to C$, i.e. the property $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$, then the equivalence $\approx = \sqsubseteq \cap (\sqsubseteq)^{-1}$ is a congruence with respect to $f$. If $g$ defines a mapping $C \times C \to C$ and $x \sqsubseteq y$, $z \sqsubseteq v$ implies $g(x, z) \sqsubseteq g(y, v)$, then $\approx$ is a congruence with respect to $g$.

Proof. By the definition of congruence, we have to prove that $x \approx y$ implies $f(x) \approx f(y)$. This is clear, because $x \sqsubseteq y$ and $y \sqsubseteq x$ and thus $f(x) \sqsubseteq f(y), f(y) \sqsubseteq f(x)$, i.e. $f(x) \approx f(y)$. In the case of $g$ we can reason in the same way. □

A labeled directed graph consists of a set $N$ of nodes, a set $A$ of labels and a relation $E \subset N \times A \times N$. The relation $E$ is called the set of arcs. An element $(v, a, w) \in E$ is called an arc labeled with $a$. For every $a \in A$ we denote by $E_a$ the set of triples $(v, a, w)$ such that $(v, a, w) \in E$. The set $E_a$ is called the set of arcs labeled with $a$.

A path $(v_1, a_1, v_2, a_2, \ldots, a_{k-1}, v_k)$ in a labeled directed graph is a sequence of arcs, where $v_i, 1 \leq i \leq k$, are nodes, $a_j, 1 \leq j < k$, are labels and $(v_i, a_i, v_{i+1}) \in E$ for every $i = 1, 2, \ldots, k - 1$. The length of the path is $k - 1$, the number of arcs along the path. The trivial case $k = 1$, empty path, is permitted. According to the previous definition, a path is of finite length. Later we also define infinite paths, but the term path without attributes always refers to a finite path.

Suppose $G_1$ and $G_2$ are graphs with node sets $V_1$ and $V_2$, label sets $A_1$ and $A_2$ and arc sets $E_1$ and $E_2$, respectively. The graphs $G_1$ and $G_2$ are isomorphic, if $A_1 = A_2 = A$ and there is a bijective mapping $f : V_1 \to V_2$ such that $(v, a, w) \in E_1$ if and only if $(f(v), a, f(w)) \in E_2$. If $\mathcal{G}$ is a set of labeled directed graphs, then the graph isomorphism is an equivalence relation on this set. Two labeled transition systems $P$ and $Q$ are isomorphic, if they are isomorphic as labeled graphs and the initial states map to each other. In this case we write $P \equiv Q$.

We can express some properties of the basic Lotos operators with the help of the isomorphism relation $\equiv$:

1. $P | Q \equiv Q | P$.
2. $P | B | Q \equiv Q | B | P$.
3. Parallel composition is not associative, i.e. it may be that $(P | B_1 | Q) | B_2 | R \not\equiv P | B_1 | (Q | B_2 | R)$.
4. Let $P, Q$ and $R$ be processes with action sets $L_P$, $L_Q$ and $L_R$, respectively. Then

$$P | L_P \cap (L_Q \cup L_R) | (Q | L_Q \cap L_R | R) \equiv (P | L_P \cap L_Q | Q) | (L_P \cup L_Q) \cap L_R | R.$$
A state in the processes on both sides of the equality can be represented by a triple \((P_i, Q_j, R_k)\), where a component in the triple is a state of the respective component process. We show that for every transition in the first system there is a corresponding transition in the second system and the other way around. Let \((P_1, Q_1, R_1)\) be the initial state of \(A = P|L_P \cap (L_Q \cup L_R)||Q|L_Q \cap L_R|R\). It is also the initial state of \(B = P|L_P \cap L_Q|Q|(L_P \cup L_Q) \cap L_R|R\). Suppose \((P_1, Q_1, R_1) \xrightarrow{a} (P_i, Q_j, R_k)\) is a transition in \(A\). We show that it is also a transition in \(B\). There are several alternatives to be checked.

(i) Suppose \(a \notin L_P \cap (L_Q \cup L_R)\) and \(P_1 \xrightarrow{a} P_i, Q_j = Q_1, R_k = R_1\). Because \(a \in L_P\), it follows that \(a \notin L_Q \cup L_R\) and thus \(a \notin (L_P \cup L_Q) \cap L_R\) and \(a \notin L_P \cap L_Q\). So we have the transition \((P_1, Q_1, R_1) \xrightarrow{a} (P_i, Q_1, R_1)\) also in \(B\).

(ii) Suppose \(a \notin L_P \cap (L_Q \cup L_R), a \notin L_Q \cap L_R\) and \((P_i, Q_j, R_k) = (P_1, Q_j, R_1)\). Because \(a \in L_Q\), it follows that \(a \notin L_P \cap L_Q\) and \(a \notin (L_P \cup L_Q) \cap L_R\). Thus the transition is also in \(B\). The case \((P_1, Q_j, R_k) = (P_1, Q_1, R_1)\) is similar.

(iii) Suppose \(a \notin L_P \cap (L_Q \cup L_R), a \notin L_Q \cap L_R\), but \(a \in L_Q \cap L_R\) and the next state in \(A\) is \((P_1, Q_j, R_k)\). Because \(a \notin L_P\), we have \(a \notin L_P \cap L_Q\), \(a \in (L_P \cup L_Q) \cap L_R\). So in \(P|L_P \cap L_Q|Q\) there is the transition \((P_1, Q_1) \xrightarrow{a} (P_1, Q_j)\). We can use this transition and the transition \(R_1 \xrightarrow{a} R_k\) to get the transition from the initial state in \(B\) to the state \((P_1, Q_j, R_k)\).

(iv) If \(a \in L_P \cap (L_Q \cup L_R)\) and \(a \in L_Q \cap L_R\), then \(a \in L_P \cap L_Q \cap L_R\). Especially, \(a \in L_P \cap L_Q\) and \(a \in (L_P \cup L_Q) \cap L_R\).

iv) If \(a \in L_P \cap (L_Q \cup L_R)\) and \(a \notin L_Q \cap L_R\), then \(a \in L_P \cap L_Q\) or \(a \in L_P \cap L_R\). If \((P_1, Q_1, R_1) \xrightarrow{a} (P_i, Q_j, R_1)\), then \(a \in L_P \cap L_Q\), \(a \notin L_P \cap L_R\). Hence the transition is also a transition in \(B\). If \((P_1, Q_1, R_1) \xrightarrow{a} (P_1, Q_1, R_k)\), then \(a \in L_P \cap L_R, a \notin L_P \cap L_Q\). In this case the transition is also a transition in \(B\).

Thus in all the cases a transition by \(a\) in \(A\) is also a transition in \(B\). That a transition in \(B\) is also a transition in \(A\) is proved in the same way. □

5. For an arbitrary action set \(B\),

\[
P|B|(Q|B|R) \equiv (P \mid B \mid Q) \mid B \mid R.
\]

The above formula can be proved in the same way as the previous formula with the exception that now there are fewer alternatives \((a \in B\) and \(a \notin B\)).

6. \(P|B_1|(Q|B_2|R) \equiv (P|B_1|Q)|B_2|R\), if \(L_P \cap B_2 = \emptyset\) and \(L_R \cap B_1 = \emptyset\). The proof is a modification of the proof in 4 (see also [van Sinderen 89]).

2.5 Notations

Let \(P = (S, A, T, \uparrow, q_0) \in \text{ALTS}\) be a process, \(p, q \in S\) states, \(a \in A\) an action and \(u = a_1 a_2 \cdots a_n \in (A \setminus \{\tau\})^*\) a string of actions. A string can also be an empty string
We use the following notations and concepts throughout this report.

1. There is a $\tau$-path from a state $p$ to a state $q$ in $P$, if $p = q$ or there are transitions

$$p = p_1 \xrightarrow{\tau} p_2 \xrightarrow{\tau} \cdots \xrightarrow{\tau} p_n \xrightarrow{\tau} q.$$ 

In this case we write $p \Rightarrow q$.

2. We write $p \Rightarrow r_1 \xrightarrow{\tau} r_2 \Rightarrow q$, if $p \Rightarrow r_1 \xrightarrow{\tau} r_2 \Rightarrow q$ for some states $r_1$ and $r_2$.

3. A string $u$ leads from $p$ to $q$, $p \xrightarrow{u} q$, if $u = \varepsilon$ and $p \Rightarrow q$ or there is a path

$$p \Rightarrow p_1 \xrightarrow{a_1} r_1 \Rightarrow p_2 \xrightarrow{a_2} r_2 \Rightarrow \cdots \Rightarrow p_n \xrightarrow{a_n} r_n \Rightarrow q,$$

where $u = a_1a_2\cdots a_n$, $n \geq 1$. If there is no path from $p$ to $q$ with $u$, then we denote this by $p \xrightarrow{u} q$. Note that $p \xrightarrow{\varepsilon} q$ means the same as $p \Rightarrow q$.

4. There is a path from $p$ with a string $u$, $p \xrightarrow{u}$, if there is a state $q$ such that $p \xrightarrow{u} q$. If there is no path from $p$ with $u$, we denote this by $p \xrightarrow{u}$.

5. If $p$ is a state, we write $p \uparrow$, if for some state $q$, $p \Rightarrow q$ and $q \uparrow$.

6. A string $u$ leads from a state $p$ to a partially defined state, $p \uparrow u$, if there is a state $q$ such that $u = u_1u_2$, $p \xrightarrow{u_1} q$ and $q \uparrow$.

7. We write $p \downarrow u$ if not $p \uparrow u$.

8. We can use processes instead of states in these definitions. For example, if $P$ and $P'$ are processes, then $P \xrightarrow{u} P'$ means that $p \xrightarrow{u} p'$, where $p$ is the initial state of $P$ and $p'$ is the initial state in $P'$ (of course, $p'$ is also a state in $P$).

9. The trace set is defined by

$$\operatorname{tr}(P) = \{ u \in A^* \mid P \xrightarrow{u} \}.$$ 

10. The boolean function $\operatorname{Div}(P)$, defined on the set $\operatorname{ALTS}$, is true if and only if there is an infinite $\tau$-path

$$p_1 \xrightarrow{\tau} p_2 \xrightarrow{\tau} \cdots,$$

where $p_1$ is the initial state of $P$. This unary divergence function is extended to the binary function $\operatorname{Div}(P, u)$, where $P$ is a process and $u$ is a string of actions, by defining $\operatorname{Div}(P, u)$ to be true if and only if $\operatorname{Div}(P)$ or there is a $P'$ such that for some prefix $u'$ of $u$, $P \xrightarrow{u'} P'$ and $\operatorname{Div}(P')$. If not $\operatorname{Div}(P)$ or $\operatorname{Div}(P, u)$, then we write $\operatorname{Conv}(P)$ or $\operatorname{Conv}(P, u)$, respectively. Notice that $\operatorname{Div}(P)$ if and only if $\operatorname{Div}(P, \varepsilon)$. 


Chapter 3

Weak Bisimilarity and Related Refinement Relations

In this and the following chapters we extend the well known concepts bisimulation, weak bisimulation equivalence etc. to include partially defined systems. The generalizations are made in such a way that in the absence of partially defined states the generalized concepts coincide with the original ones. This results in terminological problems. We resolve these problems by using the customary terms also in the context of partially defined systems. If one wants to make a distinction between the customary and generalized concepts, it is possible to use the qualifying word "specification" with the generalized concepts, e.g. bisimulation — specification bisimulation, weak bisimulation equivalence — weak specification bisimulation equivalence etc. In definitions, we usually point out explicitly, if a new concept is a generalization (specification in parentheses) or really a new one (without specification in parentheses). Weak bisimulation equivalence and bisimulation refinement in the context of partially defined states are well known, although the terms can vary ([Cleaveland and Steffen 90]).

3.1 Strong bisimilarity

The strong bisimilarity is one of the strongest process equivalences. It is useful in theoretical considerations as a basic identity relation and the algorithms deciding the strong bisimilarity of two systems usually generalize to other bisimulation equivalences. It is rather near to the isomorphism relation of process graphs. Due to its strength it is not widely used in practical verifications or compositional minimizations and that's why we will not develop refinement relations with respect to strong bisimilarity. It would not be too difficult an exercise for an interested reader to develop the definitions himself on the basis of the weak bisimulation equivalence and refinement relation.
Definition 3.1.1 A relation $\mathcal{R} \subset \text{ALTS} \times \text{ALTS}$ is a strong (specification) bisimulation, if $(P_1, P_2) \in \mathcal{R}$ implies that:

1. $P_1 \uparrow$ if and only if $P_2 \uparrow$;
2. for all $a \in \mathcal{A}$,
   
   a) $P_1 \xrightarrow{a} Q_1$ implies that there is a $Q_2$ such that $P_2 \xrightarrow{a} Q_2$ and $(Q_1, Q_2) \in \mathcal{R}$;
   
   b) $P_2 \xrightarrow{a} Q_2$ implies there is a $Q_1$ such that $P_1 \xrightarrow{a} Q_1$ and $(Q_1, Q_2) \in \mathcal{R}$.

Strong bisimilarity, $\approx_s$, is the largest strong bisimulation

$$\approx_s = \bigcup \{\mathcal{R} | \mathcal{R} \text{ is a strong bisimulation}\}.$$

Thus two processes $P_1$ and $P_2$ are strongly bisimilar, $P_1 \approx_s P_2$, if there is a strong bisimulation containing the pair $(P_1, P_2)$.

We have defined strong bisimilarity for partially defined labeled transition systems. But the definition can also be used in the case where processes are Lotos processes with udef. This is true also for all other equivalence and refinement relation definitions in this thesis.

The relation $\approx_s$ is an equivalence relation. This can be seen by the same kind of reasoning as in the case of weak bisimulation equivalence in the next section. Strong bisimilarity is also congruence with respect to all the Lotos operators.

We have used the predicate $\uparrow$ in the definition. As a matter of fact, the condition 1 in the definition could be left out, if we allowed $a \in \mathcal{A} \cup \{\eta\}$. But we have formulated the definition with $\uparrow$, because it will have a key role in the definitions of the refinement relations. It would be very cumbersome to define refinement relations using only $\eta$.

3.2 Weak bisimilarity

One of the best established equivalences is the weak bisimulation equivalence or in short weak bisimilarity. It is necessary to modify its definition in order to take into account the possibility of partiality in our process concept [Cleaveland and Steffen 90].

Definition 3.2.1 A relation $\mathcal{R} \subset \text{ALTS} \times \text{ALTS}$ is a weak (specification) bisimulation, if $(P_1, Q_1) \in \mathcal{R}$ implies that:

1. $P_1 \uparrow$ if and only if $Q_1 \uparrow$;
2. for all $a \in (\mathcal{A} \setminus \{\tau\}) \cup \{\varepsilon\}$,
   
   a) $P_1 \xrightarrow{a} P_2$ implies that there is a $Q_2$ such that $Q_1 \xrightarrow{a} Q_2$ and $(P_2, Q_2) \in \mathcal{R}$;
3.3. BISIMULATION REFINEMENT

b) \( Q_1 \xrightarrow{a} Q_2 \) implies there is a \( P_2 \) such that \( P_1 \xrightarrow{a} P_2 \) and \((P_2, Q_2)\) \(\in \mathcal{R}\).

Weak bisimilarity, \( \cong_{\text{bis}} \), is now defined as the largest bisimulation,

\[
\cong_{\text{bis}} = \bigcup \{ \mathcal{R} \mid \mathcal{R} \text{ is a bisimulation} \}.
\]

Hence two processes \( P, Q \) are weakly bisimilar, \( P \cong_{\text{bis}} Q \), if there is a bisimulation containing the pair \((P, Q)\).

We have defined the weak bisimilarity using processes instead of states. But as we have explained, a state can be considered the initial state of a process. Thus we can also speak about the equivalence of two states in the same process and combine the equivalent states to form a state minimal process (see [Eloranta 94]).

**Proposition 3.2.1** The relation \( \cong_{\text{bis}} \) is an equivalence relation.

**Proof.** (See [Milner 89].) The claim follows from the fact that the following relations are bisimulations:

a) \( \mathcal{R} = \{(P, P) \mid P \in \text{ALTS}\}. \)

b) \( \mathcal{R}^{-1} = \{(P_2, P_1) \mid (P_1, P_2) \in \mathcal{R}\}, \) where \( \mathcal{R} \) is a bisimulation.

c) \( \mathcal{R} = \{(P_1, P_3) \mid (P_1, P_2) \in \mathcal{R}_1, (P_2, P_3) \in \mathcal{R}_2 \text{ for some } P_2 \in \text{ALTS}\}, \) where \( \mathcal{R}_1 \) and \( \mathcal{R}_2 \) are bisimulations. \(\square\)

The weak bisimulation equivalence is a congruence with respect to parallel composition, prefix, renaming, hiding and sequential composition. The proof is similar to the proof in the next section dealing with the corresponding properties of the related bisimulation refinement.

### 3.3 Bisimulation refinement

The bisimulation refinement, \( \sqsubseteq_{\text{bis}, \text{ref}}, \) ([Cleaveland and Steffen 90]), is used to determine when one process is more specified than another. The relation is defined in terms of prebisimulations.

**Definition 3.3.1** A relation \( \mathcal{R} \subseteq \text{ALTS} \times \text{ALTS} \) is a prebisimulation, if \((P_1, Q_1) \in \mathcal{R}\) implies that the following holds for all \(a \in (\mathcal{A}\backslash \{\tau\}) \cup \{\varepsilon\} \).

1. If \( P_1 \xrightarrow{a} P_2 \), then there is a \( Q_2 \) such that \( Q_1 \xrightarrow{a} Q_2 \) and \((P_2, Q_2) \in \mathcal{R}\).

2. If \( P_1 \downarrow a \), then

   a) \( Q_1 \downarrow a \),

   b) \( Q_1 \xrightarrow{a} Q_2 \) implies there is a \( P_2 \) such that \( P_1 \xrightarrow{a} P_2 \) and \((P_2, Q_2) \in \mathcal{R}\).
By the same kind of technique as in the construction of the weak bisimulation equivalence it can be shown that the largest prebisimulation

\[ \sqsubseteq_{\text{wbs}\text{ref}} = \bigcup \{ \mathcal{R} \mid \mathcal{R} \text{ is a prebisimulation} \} \]

is a preorder. This is the \textit{bisimulation refinement}. It is compositional with respect to all the operators with the exception of choice and disabling. Disabling, however, is compositional in a restricted form.

We can see from the definition of \( \sqsubseteq_{\text{wbs}\text{ref}} \) that different udef's are not differentiated from each other. If we want to test the relation \( P \sqsubseteq_{\text{wbs}\text{ref}} Q \) for two processes \( P \) and \( Q \), we can, for example, first minimize \( P \) to get \( P' \) and then test the condition \( P' \sqsubseteq_{\text{wbs}\text{ref}} Q \). It doesn’t matter in \( \sqsubseteq_{\text{wbs}\text{ref}} \), if different udef's have been identified in \( P' \).

The condition 2 deserves further consideration. As [Walker 90] observes, it is not sufficient to demand in the condition 2 that if \( P_1 \downarrow \) then \( Q_1 \downarrow \) (i.e. without \( a \) in \( \downarrow \)). Namely, let \( P \) and \( Q \) be the following transition systems:

\[
\begin{align*}
P & \xrightarrow{a} P_1 \xrightarrow{b} P_2 \\
Q & \xrightarrow{\tau} Q_1 \xrightarrow{b} Q_2
\end{align*}
\]

If we used \( \downarrow \) without an action, then we would have \( P \sqsubseteq_{\text{wbs}\text{ref}} Q \), because \( \{(P_1, Q_1), (P_2, Q_2)\} \) were a prebisimulation. But \( a.P \nsubseteq_{\text{wbs}\text{ref}} a.Q \), because there would be no prebisimulation for \( a.P \) and \( a.Q \) (\( Q_3 \) would have no correspondence in \( P! \)). Thus \( \sqsubseteq_{\text{wbs}\text{ref}} \) would not be a congruence with respect to action prefix. That is why we must formulate the definition using \( \downarrow a \) instead of \( \downarrow \).

Next we show that the number of udef-states in a transition system is not essential. By a udef-state in a transition system we mean a partially defined state with no outgoing transitions. Let \( P = (S, A, T, \uparrow, q_0) \) be a transition system and \( s_1, s_2 \in S \). Suppose \( u_1 \) and \( u_2 \) are new partially defined states \( (u_1, u_2 \not\in S) \) such that no transitions start from them. Construct transition systems \( R_1 \) and \( R_2 \) as follows. The system \( R_1 \) is got from \( P \) by adding a transition from \( s_1 \) to \( u_1 \) with an action \( a \) and by adding a transition from \( s_2 \) to \( u_2 \) with an action \( b \). The action \( a \) or \( b \) can also be a silent action \( \tau \). The system \( R_2 \) is constructed by adding a transition from \( s_1 \) to \( u_1 \) with \( a \) and a transition from \( s_2 \) to \( u_1 \) with \( b \).

**Proposition 3.3.1** The system \( R_1 \) is strongly bisimilar with \( R_2 \).

**Proof.** We construct a strong bisimulation \( \mathcal{R} \) between \( R_1 \) and \( R_2 \). The relation \( \mathcal{R} \) consists of pairs \( (q,q), q \in S \), plus the pairs \( (u_1, u_1) \) and \( (u_2, u_1) \). It is seen at once that with this definition \( \mathcal{R} \) is a strong bisimulation. \( \square \)

**Theorem 3.3.2** If \( P, Q, R \in \text{ALTS} \), \( P \sqsubseteq_{\text{wbs}\text{ref}} Q \), \( a \in A \), \( B \subset A \backslash \{\tau\} \) and \( f \) is a renaming function, then
3.3. BISIMULATION REFINEMENT

(a) \( a.P \sqsubseteq_{\text{bisim/ref}} a.Q \),

(b) \( P | B | R \sqsubseteq_{\text{bisim/ref}} Q | B | R \),

(c) \( h_B(P) \sqsubseteq_{\text{bisim/ref}} h_B(Q) \),

(d) \( P[f] \sqsubseteq_{\text{bisim/ref}} Q[f] \).

(e) \( P \gg R \sqsubseteq_{\text{bisim/ref}} Q \gg R \) and \( R \gg P \sqsubseteq_{\text{bisim/ref}} R \gg Q \),

(f) \( P \gg R \sqsubseteq_{\text{bisim/ref}} Q \gg R \).

Proof. The proof resembles a congruence proof in [Walker 90]. We give a detailed proof for b) and c). In the other cases we mention only the prebisimulation relation.

(a) There is a prebisimulation \( \mathcal{R} \) such that \( (P, Q) \in \mathcal{R} \). Now \( \mathcal{R} \cup \{(a.P, a.Q)\} \) is also a prebisimulation.

(b) Let \( \mathcal{R} = \{ (P|B|Q|B|R) \mid P \sqsubseteq_{\text{bisim/ref}} Q \} \). By definition it suffices to prove that \( \mathcal{R} \) is a prebisimulation. Suppose \( P \sqsubseteq_{\text{bisim/ref}} Q \) and assume that \( P|B|R \rightarrow a P'|B|R' \), \( a \in (A \setminus \{\tau\}) \cup \{\varepsilon\} \). We show the first part of the definition of the prebisimulation.

(b1) If \( a \in B \cup \{\delta\} \), then \( a \neq \varepsilon \) and \( P \xrightarrow{a} P' \), \( R \xrightarrow{a} R' \). Because \( P \sqsubseteq_{\text{bisim/ref}} Q \), there is a \( Q' \) such that \( Q \xrightarrow{a} Q' \) and \( P' \sqsubseteq_{\text{bisim/ref}} Q' \). Obviously also \( Q|B|R \rightarrow a Q'|B|R' \) and \( (P'|B|R', Q'|B|R') \in \mathcal{R} \). So in the case \( a \notin B \cup \{\delta\} \) \( \mathcal{R} \) satisfies the first part of the definition of the prebisimulation.

(b2) Suppose \( a \notin B \cup \{\delta\} \). Then either \( P \xrightarrow{a} P' \) and \( R \rightarrow R' \) or \( P \rightarrow P' \) and \( R \xrightarrow{a} R' \). Note that \( a \) can be \( \varepsilon \). If \( P \xrightarrow{a} P' \), then there is a \( Q' \) such that \( Q \xrightarrow{a} Q' \) and \( P' \sqsubseteq_{\text{bisim/ref}} Q' \). It follows that \( (P'|B|R', Q'|B|R') \in \mathcal{R} \). If \( P \rightarrow P' \) and \( R \xrightarrow{a} R' \), then also there is a \( Q' \) such that \( Q \rightarrow Q' \) and \( P' \sqsubseteq_{\text{bisim/ref}} Q' \). In this case \( Q|B|R \rightarrow a Q'|B|R' \), and we have \( (P'|B|R', Q'|B|R') \in \mathcal{R} \). Thus also in the case \( a \notin B \cup \{\delta\} \) \( \mathcal{R} \) satisfies the first part of the definition of the prebisimulation.

We have still to verify condition 2 in the definition of the bisimulation refinement. Suppose that \( (P|B|R) \downarrow a \). If \( a \notin B \cup \{\delta\} \), then \( P \downarrow a \) and \( R \downarrow a \). Because \( P \sqsubseteq_{\text{bisim/ref}} Q \), it follows that also \( Q \downarrow a \). Thus \( (Q|B|R) \downarrow a \). If now \( Q|B|R \xrightarrow{a} Q'|B|R' \), then it is seen in the same way as in b2) that \( P|B|R \xrightarrow{a} P'|B|R' \) and \( P' \sqsubseteq_{\text{bisim/ref}} Q' \) for some \( P' \). It can be concluded that in the case \( a \notin B \cup \{\delta\} \) \( \mathcal{R} \) is a prebisimulation.

If \( (P|B|R) \downarrow \), i.e. \( a = \varepsilon \), then \( P \downarrow \) and \( R \downarrow \) and we can reason in the same way as above that \( \mathcal{R} \) is a prebisimulation.

Suppose \( (P|B|R) \downarrow a \) and \( a \in B \cup \{\delta\} \). At first we know that \( P \downarrow \) and \( R \downarrow \). If \( P \downarrow a \) and \( R \downarrow a \), then we can proceed as above. If \( P \uparrow a \) and \( R \downarrow a \) or \( P \downarrow a \) and \( R \uparrow a \), then there are no transitions \( \xrightarrow{a} \) from \( P|B|R \) because of \( (P|B|R) \downarrow a \). Thus \( P \xrightarrow{a} P' \) or \( R \xrightarrow{a} R' \). If \( P \xrightarrow{a} P' \), then also \( Q \xrightarrow{a} Q' \), because \( P \downarrow a \) and \( P \sqsubseteq_{\text{bisim/ref}} Q \). It follows that there are no transitions \( \xrightarrow{a} \) from \( Q|B|R \) and \( (Q|B|R) \downarrow a \). So \( \mathcal{R} \) is a prebisimulation in all the cases.
c) Let $\mathcal{R} = \{(h_B(P), h_B(Q)) \mid P \sqsubseteq_{\text{ubisref}} Q\}$. We check first that the condition 1 in Definition 3.3.1. holds. Suppose $h_B(P) \Rightarrow P'$. Then there is a $P''$ such that $P'' = h_B(P')$ and $P'' \Rightarrow P'$, where $u = a_1 \cdots a_n$, $a_i \in B$, $i = 1, \cdots, n$, or $u = \varepsilon$. If $h_B(P') \overset{a}{\Rightarrow} P''$, $a \neq \varepsilon$, then there is a $P'$ such that $h_B(P') = P''$ and $P'' \Rightarrow P'$, where $v = a_1 \cdots a_n$, $a = a_i$ and $a_1, \cdots, a_{i-1}, a_{i+1}, \cdots, a_n \in B$. In either case it follows from the definition of $\sqsubseteq_{\text{ubisref}}$, using induction on the length of $u$ or $v$, that there is a $Q'$ such that $Q \overset{u}{\Rightarrow} Q'$ or $Q \overset{v}{\Rightarrow} Q'$ and $P' \sqsubseteq_{\text{ubisref}} Q'$. Thus $h_B(Q) \Rightarrow h_B(Q')$ or $h_B(Q) \overset{a}{\Rightarrow} h_B(Q')$ and $(h_B(P), h_B(Q')) \in \mathcal{R}$.

We further check the condition 2 in Definition 3.3.1. Suppose that $h_B(P) \Downarrow a$, $a \neq \varepsilon$. It follows by the definition of $\Downarrow a$ that $h_B(P) \Downarrow$ and also $P \Downarrow$. Because $P \sqsubseteq_{\text{ubisref}} Q$, we know that $Q \Downarrow$. Hence especially $Q \Downarrow$ and by the definition of $h_B(P)$ also $h_B(Q) \Downarrow$. Assume now $h_B(Q) \Uparrow a$. Then either $h_B(Q) \Uparrow$ or $h_B(Q) \overset{a}{\Rightarrow} h_B(Q')$, $a \neq \varepsilon$, $h_B(Q) \Downarrow$ and $h_B(Q') \Uparrow$.

c1) Assume $h_B(Q) \Uparrow$. Then there is a chain of transitions

$$h_B(Q) \overset{\tau}{\Rightarrow} h_B(Q_1) \overset{\tau}{\Rightarrow} \cdots \overset{\tau}{\Rightarrow} h_B(Q_{k-1}) \overset{\tau}{\Rightarrow} h_B(Q_k),$$

$k \geq 1$, such that $h_B(Q) \Downarrow$, $h_B(Q_i) \Downarrow$, $1 \leq i \leq k-1$, and $h_B(Q_k) \Uparrow$. By the definition of the hiding operator we also have a chain

$$Q \overset{b_1}{\Rightarrow} Q_1 \overset{b_2}{\Rightarrow} \cdots \overset{b_k}{\Rightarrow} Q_k,$$

where $b_i \in B \cup \{\tau\}$ for all $i$, $1 \leq i \leq k$, and $Q \Downarrow$, $Q_i \Downarrow$, $1 \leq i \leq k-1$, $Q_k \Uparrow$. If $P \Uparrow b_1$, then by the assumption $P \Downarrow$ we know $b_1 \neq \tau$. Now the only possibility is $P \overset{b_1}{\Rightarrow} P'$ and $P' \Uparrow$ for some $P'$. Thus $h_B(P) \Rightarrow h_B(P')$ and $h_B(P') \Uparrow$. In other words, $h_B(P) \Uparrow$, which is a contradiction. So it has to be that $P \Downarrow b_1$. Because $P \Downarrow b_1$ and $P \sqsubseteq_{\text{ubisref}} Q$, there is a $P_1$ such that $P \overset{b_1}{\Rightarrow} P_1$ and $P_1 \sqsubseteq_{\text{ubisref}} Q_1$.

Consider now the transition $Q_1 \overset{b_2}{\Rightarrow} Q_2$. Because $P \Downarrow b_1$, we know that $P_1 \Downarrow$. Now exactly in the same way as in the case of $P$ and $b_1$ we can conclude that there is a $P_2$ such that $P_1 \overset{b_2}{\Rightarrow} P_2$ and $P_2 \sqsubseteq_{\text{ubisref}} Q_2$. Continuing in this way we get a chain of transitions

$$P \overset{b_1}{\Rightarrow} P_1 \overset{b_2}{\Rightarrow} \cdots \overset{b_k}{\Rightarrow} P_k,$$

where $P_i \sqsubseteq_{\text{ubisref}} Q_i$, $1 \leq i \leq k$, and $P_i \Downarrow$, $1 \leq i \leq k$. But now we must have $P_k \Uparrow$, because $Q_k \Uparrow$. Hence also $h_B(P) \Uparrow$, which is a contradiction. Thus $h_B(Q) \Downarrow$.

c2) Assume that $h_B(Q) \overset{a}{\Rightarrow} h_B(Q')$, $a \neq \varepsilon$, $h_B(Q) \Downarrow$ but $h_B(Q') \Uparrow$. Then there is a chain of transitions

$$Q = Q_0 \overset{b_1}{\Rightarrow} Q_1 \overset{b_2}{\Rightarrow} \cdots \overset{b_k}{\Rightarrow} Q_k \overset{a}{\Rightarrow} Q_{k+1} \overset{c_{k+1}}{\Rightarrow} Q_{k+2} \overset{c_{k+2}}{\Rightarrow} \cdots \overset{c_{k+m-1}}{\Rightarrow} Q_{k+m},$$

where $k \geq 0$, $m \geq 1$, $b_i \in B \cup \{\tau\}$ for all $i$, $1 \leq i \leq k$, $c_j \in B \cup \{\tau\}$ for all $j$, $k+1 \leq j \leq k+m-1$, $Q_i \Downarrow$, $0 \leq t \leq k + m - 1$, $Q_{k+m} \Uparrow$. As in c1) it can be
shown that for every $Q_t$, $0 \leq t \leq k + m$, there is a $P_t$ such that $P_t \sqsubseteq_{\text{usiref}} Q_t$ and

$$P = P_0 \xrightarrow{b_1} P_1 \xrightarrow{b_2} \cdots \xrightarrow{b_k} P_k \xrightarrow{a} P_{k+1} \xrightarrow{c_{k+1}} P_{k+2} \xrightarrow{c_{k+2}} \cdots \xrightarrow{c_{k+m-1}} P_{k+m}.$$  

It follows from this that $h_B(P) \xrightarrow{\alpha} h_B(P_{k+m})$. Because $h_B(P) \Downarrow a$, we have furthermore that $h_B(P_t) \Downarrow$ for all $0 \leq t \leq k + m$. Because $h_B(P_{k+m}) \Downarrow$ and $P_{k+m} \sqsubseteq_{\text{usiref}} Q_{k+m}$, we have a contradiction with the antithesis $Q_{k+m} \Uparrow$. Thus it should be $h_B(Q) \Downarrow a$.

It remains to check the case $h_B(P) \Downarrow$. This is in fact the same situation as in the case c1).

The last part is to show that $h_B(Q) \xrightarrow{\alpha} h_B(Q')$ implies there is a $P'$ such that $h_B(P) \xrightarrow{\alpha} h_B(P')$ and $h_B(P') \sqsubseteq_{\text{usiref}} h_B(Q')$. This follows quite directly from the facts $P \sqsubseteq_{\text{usiref}} Q$ and $P \Downarrow$.

d) Let $f : A \setminus \{\tau\} \to A \setminus \{\tau\}$ be a renaming function. Now

$$\mathcal{R} = \{(P[f], Q[f]) \mid P \sqsubseteq_{\text{usiref}} Q\}$$

is a prebisimulation.

e) Define the prebisimulation by

$$\mathcal{R} = \{(P >> R, Q >> R) \mid P \sqsubseteq_{\text{usiref}} Q\} \cup \{(R, R) \mid R \in \text{ALTS}\}$$

or by

$$S = \{(R >> P, R >> Q) \mid P \sqsubseteq_{\text{usiref}} Q\} \cup \{(P, Q) \mid P \sqsubseteq_{\text{usiref}} Q\}$$

f) Let

$$\mathcal{R} = \{(P[> R, Q[> R) \mid P \sqsubseteq_{\text{usiref}} Q\} \cup \{(R', R) \mid R' \in \text{ALTS}\}.$$

The relation $\sqsubseteq_{\text{usiref}}$ is not compositional with respect to the choice operator $[\cdot]$. For example, let $P$, $Q$ and $R$ be the processes

$$P : \to P1 \xrightarrow{\tau} P2, \quad Q : \to Q1, \quad R : \to R1 \xrightarrow{\alpha} R2.$$

Then $P \sqsubseteq_{\text{usiref}} Q$, but $P[\cdot R \sqsubseteq_{\text{usiref}} Q[\cdot R$. These processes can also be used to show that it is not possible to change the order of the processes in the previous theorem in f), because $R[>] P \sqsubseteq_{\text{usiref}} R[>] Q$. These deficiencies have some significance in practice, although the most important operators in verifications are $h_B$ and $|B|$. The preorder $\sqsubseteq_{\text{usiref}}$ has another imperfection, which is shown in the next theorem in d). The theory would be more complete and symmetric if the equivalence induced by the refinement $\sqsubseteq_{\text{usiref}}$ would agree with $\approx_{\text{us}},$ but this is not the case. The statements a), b) and c) show, however, that $\sqsubseteq_{\text{usiref}}$ is not without merits.
**Proposition 3.3.3** Let \( \approx_{\text{wbisref}} \) be the equivalence \( \subseteq_{\text{wbisref}} \cap \subseteq_{\text{wbisref}}^{-1} \).

a) The preorder \( \subseteq_{\text{wbisref}} \) agrees with the equivalence \( \approx_{\text{wbis}} \) on totally defined processes.

b) If \( P \) is totally defined and \( P \subseteq_{\text{wbisref}} Q \), then \( Q \) is totally defined and \( P \approx_{\text{wbis}} Q \).

c) If \( P \approx_{\text{wbis}} Q \), then \( P \approx_{\text{wbisref}} Q \).

d) \( \approx_{\text{wbis}} \subseteq \approx_{\text{wbisref}} \) strictly.

**Proof.** a) and b) are obvious by the definitions of \( \approx_{\text{wbis}} \) and \( \subseteq_{\text{wbisref}} \).

c) Suppose \( P \approx_{\text{wbis}} Q \). Let \( R \) be a bisimulation such that \( (P, Q) \in R \). Then \( R \) is also a prebisimulation. Namely, the definition of bisimulation implies the first condition of prebisimulation. If \( P \Downarrow a \), then \( P \Downarrow \) and because \( P \approx_{\text{wbis}} Q \), also \( Q \Downarrow \). If \( Q \Rightarrow Q' \), then there is a \( P' \) such that \( P \Rightarrow P' \) and \( P' \approx_{\text{wbis}} Q' \). Now \( Q' \Downarrow \) implies \( P' \Downarrow \), a contradiction. Thus \( Q \Downarrow a \) and the second condition of prebisimulation follows. Hence \( P \subseteq_{\text{wbisref}} Q \). In a similar way we can show that \( R^{-1} \) is a prebisimulation and so \( Q \subseteq_{\text{wbisref}} P \). Thus \( P \approx_{\text{wbisref}} Q \).

d) Let \( P \) and \( Q \) be the processes

\[
\begin{array}{c}
\rightarrow P1 \uparrow \overset{a}{\rightarrow} P3 \uparrow, \quad \rightarrow Q1 \uparrow \overset{a}{\rightarrow} Q2.
\end{array}
\]

Then \( P \subseteq_{\text{wbisref}} Q \), because \( \{(P1, Q1), (P2, Q2), (P3, Q2)\} \) is a prebisimulation. Also \( Q \subseteq_{\text{wbisref}} P \), because \( \{(Q1, P1), (Q2, P2)\} \) is a prebisimulation. But \( P \not\approx_{\text{wbis}} Q \), because \( P3 \uparrow \) and \( Q2 \Downarrow \).

\( \Box \)

The preorder \( \subseteq_{\text{wbisref}} \) preserves the same properties as the equivalence \( \approx_{\text{wbis}} \) in the following sense. Let \( P \) be a process and \( P' \) a state of \( P \), which as a process is totally defined. If \( P \subseteq_{\text{wbisref}} Q \), then there is a state \( Q' \) of \( Q \) such that \( P' \approx_{\text{wbis}} Q' \). To see this, let \( R \) be a prebisimulation such that \( (P, Q) \in R \). It follows from the definition of the prebisimulation that there is a \( Q' \) such that \( (P', Q') \in R \). In other words, \( P' \subseteq_{\text{wbisref}} Q' \). But \( P' \) is totally defined, so by the 3.3.3 \( Q' \) also is totally defined and thus \( P' \approx_{\text{wbis}} Q' \).

### 3.4 Partial bisimulation refinement

For comparison we define another bisimulation preorder ([Cleaveland and Steffen 90]).

**Definition 3.4.1** A relation \( R \subseteq \text{ALTS} \times \text{ALTS} \) is a partial bisimulation, if for all \( (P, Q) \in R \) and for all \( a \in (A \setminus \{\tau\}) \cup \{\varepsilon\} \) such that \( P \Downarrow a \) the following holds:
3.4. PARTIAL BISIMULATION REFINEMENT

1. \( Q \downarrow a; \)
2. if \( P \xrightarrow{a} P', \) then there is \( Q' \) such that \( Q \xrightarrow{a} Q' \) and \( (P', Q') \in \mathcal{R}; \)
3. if \( Q \xrightarrow{a} Q', \) then there is \( P' \) such that \( P \xrightarrow{a} P' \) and \( (P', Q') \in \mathcal{R}. \)

The partial bisimulation refinement \( \sqsubseteq_{\text{bisref}} \) is defined to be the relation

\[
\sqsubseteq_{\text{bisref}} = \bigcup \{ \mathcal{R} \mid \mathcal{R} \text{ is a partial bisimulation} \}.
\]

In the same way as in the case of the bisimulation refinement it is seen that \( \sqsubseteq_{\text{bisref}} \) is a preorder. The theorem 3.3.2 is also valid in the case of the partial bisimulation refinement. Let \( \approx_{\text{bisref}} \) be the equivalence \( \sqsubseteq_{\text{bisref}} \cap \sqsubseteq_{\text{bisref}}^{-1} \). The next proposition shows the most important properties of \( \sqsubseteq_{\text{bisref}} \) and \( \approx_{\text{bisref}}. \)

**Proposition 3.4.1**

a) The partial bisimulation refinement \( \sqsubseteq_{\text{bisref}} \) agrees with the equivalence \( \approx_{\text{bis}} \) on totally defined systems.

b) If \( P \sqsubseteq_{\text{bisref}} Q, \) then \( P \sqsubseteq_{\text{bisref}} Q. \) There are processes \( P \) and \( Q \) such that \( P \sqsubseteq_{\text{bisref}} Q \) but \( P \not\sqsubseteq_{\text{bisref}} Q. \)

c) We have that \( \approx_{\text{bisref}} \subset \approx_{\text{bisref}} \) properly and \( \approx_{\text{bis}} \subset \approx_{\text{bisref}} \) properly.

**Proof.**

a) This is obvious from the definitions.

b) The first claim is readily seen from the definitions. The second claim is also obvious, because if the refinement relation is \( \sqsubseteq_{\text{bisref}} \) and if the initial states are partially defined, the inner structures of the related processes can be arbitrary. For example, let \( P \) and \( Q \) be the processes

\[ P1 \uparrow \xrightarrow{a} P2, \quad Q1 \uparrow. \]

Then \( P \sqsubseteq_{\text{bisref}} Q, \) but \( P \not\sqsubseteq_{\text{bisref}} Q. \)

c) The processes in the previous example are such that \( P \sqsubseteq_{\text{bisref}} Q \) and \( Q \sqsubseteq_{\text{bisref}} P. \) Consequently \( P \approx_{\text{bisref}} Q, \) but \( \not\approx_{\text{bis}} Q \) and \( \not\approx_{\text{bisref}} Q. \) If \( P \approx_{\text{bisref}} Q, \) then \( P \sqsubseteq_{\text{bisref}} Q \) and \( Q \sqsubseteq_{\text{bisref}} P. \) It follows from b) that also \( P \sqsubseteq_{\text{bisref}} Q \) and \( Q \sqsubseteq_{\text{bisref}} P. \) From this and proposition 3.3.3 we get that \( \approx_{\text{bis}} \subset \approx_{\text{bisref}} \subset \approx_{\text{bisref}}. \)

\[ \square \]

The partial bisimulation refinement does not preserve properties as accurately as the bisimulation refinement. Let \( P \) be a process and \( P' \) a state in \( P. \) Suppose there is a path from \( P \) to \( P', \) where every state including \( P \) and \( P' \) are totally defined. If \( P \sqsubseteq_{\text{bisref}} Q, \) then there is a state \( Q' \) in \( Q \) such that \( P' \sqsubseteq_{\text{bisref}} Q'. \) If \( P' \) as a process is totally defined, then so is \( Q' \) and \( P' \approx_{\text{bis}} Q'. \) Note that with the bisimulation refinement, in order to get the same result, it is not necessary that the path consists of totally defined states.
There is a different philosophy behind the two preorders. As for the bisimulation refinement, one can think that in the case $P \sqsubseteq_{\text{ubiref}} Q$ the more specified process $Q$ contains a copy of a process equivalent to $P$. It is possible to add transitions to partially defined states and maintain the relation. In the case of the partial bisimulation refinement the larger process contains a copy of an equivalent process to the smaller one only up to partially defined states. It is left out of consideration what happens in partially defined states.

It seems that bisimulation refinement corresponds more accurately with the idea of stepwise refinement than partial bisimulation refinement. But the latter also has merits, especially in some theoretical considerations (see the concept "adequacy" in [Cleaveland and Steffen 90]).

3.5 Other bisimulation based refinement relations

We have many other technical and conceptual possibilities to define refinement relations. For instance, we could define

$$P \sqstile{a} \text{ if and only if } \exists P' : \quad P \xrightarrow{a} P' \quad \text{and } P' \uparrow$$

and use this relation instead of $P \uparrow a$. This technical change does not influence the definition of the weak bisimulation equivalence, but the resulting preorder is coarser than the one we have defined.

It would be more far-reaching to change the way the partiality is used. If a state is partially defined, then according to the present definitions we don't know which transitions are missing, just that something is missing. In practice, however, we often know the transitions that have been left out. It would be reasonable to equip every partially defined state with a set of those labels that occur in the missing transitions and define a preorder according to this scheme. The resulting equivalence and preorder would be tighter than the present ones. I feel that it would be worthwhile to study these variations, too, but it is desirable to clarify the basic questions concerning simpler relations first. Moreover, full Lotos presents certain complications, if data and its refinements are considered and I am interested to examine how far in this direction one can proceed. For this purpose basic equivalences and refinement relations are sufficient.

In addition, there are other options; see [Larsen and Thomsen 88] and [Cleaveland and Steffen 90] for further remarks.
Chapter 4

Applications

In this chapter, we present three applications for partially defined systems to motivate the concepts and results developed so far. The first application is compositional verification. When done without a detailed analysis of the case, compositional verification can in some cases lead to larger state explosion than the straightforward construction of the system as a whole. A partially defined system is one method to encode the results of the analysis in order to avoid unnecessary states in intermediate minimization phases.

The second and main application is stepwise design of a provably correct protocol. A traditional way to develop correct distributed algorithms or protocols is first to write complete formal specifications for the protocol and the service of the protocol. After this, designers try to show these two specifications equivalent with respect to some equivalence relation. Usually the first attempts fail because of syntactic errors, minor or major design errors in the specifications or errors in integrating subsystems. These errors, although not severe in principle, can be very difficult to detect in practice. A more designer-friendly approach is to specify only one subsystem at a time, integrate it with the existing skeleton and to verify that the partially defined system can be completed to a correct completely defined system. As the words chosen to describe the approach suggest, the above method can be formalised with the help of partially defined systems and their refinement relations.

As the third application we show how partially defined processes can be applied for distributed detection of deadlocks. The reachability graphs of real systems are often too large to be examined completely in one computer and in one run. By cutting a system to several partially defined subsystems it is possible to analyse the subsystems separately either in several computers at the same time or one after another in one computer.

All applications to be presented now are small in order to illustrate the basic ideas. After developing further concepts we later present a larger case study to better illustrate the stepwise refinement of protocols. Our first application is from the article [Larsen and Thomsen 88]. The applications for stepwise refinements are based on [Celikkan and Cleaveland 95], whereas the distributed detection of
deadlocks is new to our knowledge.

4.1 Compositional verification, a conceptual example

Let \( P, Q, R, S \) and \( V \) be the following processes:

\[
\begin{align*}
& \quad \text{S} \\
\rightarrow & P_1 \xrightarrow{a} P_2 \xrightarrow{p} P_3 \\
\rightarrow & Q_1 \xrightarrow{p} Q_2 \xrightarrow{b} Q_3 \\
\rightarrow & R_1 \xrightarrow{q} R_2 \xrightarrow{c} R_3 \\
\rightarrow & S_1 \xrightarrow{r} S_2 \xrightarrow{d} S_3 \\
\rightarrow & V_1 \xrightarrow{a} V_2 \xrightarrow{b} V_3 \xrightarrow{c} V_4
\end{align*}
\]

Let \( B = \{ p, q, r, s \} \) and define

\[ T = \text{hide } B \text{ in } ((P|\{p\}|Q)|\{q\}|R)|\{r, s\}|S. \] (1)

We want to show that \( T \) is weakly bisimilar with the process \( V \). First notice that we can apply the property 4 on the page 16 to get

\[ T = \text{hide } B \text{ in } (P|\{p\}|Q)|\{q, s\}|(R|\{r\}|S). \] (2)

To proceed compositionally we decompose \( T \) into two processes:

\[ T_1 = \text{hide } \{p\} \text{ in } P|\{p\}|Q, \]
\[ T_2 = \text{hide } \{r\} \text{ in } R|\{r\}|S. \]
4.1. COMPOSITIONAL VERIFICATION, A CONCEPTUAL EXAMPLE

Now
\[ T = \text{hide} \{q, s\} \text{ in } T_1 \{q, s\} | T_2. \] (3)

The processes \( T_1 \) and \( T_2 \) are unnecessarily large, because they contain transitions, which are not needed in the final combination (3). As we shall see later, we can detect the unnecessary transitions by analyzing the order of actions. By analyzing first the individual processes we get the following orders, obtained by examining the cycles from the initial state to the initial state:

\[
P: \ a < p < s, \\
Q: \ p < b < q, \\
R: \ q < c < r, \\
S: \ r < d < s.
\]

The notation \( a < p \) means that \( a \) is always executed before \( p \) in \( P \). In this example it is very easy to get clear precedence relations between all the actions in one process, because the processes contain neither branches nor inner cycles. In more complicated systems it is more difficult to get precedence relations between all the actions in a process. In this kind of situations one has to consider a subset of all actions and restrict the precedence relation between the actions in this subset.

In our example the action \( p \) is a synchronizing action between \( P \) and \( Q \) and hence we can combine the precedence relations of \( P \) and \( Q \) to get a precedence relation for \( T_1 \). In the same way \( r \) is a synchronizing action between \( R \) and \( S \) and we get the precedence relation for \( T_2 \):

\[
T_1: \begin{cases} 
a < s \\
b < q \\
a < b 
\end{cases} \quad T_2: \begin{cases} 
q < c \\
c < d \\
d < s 
\end{cases}
\]

We have left the actions \( p \) and \( r \) out of the relations because they are invisible in \( T_1 \) and \( T_2 \).

Furthermore, \( q \) and \( s \) are synchronizing actions and with the help of these actions we can combine the precedence relations of \( T_1 \) and \( T_2 \) to get the global precedence relation of \( T \):

\[ a < b < q < c < d < s. \] (4)

The next step is to construct \( T_1 \) in such a way that the ordering of actions does not violate the global order. If there is a transition in \( T_1 \) which is not in accordance with the relation (4), then we can leave that transition out, because it is not needed in the final composition. We get the process \( T_1^* \) (figure 4.1). The process \( T_1^* \) contains 5 totally defined states and 5 transitions (altogether 7 states and 7 transitions). The totally defined states are essential in the sense that they are needed in the transition system of \( T \). The partially defined states are not needed in the transition system of \( T \). Without restrictions \( T_1 \) would contain 9 states and 13 transitions. Thus \( T_1^* \) is properly simpler than \( T_1 \).
The first two transitions by $s$ have happened too early in the process $T_1$ and consequently continuations can be dropped. The result is a partially defined process. This situation also clarifies a minor technical detail. We can't leave the $s$-transitions out and mark the states $P3Q2$ and $P3Q3$ partially defined, because then we would also have partially defined states in the final composition $T$. We must include the first unnecessary transitions and stop only after this.

We can proceed in the same way in the case of $T_2$. In addition to the precedence relation (4) we also use the process $T_2^*$ with its precedence relations. In the relation (4) the actions $q$ and $s$ are synchronizing relations between $T_1$ and $T_2$. The action $q$ can occur in $T_2$ for the second time only after the action $s$ has occurred for the first time in $T_1$. It follows that $s$ must occur also in $T_2$ before $q$ has occurred for the second time. We get the process $T_2^*$ representing $T_2$ (figure 4.2). Without restrictions $T_2$ would be of the same size as $T_1$.

We see from the definition of $\sqsubseteq_{bisref}$ that

$$T_1^* \sqsubseteq_{bisref} T_1, \quad T_2^* \sqsubseteq_{bisref} T_2,$$

because $T_1^*$ is a subgraph of $T_1$ and $T_2^*$ is a subgraph of $T_2$ and the partially defined states are situated in the cut off points. Parallel composition and hiding are compositional with respect to the bisimulation refinement and thus

$$\text{hide } \{q, s\} \text{ in } (T_1^*|\{q, s\} T_2^*) \sqsubseteq_{bisref} \text{ hide } \{q, s\} \text{ in } (T_1|\{q, s\}|T_2) = T.$$
We still have to show that

\[ V \sqsubseteq_{\text{uf}_{\text{hid}}} \{q, s\} \text{ in } (T_1^s \{q, s\} | T_2^s). \]

The states $PQ1$ and $PQ2$ in $T_1^s$ are $\approx_{\text{uf}_{\text{hid}}}$-equivalent with each other as well as the states $R3S1$ and $R1S2$ in $T_2^s$. Let $T_{1m}^s$ and $T_{2m}^s$ be the minimizations of $T_1^s$ and $T_2^s$, respectively. Because $T_{1m}^s \approx_{\text{uf}_{\text{hid}}} T_1^s$ and $T_{2m}^s \approx_{\text{uf}_{\text{hid}}} T_2^s$, by the proposition 3.3.2 c) and the transitivity of a preorder we still have

\[ T_{1m}^s \sqsubseteq_{\text{uf}_{\text{hid}}} T_1, \quad T_{2m}^s \sqsubseteq_{\text{uf}_{\text{hid}}} T_2. \]

Now it is straightforward to construct $W = \text{hide } \{q, s\} \text{ in } (T_{1m}^s \{q, s\} | T_{2m}^s)$:
Without restrictions \( W \) (i.e. \( T \)) would be two times larger, so we have acquired a 50 per cent reduction. The result \( W \) is \( \approx_{wbi} \)-equivalent to \( V \). The refinement bisimulation is transitive, hence

\[
V \sqsubseteq_{wbi ref} W \sqsubseteq_{wbi ref} T
\]

implies that

\[
V \sqsubseteq_{wbi ref} T.
\]

Both \( V \) and \( T \) are totally defined and thus

\[
V \approx_{wbi} T \approx_{wbi} W.
\]

It may seem that it is very difficult, especially in larger specifications, to find precedence relations. However, if we had software tools to test the equivalence and refinement relations, it would not be necessary to be sure in advance that precedences are correct. If the guesses are wrong, then there are partially defined states in the final parallel composition and we cannot prove our claims, because some of our relations are not valid and the verification tool points that out. After this we can make new assumptions and proceed similarly.

### 4.2 Stepwise design of protocols

We present two examples. The first one shows how Alternating Bit Protocol or AB-protocol can be designed stepwise and how wrong design decisions can be avoided at an early stage. The second example represents the detection of erroneous basic ideas in constructing a data communication protocol.

The AB-protocol consists of a sender and a receiver connected with a half-duplex channel. The sender takes data messages \( s \) from a user and sends them to the
receiver. The receiver gives the received messages \( r \) to another user. It is understood that the receiver gives every \( s \) only once to the user. The service description of the protocol is then very simple:

\[
\text{ABservice} := s; r; \text{ABservice}.
\]

The channel can drop messages. In principle, it can also distort packets, but we think that erroneous messages are not delivered further, i.e. they are discarded before they are given to the sender or receiver. In order to ensure that every message goes to the aimed destination, \( s \)-messages are given sequence numbers 0 or 1. We denote by \( d_0 \) an \( s \)-message with 0 and by \( d_1 \) an \( s \)-message with 1. The receiver acknowledges the received \( d_0 \) and \( d_1 \) by \( a_0 \) and \( a_1 \), respectively. In the specification of the channel we must make a distinction between incoming \( d_0 \) and \( d_1 \) and delivered \( d_0 \) and \( d_1 \). That’s why we denote by \( dd_0 \) and \( dd_1 \) the incoming messages from the sender. Respectively, \( aa_0 \) and \( aa_1 \) are the incoming acknowledgements from the receiver. The channel \( C \) can be represented as

\[
C := C_1 [:] C_2,
\]

where

\[
C_1 := dd_0; (d_0; C_1 [ i ]; C_1) [ i ]; dd_1; (d_1; C_1 [ i ]; C_1)
\]

and

\[
C_2 := aa_0; (a_0; C_2 [ i ]; C_2) [ i ]; aa_1; (a_1; C_2 [ i ]; C_2).
\]

Now we can start to specify the AB-protocol itself. The complete system is

\[
AB := \text{hide } A \text{ in } (S|\mid R) \mid B \mid C,
\]

where \( S \) is the sender, \( R \) is the receiver, \( A \) is the set containing all the actions in the system with the exception of \( s \) and \( r \) and

\[
B = \{d_0, dd_0, d_1, dd_1, a_0, aa_0, a_1, aa_1\}.
\]

Let the first designs for \( S \) and \( R \) be:

\[
S_1 : \quad \rightarrow S1 \uparrow \xrightarrow{s} S2 \uparrow \xrightarrow{dd_0} S3 \uparrow
\]

\[
R_1 : \quad \rightarrow R1 \uparrow \xrightarrow{d_0} R2 \uparrow
\]

Now

\[
\text{hide } A \text{ in } (S_1|\mid R_1) \mid B \mid C \subseteq_{\text{bisim}} \text{ABservice}, \tag{4.1}
\]

so \( S_1 \) and \( R_1 \) are correct starting points. Next we refine

\[
S_2 : \quad S1 \uparrow \xrightarrow{s} S2 \uparrow \xrightarrow{dd_0} S3 \uparrow \xrightarrow{a_0} S4 \uparrow \xrightarrow{s} S5 \uparrow
\]

\[
R_2 : \quad R1 \uparrow \xrightarrow{d_0} R2 \uparrow \xrightarrow{aa_0} R3 \uparrow \xrightarrow{r} R4 \uparrow.
\]
But now

\[ \text{hide } A \text{ in } (S_2 \| R_2) \overset{a}{\|} B \overset{\text{wbsreb}}{\mid} C \overset{\text{Abserf}}{\mid} \text{ABservice}, \]

so we have made a mistake and \( S_2 \) and \( R_2 \) cannot be completed to correct processes \( S \) and \( R \). We can analyse the reason for the mistake, if we draw the transition graph of the whole, still partially defined, system.

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ AB1 \]

\[ \]

\[ AB2 \]

\[ \]

\[ AB3 \]

\[ \]

\[ AB4 \]

\[ \]

\[ AB5 \]

\[ \]

\[ AB6 \]

\[ \]

\[ AB7 \]

\[ \]

\[ AB8 \]

\[ \]

\[ AB9 \]

\[ \]

\[ AB10 \]

\[ \]

We see that \( s \) can occur two times before \( r \) occurs. The reason for this is that \( R_2 \) acknowledges \( a0 \) before delivering \( r \) to the user. If we modify \( R_2 \) as follows

\[ R_2 : \quad \rightarrow R1 \overset{d0}{\rightarrow} R2 \overset{r}{\rightarrow} R3 \overset{a0\circ}{\rightarrow} R4 \overset{a0}{\rightarrow}, \]

the anomaly disappears and we can continue to develop \( R_2 \) and \( S_2 \) into \( R_3 \) and \( S_3 \), respectively.

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

Now the equation 4.1 for \( S_3 \) and \( R_3 \) is also valid, so \( S_3 \) and \( R_3 \) describe the behaviour correctly in a case where no transmission errors occur. Notice that the channels are totally defined and they can corrupt messages. Because the sender and receiver do not consider error situations, errors in channels lead to partially defined deadlocked states. This means that errors are handled in those parts of the sender and receiver that will be specified in the following steps.

Next we add error routines. First we remove the partial state symbol \( \uparrow \) from the states \( S1, S2, S4, S5 \) and \( R2, R3, R5, R6 \) and test that the states \( S4 \) and \( R4 \) so acquired
processes satisfy the equation 4.1. Because they satisfy it, it is necessary to add more actions only to the states $S_3$, $S_6$ and $R_1$, $R_4$. The results are:

$$
\begin{align*}
\downarrow & \quad S_1 \xrightarrow{a_1} S_6 & \downarrow & \quad R_1 \xrightarrow{d_1} R_6 \\
\quad & \xrightarrow{s} S_2 & \xrightarrow{t \ x_1} \quad S_5 & \xrightarrow{d_0} R_2 \xrightarrow{r} R_5 \\
\quad & \xrightarrow{t \ x_0 \ x_1} S_3 \xrightarrow{a_0} S_4 & \xrightarrow{r} & \quad R_3 \xrightarrow{a_0 \ x_2} R_4
\end{align*}
$$

Now the whole system, $AB$, is totally defined and

$$AB \approx_{w_{bis}} AB_{service},$$

so we have completed the design.

Next we present a warning example. In it we try to design a similar protocol, which is based on erroneous ideas ([Lynch 68]). In spite of using refinement relations we do not detect the error at an early stage. For simplicity we assume that the communication between the sender and receiver is synchronous. Messages between the protocol entities can be corrupted but they never vanish. The basic idea in the protocol is to send negative acknowledgements to erroneous messages. The service the protocol produces is the same as in the AB-protocol, but now we simply call it Service. The basic loops in the sender and receiver are as follows:

\[ \begin{array}{c}
\text{S:} \\
\xrightarrow{a} S_1 \xrightarrow{s} \quad \xrightarrow{a_0 \ x_2} S_3
\end{array} \]

\[ \begin{array}{c}
\text{R:} \\
\xrightarrow{r} R_1 \xrightarrow{da \ x_1} \quad R_3
\end{array} \]

After getting the message from a user the sender sends it with a positive acknowledgement (da). After receiving a correct message the receiver delivers it to a user and sends a positive acknowledgement. We have marked "undefined" those states which must handle the erroneous messages.

Let $B$ be the messages common to $S$ and $R$. Because

$$\text{hide } B \text{ in } S|B|R \subseteq_{w_{isref}} \text{ Service},$$

(4.2)

the partial specifications $S$ and $R$ can, in principle, be developed to correct totally defined processes. Next we add the case of a corrupted acknowledgement to the protocol:
Still, the equation 4.2 is valid and we can continue by taking corrupted data messages into account. Let us first consider the case where $da$ can be corrupted:

The equation 4.2 is valid and we continue by including the case of corrupted $n$ into the system. This can be done by adding the arc $R4 \xrightarrow{e} R1$ and marking $R4$ completely defined. But now the equation 4.2 is not valid any more. We can try to remedy the situation by a different strategy for $n$: 
With these modifications the equation 4.2 is again valid. The only remaining
detail is the distortion of $dn$ in $S$. But if we add this possibility to $S$ and then
consider the specifications complete, the protocol does not produce Service. After
minimizing the system it is possible to see the reason for the failure: $R$ can deliver
the same $r$ more than once to the user. And it is impossible to correct $R$ and $S$
without making fundamental changes to the protocol. This example shows that
sometimes a 'correct' partial specification can not be completed to a reasonable
algorithm. In fact, this is easily understandable, because we can always first specify
the behaviour without error cases, and after this, one can try to add incorrect error
routines to the specification. This will, of course, fail.

It should be noted that in the article [Celikkan and Cleaveland 95] the authors
also present a diagnostic algorithm, which helps in locating a failure.

### 4.3 Distributed detection of deadlocks

Let $P \in \text{ALTS}$ be a totally defined process. A **partially defined subsystem** or **sub-
process** of $P$ is a connected subgraph $Q$ of $P$ such that

- the initial state of $P$ is the initial state of $Q$;
- every state in $Q$ is reachable from the initial state of $P$;
- if a state $s$ is in $Q$, but $Q$ does not contain every transition from $s$ in $P$, then
  $s$ is partially defined in $Q$.

Let $Q_1$ and $Q_2$ be two subprocesses of $P$. Assume that transition systems $Q_1$
and $Q_2$ are

$$Q_1 = (S_1, A_1, T_1, s_0, \uparrow), \quad Q_2 = (S_2, A_2, T_2, s_0, \uparrow).$$

Define the **process union** of $Q_1$ and $Q_2$ to be the transition system

$$Q_1 \uplus Q_2 = (S, A, T, s_0, \uparrow),$$

where

- $S = S_1 \cup S_2$,
- $A = A_1 \cup A_2$,
- $T = T_1 \cup T_2$,
- if $s \in S_1 \cap S_2$ and $s \uparrow$ in $Q_1$ and $Q_2$, then $s \uparrow$ in $Q_1 \uplus Q_2$, otherwise $s \downarrow$; if $s \in S \setminus (S_1 \cap S_2)$, then $s \uparrow$ if and only if $s \uparrow$ in $Q_1$ or in $Q_2$, otherwise $s \downarrow$.

A collection $B = \{P_i | i \in I\}$ of partially defined processes is a **cover** for a totally
defined process $P$, if
\[ P_i \] is a subprocess of \( P \) for all \( i \in I \);

\[ \biguplus_{i \in I} P_i = P. \]

By the definitions of cover and process union, a completely defined state in \( P \) is completely defined at least in one of the cover processes.

**Proposition 4.3.1** Let \( B_R = \{ R_i | i \in I \} \) be a cover for \( R \), \( B_S = \{ S_j | j \in J \} \) a cover for \( S \) and \( B \) a set of actions. Then

\[
\biguplus_{i \in I} R_i | B | S_j \sqsupseteq_{\text{ubisref}} R | B | S.
\]

**Proof.** By the definition of a cover, \( R_i \sqsubseteq_{\text{ubisref}} R \) and \( S_j \sqsubseteq_{\text{ubisref}} S \) for every \( i \in I \) and \( j \in J \). Because the parallel composition is compositional with respect to \( \sqsubseteq_{\text{ubisref}} \), we have \( R_i | B | S_j \sqsubseteq_{\text{ubisref}} R | B | S \) for every \( i \in I \), \( j \in J \). As a matter of fact, \( R_i | B | S_j \) is a subprocess of \( R | B | S \). By the definition of the process union, \((R_1 | B | S_j_1) \uplus (R_2 | B | S_j_2)\) is also a subprocess of \( R | B | S \). \( \square \)

The ideal situation would be such that \( \biguplus_{i \in I, j \in J} R_i | B | S_j = R | B | S \), i.e. the process union \( \biguplus_{i \in I, j \in J} R_i | B | S_j \) would be a cover for \( R | B | S \). Unfortunately, the construction of a cover is not compositional with respect to the parallel operator. It may occur that \( \biguplus_{i \in I, j \in J} R_i | B | S_j \not\sqsupseteq_{\text{ubis}} R | B | S \) as the following example shows.

Let \( R \) and \( S \) be the following processes:

\[
\begin{align*}
\rightarrow & R_1 \xrightarrow{b} R_3 \quad \rightarrow & S_1 \xrightarrow{b} S_3 \\
\quad & R_2 \xrightarrow{e} R_3 \quad & S_2 \xrightarrow{e} S_4 \quad & S_3 \xrightarrow{a} S_5
\end{align*}
\]

Let a cover for \( R \) consist of the processes \( R_1 \) and \( R_2 \):

\[
\begin{align*}
\downarrow & R_1 \xrightarrow{b} R_3 & \downarrow & R_1 \xrightarrow{a} R_3 \\
\quad & R_2 \xrightarrow{c} R_3 & \quad & R_2 \xrightarrow{d} R_3
\end{align*}
\]

The process \( S \) itself is a cover for \( S \). Let \( B = \{ a, b, c, d \} \) and let the processes \( R_1 | B | S \) and \( R_2 | B | S \) be as follows:

\[
\begin{align*}
\downarrow & R_1 S_1 \xrightarrow{b} R_3 S_3 & \downarrow & R_1 S_1 \xrightarrow{a} R_3 S_3 \\
\quad & R_2 S_2 \xrightarrow{c} R_3 S_3 & \quad & R_2 S_2 \xrightarrow{d} R_1 S_4 \xrightarrow{a} R_3 S_5
\end{align*}
\]
The union \((R_1 \mid B \mid S) \uplus (R_2 \mid B \mid S)\) is

\[
\begin{align*}
R_1S1 & \xrightarrow{b} S3R3 \\
R_2S2 & \xrightarrow{a} R1S4 \xrightarrow{a} R3S5 \\
\end{align*}
\]

whereas \(R \mid B \mid S\) is the same with the exception that the state \(R3S5\) is totally defined.

Even if \(\cup_{i \in I, j \in J} R_i \mid B \mid S_j\) is not a cover, we can use it to test deadlocks in the following way. Our example is the erroneous protocol from the previous section [Lynch 68]. This time we modify the protocol so that the erroneous behaviour results in a deadlock [Tienari 89]. We also use a channel to deliver messages, i.e. the sender and receiver communicate asynchronously.

The sender process \(S\) and the receiver \(R\) are as follows:

**Sender**

- \(S1\)
  - take(d0)
- \(S2\)
  - \(n\)
    - d0a, de
  - \(a\)
- \(S3\)
  - a
- \(S4\)
  - e
- \(S5\)
  - take(d1)
- \(S6\)
  - \(n\)
    - d1a, de
  - \(e\)
- \(S7\)
  - d1n, de
- \(S8\)
  - e

**Receiver**

- \(R1\)
  - give(d0)
- \(R2\)
  - \(d1n, d0n\)
  - a, e
- \(R3\)
  - \(n, e\)
  - \(d1a\)
  - de
- \(R4\)
  - \(d0a\)
  - \(R5\)
  - \(R6\)
  - \(R7\)
  - \(R8\)
  - de
The sender receives a message (take(d0) or take(d1)) from an application. It sends the received message into a channel (d0a or d1a) with a positive acknowledgement, if everything has gone well. The sender gets a positive or negative acknowledgement (aa or nn) from the receiver. In addition, an acknowledgement can be distorted (e). A negative uncorrupted acknowledgement causes a retransmission of the previously sent message with a positive acknowledgement. The distorted acknowledgement also causes a retransmission of the previous message, but this time with a negative acknowledgement. Notice that the incoming messages are aa, nn and e (from the channel) whereas the outgoing messages (into the channel) are d0a, d0n, d1a, d1n. The messages take(d0) and take(d1) come from the application process that is not necessary to define.

The incoming messages to the receiver (from the channel) are dd0a, dd0n, dd1a, dd1n and a corrupted data packet de. R sends messages a and n into the channel. The messages give(d0) and give(d1) are delivered to an application which is not specified.

The channel consists of six parts,

\[ C = C1 \parallel C2 \parallel C3 \parallel C4 \parallel C5 \parallel C6, \]

where the components as Lotos processes are as follows:

\[ \begin{align*}
C1 & := d0a;(i;dd0a;C1 \parallel i;de;C1), \\
C2 & := d1a;(i;dd1a;C2 \parallel i;de;C2), \\
C3 & := d0n;(i;dd0n;C3 \parallel i;de;C3), \\
C4 & := d1n;(i;dd1n;C4 \parallel i;de;C4), \\
C5 & := a;(i;aa;C5 \parallel i;e;C5), \\
C6 & := n;(i;nn;C6 \parallel i;e;C6). 
\end{align*} \]

Define the action sets A and B as follows:

\[ \begin{align*}
A & = \{d0a, d0n, d1a, d1n, aa, nn, e\}, \\
B & = \{dd0a, dd0n, dd1a, dd1n, de, a, n\}. 
\end{align*} \]

The whole system is

\[ (S|A|C)|B|R. \]

Next we search for deadlocks. The reachability graph of the whole system consists of 78 states and 91 transitions, but let us pretend it is too large. We construct a cover for every channel process Ci, i = 1, ..., 6. In this way, we can partition the whole system into many processes as in Proposition 4.3.1. We do not know if the collection of processes is a cover or not, but we can analyse the existence of deadlocks. In this example we proceed sequentially and examine one process at a time.

Consider first C1. We partition it into two parts C11 and C12 as follows:
4.3. DISTRIBUTED DETECTION OF DEADLOCKS

\[ C11 := d0a; (i;dd0a; C11 \sqcup i;de;C11), \]
\[ C12 := d0a; (i;dd0a; C11 \sqcup i; udef). \]

Notice that \( C11 \) is same as \( C1 \). All the other channel processes are partitioned in the same way. Then we start to combine them. There will be 64 different channel processes \( C \) altogether and one of them is the original \( C \). We start to examine them from the simplest combinations. First we try the possibility that only the messages \( dd0 \) can be distorted:

\[ C1 := d0a;(i;dd0a;C1 \sqcup i;de;C1), \]
\[ C2 := d1a;(i;dd1a;C2 \sqcup i; udef), \]
\[ C3 := d0n;(i;dd0n;C3 \sqcup i; udef), \]
\[ C4 := d1n;(i;dd1n;C4 \sqcup i; udef), \]
\[ C5 := a;(i;aa;C5 \sqcup i; udef), \]
\[ C6 := n;(i;nn;C6 \sqcup i; udef). \]

With these definitions of the processes \( Ck, k = 1, \cdots, 6 \), we get a channel process \( CH_1 \) and the whole system is now

\[ (S\lvert A\rvert CH_1\rvert B \rvert R. \]

Its reachability graph consists of 32 states and 38 transitions. We have used the ARA-tool (Advanced Reachability Analysis) developed in the Computer Technology Laboratory of Technical Research Center in Finland to generate the transition graphs from Lotos specifications [Valmari & al 93]. The tool shows small graphs, less than 100 states, graphically. It is not able to handle partially defined states, but we have defined \texttt{udef} as an ordinary process and checked visually, if a state is totally or partially defined. Also we have analysed the graphs visually to detect deadlocks. In this case there are no totally defined deadlocked states.

Next, we can keep \( C2 \) in its original form and modify the other including \( C1 \) as previously. In this way we get the process

\[ (S\lvert A\rvert CH_2\rvert B \rvert R. \]

Again, no deadlocks are detected. We proceed in this way until we come to the process \( C5 \). The reachability graph now consists of 35 states and 40 transitions. If \( a \) can be distorted, it leads to two totally defined deadlocked states. The messages leading to the first deadlocked state are in order as follows:

\texttt{take(d0), d0a, i, dd0a, give(d0), a, i, e, d0n, i}

We see that a distortion of the acknowledgement \( a \) leads to a deadlock. The other deadlock is induced by a corrupted acknowledgement \( a \) of the message \( d1a \). \( \Box \)

In the above example we had, in fact, a cover, because the original system also belonged to the collection. This kind of cover is not very suitable, because some processes in it are nearly as large as the original system. Only the 6 or 12 smallest
processes are interesting and indeed we have detected deadlocks by analysing the five smallest processes in the cover.

We now develop an algorithm that works in the case, where we have several processes \( S_i, i \in I \), but the collection \( \{ S_i \}_{i \in I} \) is not necessarily a cover. The algorithm behaves as follows:

- It returns 'yes', if it finds a deadlock.
- It returns 'no', if the collection is a cover and no deadlocks are detected.
- The output is 'don’t know', if no deadlock is detected and the algorithm cannot show that the collection is a cover.

The algorithm consists of two phases.

**Algorithm: Distributed detection of deadlocks**

**Phase 1.**

**Initial configuration:** Process \( S \), processes \( \{ S_i \}_{i \in I} \) such that \( S_i \sqsubseteq_{wbi:ref} S \), processors \( pr_i, i \in I \). Input for processor \( pr_i \) is \( S_i \).

**Method for processor \( pr_i \):**

1. begin
2. Generate the global state graph for \( S_i \);
3. If there are totally defined deadlocked states then return 'yes' and stop;
4. If there are no totally defined deadlocked states, enter phase 2;
5. end.

If after phase 1 we have found totally defined deadlocked states, we can stop and correct the mistakes in the process \( S \). Notice that probably only part of all the deadlocked states are found in phase 1.

If no deadlocked states are found, we must continue. Because \( \{ S_i \}_{i \in I} \) is not necessarily a cover, we must examine partially defined states carefully in order to ensure that there are no deadlocks.

In phase 2 the processors examine the partially defined states. They check, if a partially defined state in their own global state graph is totally defined in the global state graph of another processor. After phase 2 each processor has a collection of partially defined states which are not totally defined in any other processor.
Phase 2.

Initial configuration: Processors \( pr_i, i \in I \), with the global state graphs \( S_i \).

Method for processor \( pr_i \):

1. begin
2. Let \( PS_i \) be the set of the partially defined states in \( S_i \);
3. Send every \( s \in PS_i \) to every other \( pr_j, j \in I, j \neq i \);
4. Receive states from other processors;
5. Let \( RS \) be the set of pairs \( (s, j) \), where \( s \) is a received state and \( j \) is the index of the processor that sent \( s \);
6. For every \( (s, j) \in RS \)
7. check, if \( s \) is in \( S_i \) and totally defined;
8. if it is, send back to \( pr_j \) the answer \( (s, \text{deadlock - free}) \);
9. end for;
10. Receive the answers from the other processors;
11. Let \( PS_i \) the set of states in the original \( PS_i \) that have not received the answer 'deadlock-free';
12. if \( PS_i = \emptyset \) then return 'no from i' otherwise return 'don't know';
13. end.

If after the second phase all the processors have returned 'no', the final output is 'no', i.e. the collection is a cover and there are no deadlocks. If at least one of the processors has output 'don’t know', then the final output is also 'don’t know', i.e. the collection is not a cover and there may be deadlocked states.

The output of the algorithm is correct for the following reasons. If every \( PS_i \) is empty, then we know that every state of the original system \( S \) is present and totally defined at least in one of the processes \( S_i \). Namely, suppose \( s \) is a state in \( S \). If \( s \) is not present and totally defined in any of \( S_i \), some of \( S_i \) must be partially defined. In fact we can say more: there must be some \( S_i \) such that \( S_i \) is partially defined and at least one partially defined state in \( S_i \) is not totally defined in any other \( S_j \). This means that \( PS_i \neq \emptyset \).

If on the other hand \( PS_i \) is not empty, then we know that the given collection is not a cover. There are states that are totally defined, but not present in any \( S_i \).
These states may be deadlocked or deadlock-free, but the algorithm does not know which one.

If the second phase returns 'don't know', the only alternatives are to try a different collection or to generate the missing states. The latter alternative leads to the traditional global state graph generation. It is possible to distribute the generation of the missing states using the process graphs $S_i$. The generation of the missing states in this way is slower than the traditional sequential method, because now there is a great deal of communication between the processors. However, time is not our major concern, but the memory space. The distributed algorithm uses the central memories of the processors instead of the disk and this compensates the communication cost.

We present the third phase of the algorithm in the case that we want to generate the missing states in a distributed way. In the third phase each processor $pr_i$ examines its set $PS_i$. It generates new states from the states in $PS_i$. Now everything is totally defined. Processors have the specifications of the original totally defined processes. It is an easy task to construct the totally defined state or states corresponding with a given partially defined state in $PS_i$. New states are generated from this totally defined state. After generating a new totally defined state, processors check that it is not totally defined already in some partially defined global state graph $S_i$. If it is, it is not necessary to generate further states from this new state. In other words, the processors generate parts of the original, totally defined global state graph until all states have been checked.

The examination of new global states demands synchronization between the processors. If two processors generate the same new global state, it is more efficient, if only one of the processors continue to generate states from the new state. This goal can be achieved as follows. After generating a new state a processor sends it to every other processor. Next the processor receives states from other processors. It compares the received states to its old, totally defined states and also to its new state. If a received state is the same as one of the old states, the processor sends information about the situation back to the processor that sent the state. It is not necessary for that processor to examine its new state anymore. If the received state is the same as the new state, it is necessary to generate more states from this new state, but it is enough that only one processor does this. The chosen processor can be that with the smallest index or the processor can be chosen randomly. We designate this phase in the algorithm simply with the procedure check_existence(state, status). The variable status has two values, old and new.

In the third phase we mark states old. If a state has not been marked old, it is new. If $s$ is a state, the set follow($s$) consists of states reachable with one step from $s$, i.e. $s' \in $ follow($s$) if and only if $s \xrightarrow{a} s'$ in $S$ for some $a \in A \cup \{\epsilon\}$. Notice that before constructing the set follow($s$), the state $s$ is completed to a totally defined state using the original specifications of the processes. Moreover, all the states in follow($s$) are totally defined.
Phase 3.

Initial configuration: Processors \( pr_i, i \in I \), with the global state graph \( S_i \) and set \( PS_i \) of partially defined states not found totally defined in the other processors. In addition, every processor has copies of the original, totally defined processes.

Method for processor \( pr_i \):

1. begin
2. Mark every state in \( S_i \) old;
3. While \( PS_i \neq \emptyset \) loop
4. take \( s \) from \( PS_i \);
5. If follow\( (s) = \emptyset \) then
6. return "\( s \) deadlocked" and stop;
7. else
8. for every new \( r \in \text{follow}(s) \) loop
9. check_existence\( (r, \text{status}) \);
10. if \( \text{status} = \text{new} \) then add \( r \) to \( PS_i \); end if;
11. end for;
12. mark \( s \) old;
13. end if;
14. end while;
15. end.

From these algorithms we see that in the case of a cover the situation is straightforward and the algorithm is much more efficient than without a cover. However, it is not easy to construct a good cover or to confirm that a collection is indeed a cover. In these cases we must apply the second version of the algorithm. Phase 2 can also be very efficient, if there are not many partially defined states. Phase 3 is slow, but if the collection of processes is carefully chosen, most states have already been generated and distributed in the processors. Thus, it is possible to examine large state graphs in a reasonable time.

It is an open question how to show that a given collection is indeed a cover. We should at least have some necessary conditions which guarantee that the collection is a cover. This is an interesting research topic.
Chapter 5

Divergence, BKO- and CFFD-Relations

In the previous chapters we have formulated the central concepts in the framework of the weak bisimulation equivalence. Next we want to study whether the same kind of approach can be applied to other equivalences. Compared to bisimilarity, the divergence bisimilarity is tighter, because it separates divergent states from convergent ones. In other respects it is like the bisimilarity. Traditionally, it has been customary not to consider divergences at all. But there are situations, where infinite internal computations must be taken into account. In verifying liveness properties, for example, using process algebraic techniques, divergences have to be taken into account.

The alternatives to the bisimulation equivalences considered in this thesis are the BKO- and CFFD-equivalences. They are essentially weaker than the two other equivalences and they belong to a different group of equivalences. However, they preserve many important properties of processes, and because of their coarseness, they allow stronger contractions. This can be useful when verifying large systems compositionally.

5.1 Divergence bisimilarity and refinement relations

The weak bisimilarity can be generalized in a straightforward way to include divergences as well. In the next definition we make a clear distinction between divergent and partially defined states. Our definition resembles closely the definitions in [Eloranta 91] and [Eloranta 94]. Other approaches are possible. For example, [Walker 90] considers partially defined states divergent in his definition of divergence bisimilarity.
Definition 5.1.1 A relation $\mathcal{R} \subseteq \text{ALTS} \times \text{ALTS}$ is a divergence (specification) bisimulation, if $(P_1, Q_1) \in \mathcal{R}$ implies that for all $a \in (\mathcal{A} \setminus \{\tau\}) \cup \{\varepsilon\}$:

1. $P_1 \Downarrow a$ if and only if $Q_1 \Downarrow a$,
2. $\text{Div}(P_1)$ if and only if $\text{Div}(Q_1)$,
3. a) $P_1 \xrightarrow{a} P_2$ implies that there is a $Q_2$ such that $Q_1 \xrightarrow{a} Q_2$ and $(P_2, Q_2) \in \mathcal{R}$,
   b) $Q_1 \xrightarrow{a} Q_2$ implies there is a $P_2$ such that $P_1 \xrightarrow{a} P_2$ and $(P_2, Q_2) \in \mathcal{R}$.

Divergence (specification) bisimilarity $\approx_{\text{div}}$ is defined by

$$\approx_{\text{div}} = \bigcup \{R \mid R \text{ is a divergence bisimulation}\}.$$  

It is an equivalence relation and congruence with respect to the same operators as the weak bisimilarity.

If we want to get a divergence refinement, which is compositional with respect to hiding and sequential composition, we have to combine partiality and divergence in some way. We don’t want to identify them as is done in [Walker 90], but try to find a different way to combine them instead.

Definition 5.1.2 A relation $\mathcal{R} \subseteq \text{ALTS} \times \text{ALTS}$ is a divergence prebisimulation, if $(P_1, Q_1) \in \mathcal{R}$ implies the following for all $a \in (\mathcal{A} \setminus \{\tau\}) \cup \{\varepsilon\}$:

1. If $P_1 \xrightarrow{a} P_2$ then for some $Q_2$, $Q_1 \xrightarrow{a} Q_2$ and $(P_2, Q_2) \in \mathcal{R}$.
2. If $\text{Div}(P_1)$, then $\text{Div}(Q_1)$.
3. If $P_1 \Downarrow a$ then
   a) $Q_1 \Downarrow a$,
   b) $\text{Div}(P_1)$ if and only if $\text{Div}(Q_1)$,
   c) $Q_1 \xrightarrow{a} Q_2$ implies there is a $P_2$ such that $P_1 \xrightarrow{a} P_2$ and $(P_2, Q_2) \in \mathcal{R}$.

The divergence refinement, $\sqsubseteq_{\text{divref}}$, is the largest divergence prebisimulation,

$$\sqsubseteq_{\text{divref}} = \bigcup \{\mathcal{R} \mid \mathcal{R} \text{ is a divergence prebisimulation}\}.$$  

The definition has two conditions for divergence, which may seem superfluous. Condition 2 guarantees that the divergence refinement behaves according to our intuition. For example, we want to exclude the case $P \sqsubseteq_{\text{divref}} Q$ for the processes $P$ and $Q$ below:
Intuitively speaking, it is not very satisfying, if a less defined process, in this example \( P \), can contain a divergence, whereas its refinement \( Q \) contains no divergences. The condition 3.b) is needed, because we want the divergence refinement to coincide with the divergence bisimilarity on totally defined systems. In the case of convergent processes Definition 5.1.2 agrees with the bisimulation refinement in Definition 3.3.1.

We can formulate the definition of the divergence prebisimulation in an alternative way as follows:

**Definition 5.1.3** A relation \( \mathcal{R} \subseteq \text{ALTS} \times \text{ALTS} \) is a divergence prebisimulation, if \( (P_1, Q_1) \in \mathcal{R} \) implies the following for all \( a \in (\mathcal{A} \setminus \{\tau\}) \cup \{\varepsilon\} \):

1. If \( P_1 \xrightarrow{a} P_2 \) then for some \( Q_2 \), \( Q_1 \xrightarrow{a} Q_2 \) and \( (P_2, Q_2) \in \mathcal{R} \).
2. If \( \text{Div}(P_1) \), then \( \text{Div}(Q_1) \).
3. If \( P_1 \perp a \) then
   a) \( Q_1 \perp a \),
   b) \( Q_1 \xrightarrow{a} Q_2 \) implies there is a \( P_2 \) such that \( P_1 \xrightarrow{a} P_2 \) and \( (P_2, Q_2) \in \mathcal{R} \).
4. If \( P_1 \perp \) then \( \text{Div}(P_1) \) if and only if \( \text{Div}(Q_1) \).

**Proposition 5.1.1** The two definitions of divergence prebisimulation define the same relation.

*Proof.* Suppose \( \mathcal{R} \) is a divergence prebisimulation of the first definition. We show that it is also a divergence prebisimulation of the second definition. We denote by D1 the first definition and by D2 the second definition.

Let \( (P, Q) \in \mathcal{R} \). Conditions 1 and 2 are the same in both definitions, so it is enough to check that condition 3 in D1 implies conditions 3 and 4 in D2.
Choose first \( a = \varepsilon \) and suppose \( P \downarrow \varepsilon \), i.e. \( P \downarrow \). Now 3 in D1 implies 3 in D2. In addition, 3 b) in D1 implies 4 in D2. If \( a \neq \varepsilon \) and \( P \downarrow a \), then \( P \downarrow \) and 3 in D1 implies both 3 and 4 in D2.

Suppose \( \mathcal{R} \) is a divergence prebisimulation of the second definition. Let \((P, Q) \in \mathcal{R}\). The only condition to be checked is 3 b) in D1. So suppose \( P \downarrow a \). Now also \( P \downarrow \) and 4 in D2 implies 3 b) in D1. \( \square \)

Definition 5.1.3 demonstrates that divergence properties have nothing to do with \( \uparrow a \)-concepts, but it is enough to consider only \( \uparrow \varepsilon \) with divergences. In the proofs we use, however, Definition 5.1.2, because it resembles the definition of weak bisimulation refinement.

The next two results show that the divergence refinement has all the characteristics of the bisimulation refinement.

**Proposition 5.1.2** The divergence refinement is a preorder.

*Proof.* The divergence refinement is clearly reflexive. To show the transitivity, let \( P \sqsubseteq_{\text{divref}} Q \) and \( Q \sqsubseteq_{\text{divref}} R \). There are divergence prebisimulations \( \mathcal{R} \) and \( \mathcal{S} \) such that \((P, Q) \in \mathcal{R}\) and \((Q, R) \in \mathcal{S}\). Now \( \mathcal{R} \circ \mathcal{S} = \{(P, R) \mid \exists Q : (P, Q) \in \mathcal{R}, (Q, R) \in \mathcal{S}\}\) is a divergence prebisimulation. More precisely, if \( P \rightharpoonup_{\alpha} P' \), then \( Q \rightharpoonup_{\alpha} Q' \) and \((P', Q') \in \mathcal{R}\). And there is a \( R' \) such that \( R \rightharpoonup_{\alpha} R' \) and \((Q', R') \in \mathcal{S}\). Thus \((P', R') \in \mathcal{R} \circ \mathcal{S}\). Condition 2 is clearly valid for \( P \) and \( R \). If \( P \downarrow a \), then \( Q \downarrow a \), \( R \downarrow a \) and Conv\((P)\) if and only if Conv\((Q)\) if and only if Conv\((R)\). If now \( R \rightharpoonup_{\alpha} R' \), then we can repeat the former reasoning backwards to conclude that for some \( P' \), \( P \rightharpoonup_{\alpha} P' \) and \((P', R') \in \mathcal{R} \circ \mathcal{S}\). We have thus proved that \( \mathcal{R} \circ \mathcal{S} \) is a divergence prebisimulation and hence \( \sqsubseteq_{\text{divref}} \) is transitive. \( \square \)

**Theorem 5.1.3** The divergence refinement is compositional with respect to the same operators as the bisimulation refinement (see Theorem 3.3.1).

*Proof.* If processes are convergent in all states, the divergence refinement agrees with the bisimulation refinement. Thus we can use the same examples as in the case of the bisimulation refinement to show that the divergence refinement is not compositional with respect to the operators choice and disabling. Otherwise we can proceed as in the proof of the theorem 3.3.2. So let \( P, Q, R \in \text{ALTS} \), \( P \sqsubseteq_{\text{divref}} Q, a \in \mathcal{A}, B \subset \mathcal{A}\backslash \{\tau\} \) and \( f \) a renaming function. Construct the relations \( \mathcal{R} \) in the same way as in the theorem 3.3.2, but using \( \sqsubseteq_{\text{divref}} \) instead of \( \sqsubseteq_{\text{biaref}} \). We have to show that the \( \mathcal{R} \)'s are divergence prebisimulations. The only new aspect to be checked is the boolean function Div. We give detailed proof only in the case of hiding, parallel composition and sequential composition. The other cases are easier.

\textbf{a) Parallel composition.} Let \((P|B|R, Q|B|R) \in \mathcal{R}\) and suppose Div\((P | B | R)\). Then either Div\((P)\) or Div\((R)\). Hence either Div\((Q)\) or Div\((R)\) and thus Div\((Q | B | R)\). Condition 2 of the definition of \( \sqsubseteq_{\text{divref}} \) is valid.
5.1. DIVERGENCE BISIMILARITY AND REFINEMENT RELATIONS

Suppose \((P|B|R) \downarrow a\). Reasoning in the same way as in the case b) of the theorem 3.3.2, we can conclude that also \((Q|B|R) \downarrow a\). We already know that \(\text{Div}(P|B|R)\) implies \(\text{Div}(Q|B|R)\). So suppose \(\text{Conv}(P|B|R)\). It follows that \(\text{Conv}(P)\) and \(\text{Conv}(R)\). Because \((P|B|R) \downarrow a\), we know that \(P \downarrow\). From the definition of \(P \sqsubseteq\text{divref} Q\) it can be concluded that \(\text{Conv}(Q)\). Hence \(\text{Conv}(Q|B|R)\).

b) Suppose \((h_B(P), h_B(Q)) \in \mathcal{R}\) and \(\text{Div}(h_B(P))\). There is an infinite chain

\[
h_B(P) \overset{\tau}{\rightarrow} h_B(P_1) \overset{\tau}{\rightarrow} h_B(P_2) \overset{\tau}{\rightarrow} \ldots
\]

By the definition of hiding we have an infinite chain

\[
P \overset{a_1}{\rightarrow} P_1 \overset{a_2}{\rightarrow} P_2 \overset{a_3}{\rightarrow} \ldots,
\]

where \(a_i \in B \cup \{\tau\}\) for all \(i \geq 1\). We can write this chain as

\[
P \overset{a_1}{\rightarrow} P_1 \overset{a_2}{\rightarrow} P_2 \overset{a_3}{\rightarrow} \ldots
\]

Because \(P \sqsubseteq\text{divref} Q\), we can proceed by induction to find an infinite chain

\[
Q \overset{b_1}{\rightarrow} Q_1 \overset{b_2}{\rightarrow} Q_2 \overset{b_3}{\rightarrow} \ldots,
\]

where \(b_i = a_i\) if \(a_i = \tau\) and \(b_i = \varepsilon\) if \(a_i = \tau\), for all \(i \geq 1\). Furthermore, \(P_i \sqsubseteq\text{divref} Q_i\) for every \(i \geq 1\).

If there is \(n \geq 1\) such that \(a_i = \tau\) for all \(i \geq n\), then \(\text{Div}(P_n)\) and thus also \(\text{Div}(Q_n)\). It follows that in this case also \(\text{Div}(h_B(Q))\). Otherwise, suppose that \(a_i \neq \tau\) for infinitely many \(i\). Then also \(b_i \neq \varepsilon\) for infinitely many \(i\). Hence we have a chain

\[
h_B(Q) \overset{c_1}{\rightarrow} h_B(Q_1) \overset{c_2}{\rightarrow} h_B(Q_2) \overset{c_3}{\rightarrow} \ldots,
\]

where \(c_i = \varepsilon\) or \(c_i = \tau\) and \(c_i = \tau\) for infinitely many \(i \geq 1\). But this means that \(\text{Div}(h_B(Q))\) and the condition 2 has been proved.

Next suppose that \(h_B(P) \downarrow a\) and \(\text{Conv}(h_B(P))\). We must show that \(\text{Conv}(h_B(Q))\). If \(\text{Div}(h_B(Q))\), then there is an infinite chain

\[
h_B(Q) \overset{\tau}{\rightarrow} h_B(Q_1) \overset{\tau}{\rightarrow} h_B(Q_2) \overset{\tau}{\rightarrow} \ldots
\]

Because \(h_B(P) \downarrow a\), we have \(h_B(P) \downarrow\) and also \(h_B(Q) \downarrow\). Thus \(Q \downarrow\) and \(Q_i \downarrow, i \geq 1\).

By the definition of hiding we have a chain

\[
Q \overset{b_1}{\rightarrow} Q_1 \overset{b_2}{\rightarrow} Q_2 \overset{b_3}{\rightarrow} \ldots,
\]

where \(b_i \in B \cup \{\tau\}\) for all \(i \geq 1\). We can deduce in the same way as in the theorem 3.3.2 in c) that there is a chain \(P \overset{c_1}{\rightarrow} P_1 \overset{c_2}{\rightarrow} P_2 \overset{c_3}{\rightarrow} \ldots\), where \(c_i = b_i\) if \(b_i \neq \tau\) and \(c_i = \varepsilon\) if \(b_i = \tau\). Furthermore, \(P_i \sqsubseteq\text{divref} Q_i\) for all \(i \geq 1\). Because \(h_B(P) \downarrow\), we know that \(P \downarrow\) and \(P_i \downarrow\) for all \(i \geq 1\). If \(b_i = \tau\) when \(i \geq n\) for some \(n \geq 1\), then \(\text{Div}(Q_n)\).

Because \(P_n \sqsubseteq\text{divref} Q_n\) and \(P_n \downarrow\), we have that also \(\text{Div}(P_n)\) and hence \(\text{Div}(h_B(P))\), a contradiction! If, on the other hand, \(b_i \neq \tau\) for infinitely many \(i\), then also \(c_i \neq \varepsilon\)
for infinitely many $i$ and hence $\text{Div}(h_B(P))$, a contradiction! Thus we must have $\text{Conv}(h_B(Q))$. 

c) Because sequential composition is not commutative, it is necessary to consider two cases. Let $P \sqsubseteq^{\text{div ref}} Q$ and first consider the processes $P \gg R$ and $Q \gg R$. If $\text{Div}(P \gg R)$, then either $\text{Div}(P)$ or $P \xrightarrow{a} P'$, $P' \xrightarrow{\delta} P''$ and $\text{Div}(R)$. Because $P \sqsubseteq^{\text{div ref}} Q$, either $\text{Div}(Q)$ or $Q \xrightarrow{\delta} Q'$, $P' \sqsubseteq^{\text{div ref}} Q'$, $Q' \xrightarrow{\delta} Q''$, respectively. It follows that also $\text{Div}(Q \gg R)$.

First suppose that $(P \gg R) \not\Downarrow$. Then $P \not\Downarrow$ and if $\xrightarrow{\delta} \text{stop}$, also $R \not\Downarrow$. Because $P \sqsubseteq^{\text{div ref}} Q$, it follows that $Q \not\Downarrow$ and in the case $\xrightarrow{\delta} \text{stop}$ also $Q \xrightarrow{\delta} \text{stop}$ and $R \not\Downarrow$. Hence $(Q \gg R) \not\Downarrow$. If $a \in A \setminus \{\tau\}$ and $(P \gg R) \not\Downarrow a$, then $(P \gg R) \not\Downarrow$ and hence $(Q \gg R) \not\Downarrow$. If $P \gg R \xrightarrow{a} P' \gg R$, then $P \xrightarrow{a} P'$ and by $P \sqsubseteq^{\text{div ref}} Q$ there is a $Q'$ such that $Q \xrightarrow{\alpha} Q'$ and $P' \sqsubseteq^{\text{div ref}} Q'$. Because $(P \gg R) \not\Downarrow a$, also $(P' \gg R) \not\Downarrow$ and hence $(Q \gg R) = \xrightarrow{\alpha} (Q' \gg R)$ and $(Q \gg R) \not\Downarrow$. It follows $(Q \gg R) \not\Downarrow a$. Thus for all $a \in A$, $(P \gg R) \not\Downarrow a$ implies $(Q \gg R) \not\Downarrow a$.

Suppose $(P \gg R) \not\Downarrow a$, $a \in (A \setminus \{\tau\}) \cup \{\varepsilon\}$, and $\text{Conv}(P \gg R)$. This implies $\text{Conv}(P)$ and also $\text{Conv}(Q)$ because $P \sqsubseteq^{\text{div ref}} Q$ and $P \Downarrow$. If now $\text{Div}(Q \gg R)$ then $Q \xrightarrow{\alpha} Q'$ otherwise $Q \gg R \gg R$ and $\text{Div}(R)$. By $(P \gg R) \Downarrow$ and $(P \gg R) \sqsubseteq^{\text{whi ref}} (Q \gg R)$ (3.3.2), also $(P \gg R) \gg R$. Hence $\text{Div}(P \gg R)$, a contradiction. We have proved $(P \gg R) \Downarrow a$ and $\text{Conv}(P \gg R)$ imply $\text{Conv}(Q \gg R)$ and the condition 3.b) follows.

Secondly, consider the processes $R \gg P$ and $R \gg Q$. The only interesting case is $a = \delta$, but this reduces to the case $P \sqsubseteq^{\text{div ref}} Q$. \qed

We can modify the definition of the partial bisimulation refinement to obtain another divergence refinement, which we call partial divergence refinement and denote it by $\sqsubseteq^{\text{div ref}}$.

**Definition 5.1.4** A relation $\mathcal{R} \subseteq \text{ALTS} \times \text{ALTS}$ is a partial divergence bisimulation, if for all $(P, Q) \in \mathcal{R}$ and for all $a \in (A \setminus \{\tau\}) \cup \{\varepsilon\}$ such that $P \Downarrow a$ the following holds:

1. $Q \Downarrow a$;
2. $\text{Div}(P)$ if and only if $\text{Div}(Q)$;
3. if $P \xrightarrow{a} P'$, then for some $Q'$, $Q \xrightarrow{\alpha} Q'$ and $(P', Q') \in \mathcal{R}$;
4. if $Q \xrightarrow{\alpha} Q'$, then for some $P'$, $P \xrightarrow{\alpha} P'$ and $(P', Q') \in \mathcal{R}$.

The partial divergence refinement, $\sqsubseteq^{\text{div ref}}$, is the largest partial divergence bisimulation,

$$
\sqsubseteq^{\text{div ref}} = \bigcup \{\mathcal{R} | \mathcal{R} \text{ is a partial divergence bisimulation}\}.
$$
5.2. BKO-EQUIVALENCE AND REFINEMENT RELATIONS

The partial divergence refinement is a preorder and it is compositional with the same operators as the divergence refinement. It agrees with the divergence bisimilarity on totally defined processes. The next proposition shows some relations between divergence bisimilarity and divergence refinements.

**Proposition 5.1.4** Let \( \approx_{\text{divref}} \) be the equivalence \( \subseteq_{\text{divref}} \cap (\subseteq_{\text{divref}})^{-1} \) and \( \approx_{\text{divref}} \) the equivalence \( \subseteq_{\text{divref}} \cap (\subseteq_{\text{divref}})^{-1} \). Then

(i) \( \subseteq_{\text{divref}} \subset \subseteq_{\text{divref}} \) properly;

(ii) \( \approx_{\text{divref}} \subset \approx_{\text{divref}} \) properly;

(iii) \( \approx_{\text{div}} \subset \approx_{\text{divref}} \) properly.

**Proof.** i) Follows from the definitions.

ii) -iii) If there are no divergences, the divergence bisimilarity, divergence refinement and partial divergence refinement coincide with the weak bisimilarity, bisimulation refinement and partial bisimulation refinement, respectively. So we can use the counterexamples in the propositions 3.3.2 and 3.5.1 to deduce that the subsets in ii) and iii) are proper. We are left to show the inclusions.

The inclusion in ii) is clear, because \( P \subseteq_{\text{divref}} Q \) implies \( P \subseteq_{\text{divref}} Q \). The inclusion in iii) follows directly from the definitions. \( \square \)

5.2 BKO-equivalence and refinement relations

One of our main aims is to show that different types of equivalences allow the same kind of refinement relations as the bisimulation equivalence. We have chosen as our examples the BKO-equivalence ([Bergstra, Klop, Olderog 86], [Bergstra, Klop, Olderog 87]) and the CFFD-equivalence ([Valmari and Tienari 91], [Valmari and Tienari 95]). Both belong to the class of failure equivalences or decorated trace equivalences.

The BKO-equivalence resembles the testing equivalence as given, for example, in [Celikkhan and Cleaveland 92] and [Cleaveland and Hennessy 93]. The reason we have chosen the BKO-equivalence is that it is close to the CFFD-equivalence. The CFFD-equivalence is an interesting example of failure equivalences, because it behaves well in the presence of divergences. The BKO-equivalence is the same as the CFFD-equivalence without divergences. Thus, it can be used in a protocol design in our way and, moreover, the step from BKO to CFFD is a short and straightforward one.

The testing preorders defined in [Celikkhan and Cleaveland 92] and [Cleaveland and Hennessy 93] differ essentially from our refinement relations. The preorders of the previous articles are implementing relations, i.e. looser relations than our refinements. It is possible to define refinement relations for the various versions of
the testing equivalences as well. In fact, this can be done by modifying the definition of the BKO-refinement.

Before defining the BKO-equivalence and related refinement relations we introduce some concepts needed in definitions.

Let $P = (Q, A, T, q_0, \uparrow)$ be a partially defined labeled transition system. We define:

1. The set of stable failures, $\text{sfail}(P)$, is a collection of pairs $(u, L)$ such that
   (i) $u \in \text{tr}(P)$,
   (ii) $L \subset A \setminus \{\tau\}$,
   (iii) there is a state $q$ such that $q_0 \xrightarrow{u} q$, $q \downarrow$ and $q \xrightarrow{a \notin L} q$ for all $a \in L \cup \{\tau\}$.

2. The set of traces to partially defined states is
   $$\text{partr}(P) = \{u \in \text{tr}(P) \mid \exists q : q_0 \xrightarrow{u} q \text{ and } q \uparrow\},$$

3. A process $P$ is stable, if there are no $\tau$-transitions from the initial state of $P$. We denote this situation by $\text{stable}(P)$.

4. Note that the trace set $\text{tr}(P)$ consists of all possible traces. Thus for example $\text{partr}(P) \subset \text{tr}(P)$.

**Definition 5.2.1** Processes $P$ and $Q$ are BKO-(specification) equivalent, $P \approx_{\text{BKO}} Q$, if and only if

1. $\text{stable}(P)$ if and only if $\text{stable}(Q)$,
2. $\text{tr}(P) = \text{tr}(Q)$,
3. $\text{partr}(P) = \text{partr}(Q)$,
4. $\text{sfail}(P) = \text{sfail}(Q)$.

Our BKO-equivalence on totally defined processes is essentially the same as one of the equivalences presented in [Bergstra, Klop, Olderog 87]. Bergstra, Klop and Olderog defined the equivalence in different terms whereas we have used the concepts and notations presented in [Valmari and Tienari 95].

One modification in our definition is the addition of condition 3, because we have undefined states in processes. Without condition 3 we would have strange relations $P \approx_{\text{BKO}} Q$, which we want to avoid. There are many alternatives to extend the definition of the set $\text{sfail}$ to comprise partially defined processes. From our standpoint partially defined states are incomplete and any transition can start from such a state. Only completely defined states determine stable failure sets. However, we have kept the definition of the predicate $\text{stable}$ intact. Thus, a process can be stable whether or not its initial state is partially defined. It would be possible
to define stable otherwise, for example so that the partially defined initial state always is unstable. It seems that the choice between these two is arbitrary, it does not depend on deeper reasons.

In the case of totally defined processes the new definition of $\text{sfail}$ coincides with the original definition in the article [Valmari and Tienari 95] and $\text{partr}(P_1) = \emptyset$. Thus for totally defined processes the extended definition of the BKO-equivalence agrees with the original one.

Next, we define the corresponding BKO-refinement. Before giving a formal definition, it is useful to analyse how refinements are done and what kind of implications they have for labeled transition systems.

If $P$ is a partially defined Lotos process, it can be refined by replacing one $\text{udef}$ by a new Lotos process $P'$. The process $P'$ can be partially or totally defined. It can also be the process $\text{stop}$. Thus, if we denote by $Q$ a refinement of $P$, the following conditions should be valid:

C1. $\text{tr}(P) \subseteq \text{tr}(Q)$,

C2. especially, $\text{partr}(P) \subseteq \text{tr}(Q)$,

C3. every trace of $Q$ is a trace of $P$ or has a trace to a partially defined state in $P$ as a prefix,

C4. every element of $\text{sfail}(P)$ is also an element in $\text{sfail}(Q)$ and every element of $\text{sfail}(Q)$ is an element of $\text{sfail}(P)$ or it is possible to create it in $P$ by replacing $\text{udef}$ in $P$ by a new process.

Following the above conditions we get the next definition.

**Definition 5.2.2** A process $Q$ is a BKO-refinement of a process $P$, $P \sqsubseteq_{\text{BKO-ref}} Q$, if

1. $\text{tr}(P) \subseteq \text{tr}(Q)$ and for every $u \in \text{tr}(Q)$, either $u \in \text{tr}(P)$ or there is a prefix $u_1$ of $u$ such that $u_1 \in \text{partr}(P)$;

2. for every $u \in \text{partr}(Q)$ there is a prefix $u_1$ of $u$ such that $u_1 \in \text{partr}(P)$;

3. $P \downarrow \implies (\text{stable}(P) \iff \text{stable}(Q))$,

4. $\text{sfail}(P) \subseteq \text{sfail}(Q)$ and for every $(u, L) \in \text{sfail}(Q)$, either $(u, L) \in \text{sfail}(P)$ or there is a prefix $u_1$ of $u$ such that $u_1 \in \text{partr}(P)$.

**Proposition 5.2.1** The BKO-refinement $\sqsubseteq_{\text{BKO-ref}}$ is a preorder.
Proof. The relation $\sqsubseteq_{\text{BKO-ref}}$ is clearly reflexive. To show the transitivity, suppose the relations $P \sqsubseteq_{\text{BKO-ref}} Q$ and $Q \sqsubseteq_{\text{BKO-ref}} R$ are valid. Now $\text{tr}(P) \subset \text{tr}(Q) \subset \text{tr}(R)$ and $\text{sfail}(P) \subset \text{sfail}(Q) \subset \text{sfail}(R)$.

By the definition of $\Downarrow$, $P \Downarrow$ if and only if $\varepsilon \not\in \text{partr}(P)$ and $Q \Uparrow$ if and only if $\varepsilon \in \text{partr}(Q)$. Hence $P \Downarrow$ implies $Q \Downarrow$ because otherwise we have a contradiction with the assumption $P \sqsubseteq_{\text{BKO-ref}} Q$, especially with condition 2 in the definition of BKO-refinement. Similarly $Q \Downarrow$ implies $R \Downarrow$. Hence, if $P \Downarrow$, then $\text{stable}(P)$ if and only if $\text{stable}(Q)$ if and only if $\text{stable}(R)$. If $P \Uparrow$ and $\sim \text{stable}(P)$, then $\sim \text{stable}(Q)$. Now $\sim \text{stable}(Q)$ implies $\sim \text{stable}(R)$ irrespective of whether $Q \Downarrow$ or $Q \Uparrow$.

Suppose $u \in \text{partr}(R)$. Then there is a prefix $u_1$ of $u$ such that $u_1 \in \text{partr}(Q)$ and a prefix $u_2$ of $u_1$ such that $u_2 \in \text{partr}(P)$.

If $(u, L) \in \text{sfail}(R)$, then $(u, L) \in \text{sfail}(Q)$ or $u_1 \in \text{partr}(Q)$ for some prefix $u_1$ of $u$. In the former case, either $(u, L) \in \text{sfail}(P)$ or $u_2 \in \text{partr}(P)$ for some prefix $u_2$ of $u$. In the latter case, $u_2 \in \text{partr}(P)$ for some prefix $u_2$ of $u_1$. But this $u_2$ is also a prefix of $u$. $\square$

**Theorem 5.2.2** The BKO-refinement is compositional with respect to all the Lotos operators.

Proof. The proof is analogous to the corresponding congruence proof in the article [Valmari and Tienari 95].

Most of the properties are clear from the definitions, but we go through the details for safety’s sake. So suppose $P \sqsubseteq_{\text{BKO-ref}} Q$.

1) **Action prefix.** The claim is that $a; P \sqsubseteq_{\text{BKO-ref}} a; Q$ for all $a \in A$. First notice that

$$
\begin{align*}
\text{tr}(\tau; P) &= \text{tr}(P), \\
\text{partr}(\tau; P) &= \text{partr}(P), \\
\text{sfail}(\tau; P) &= \text{sfail}(P), \\
\sim \text{stable}(\tau; P)
\end{align*}
$$

and similarly for $Q$. Thus obviously we have $\tau; P \sqsubseteq_{\text{BKO-ref}} \tau; Q$.

Suppose next $a \neq \tau$. Now $\text{stable}(a; P)$ and $\text{stable}(a; Q)$. The other properties follow from the facts

$$
\begin{align*}
\text{tr}(a; P) &= \{au | u \in \text{tr}(P)\} \cup \{\varepsilon\}, \\
\text{partr}(a; P) &= \{au | u \in \text{partr}(P)\}, \\
\text{sfail}(a; P) &= \{(au, L) | (u, L) \in \text{sfail}(P)\} \cup \{(\varepsilon, L) | a \not\in L\}
\end{align*}
$$

and similarly for $Q$. 

2) Choice. Let $R \in \text{ALTS}$. We claim that $P[|]R \equiv_{\text{BKOR-ref}} Q[|]R$. Firstly note that for all $P$ and $R$,

$$
\text{tr}(P[|]R) = \text{tr}(P) \cup \text{tr}(R), \\
\text{partr}(P[|]R) = \text{partr}(P) \cup \text{partr}(R).
$$

The choice rule for $\text{sfail}$ is more complicated but easily deductible:

$$
\text{sfail}(P[|]R) = \{(u, L) \in \text{sfail}(P) \cup \text{sfail}(R) | u \neq \varepsilon\} \cup X,
$$

where

$$
X = \begin{cases} 
\{(\varepsilon, L) | (\varepsilon, L) \in \text{sfail}(P) \cap \text{sfail}(R)\}, & \text{if stable}(P) \land \text{stable}(R), \\
\{(\varepsilon, L) | (\varepsilon, L) \in \text{sfail}(R)\}, & \text{if stable}(P) \land \sim \text{stable}(R), \\
\{(\varepsilon, L) | (\varepsilon, L) \in \text{sfail}(P)\}, & \text{if } \sim \text{stable}(P) \land \text{stable}(R), \\
\{(\varepsilon, L) | (\varepsilon, L) \in \text{sfail}(P) \cup \text{sfail}(R)\}, & \text{if } \sim \text{stable}(P) \land \sim \text{stable}(R). 
\end{cases}
$$

Now it is seen at once that $\text{tr}(P[|]R) \subseteq \text{tr}(Q[|]R)$. If $u \in \text{partr}(Q[|]R)$, then $u \in \text{partr}(Q)$ or $u \in \text{partr}(R)$. If the latter, then $u \in \text{partr}(P[|]R)$. If the former, then $u_1 \in \text{partr}(P)$ for some prefix $u_1$ of $u$. Thus $u \in \text{partr}(P[|]R)$ or $u_1 \in \text{partr}(P[|]R)$.

If $(P[|]R) \downarrow$ and stable$(P[|]R)$, then $P \downarrow$, $R \downarrow$, stable$(P)$ and stable$(R)$. It follows that also stable$(Q)$ and hence stable$(Q[|]R)$. The case $(P[|]R) \downarrow$ and $\sim$ stable$(P[|]R)$ is proved in the same way as the next case. If $(P[|]R) \uparrow$ and $\sim$ stable$(P[|]R)$, then $P \uparrow$ or $R \uparrow$ and $\sim$ stable$(P)$ or $\sim$ stable$(R)$. If $\sim$ stable$(P)$, then $\sim$ stable$(Q)$. Hence $\sim$ stable$(Q[|]R)$ in all the cases.

Suppose $(u, L) \in \text{sfail}(P[|]R)$. If $u \neq \varepsilon$, then $(u, L) \in \text{sfail}(P) \cup \text{sfail}(R) \subseteq \text{sfail}(Q) \cup \text{sfail}(R)$ and hence $(u, L) \in \text{sfail}(Q[|]R)$. If $u = \varepsilon$, then we have four cases. Firstly, suppose stable$(P)$ and stable$(R)$ and $(\varepsilon, L) \in \text{sfail}(P) \cap \text{sfail}(R)$. Then $(\varepsilon, L) \in \text{sfail}(Q) \cap \text{sfail}(R)$ and thus $(\varepsilon, L) \in \text{sfail}(Q[|]R)$ independently whether stable$(Q)$ or not. Secondly, if stable$(P)$ and $\sim$ stable$(R)$, then $(\varepsilon, L) \in \text{sfail}(R)$ and hence $(\varepsilon, L) \in \text{sfail}(Q[|]R)$. Thirdly, if $\sim$ stable$(P)$ and stable$(R)$, then $(\varepsilon, L) \in \text{sfail}(P) \cap \text{sfail}(Q)$, $\sim$ stable$(Q)$ and hence $(\varepsilon, L) \in \text{sfail}(Q[|]R)$. Finally, if $\sim$ stable$(P)$ and $\sim$ stable$(R)$, then $(\varepsilon, L) \in \text{sfail}(P) \cup \text{sfail}(R)$. Because $\sim$ stable$(Q)$, we have $(\varepsilon, L) \in \text{sfail}(Q[|]R)$. We have shown that $\text{sfail}(P[|]R) \subseteq \text{sfail}(Q[|]R)$.

To show the other part of condition 4), let $(u, L) \in \text{sfail}(Q[|]R)$. Assume first that $u \neq \varepsilon$. Then $(u, L) \in \text{sfail}(Q)$ or $(u, L) \in \text{sfail}(R)$. If the latter, then $(u, L) \in \text{sfail}(P[|]R)$. If the former, then either $(u, L) \in \text{sfail}(P)$ or $u_1 \in \text{partr}(P)$ for a prefix $u_1$ of $u$. Hence either $(u, L) \in \text{sfail}(P[|]R)$ or $u_1 \in \text{partr}(P[|]R)$.

Assume now $u = \varepsilon$. If stable$(Q)$, stable$(R)$ and thus $(\varepsilon, L) \in \text{sfail}(Q) \cap \text{sfail}(R)$, then $(\varepsilon, L) \in \text{sfail}(P)$ or $\varepsilon \in \text{partr}(P)$. If the former, then $(\varepsilon, L) \in \text{sfail}(P[|]R)$ independent whether stable$(P)$ or $\sim$ stable$(P)$ by the definition of $X$. If the latter, then $P \uparrow$ and consequently $(P[|]R) \uparrow$. Thus $\varepsilon \in \text{partr}(P[|]R)$.

If stable$(Q)$, $\sim$ stable$(R)$ and thus $(\varepsilon, L) \in \text{sfail}(R)$, then $(\varepsilon, L) \in \text{sfail}(P[|]R)$ independently of the truth value of stable$(P)$. 
If $\sim$ stable($Q$), stable($R$) and thus $(\varepsilon, L) \in$ sfail($Q$), then $(\varepsilon, L) \in$ sfail($P$) or $\varepsilon \in$ partr($P$). If the latter, then $\varepsilon \in$ partr($P$)|$\{R\}$. So suppose $\varepsilon \in$ partr($P$). This means that $P \downarrow$. It follows that $\sim$ stable($P$) and hence $(\varepsilon, L) \in$ sfail($P$)|$\{R\}$.

If $\sim$ stable($Q$) and $\sim$ stable($R$) and thus $(\varepsilon, L) \in$ sfail($Q$)$\cup$sfail($R$), then either $(\varepsilon, L) \in$ sfail($Q$) or $(\varepsilon, L) \in$ sfail($R$). If the latter, then $(\varepsilon, L) \in$ sfail($P$)|$\{R\}$. If the former, then $(\varepsilon, L) \in$ sfail($P$) or $\varepsilon \in$ partr($P$). Again we can assume $\varepsilon \notin$ partr($P$), because otherwise the claim is clear. It follows that $P \downarrow$ and $\sim$ stable($P$). Thus $(\varepsilon, L) \in$ sfail($P$)|$\{R\}$.

3) Sequential composition. If $u$ is a string of actions, then the set actions($u$) consists of the actions in $u$. Now, because $P$ is $\delta$-consistent, we have

$$
\text{tr}(P \gg R) = \{u \mid u \in \text{tr}(P) \land \delta \notin \text{actions}(u)\} \cup \\
\{uv \mid u\delta \in \text{tr}(P) \land v \in \text{tr}(R)\},
$$

$$
\text{partr}(P \gg R) = \text{partr}(P) \cup \{uv \mid u\delta \in \text{tr}(P) \land v \in \text{partr}(R)\},
$$

$$
\text{sfail}(P \gg R) = \{(u, L) \mid (u, L \cup \{\delta\}) \in \text{sfail}(P) \land \delta \notin \text{actions}(u)\} \cup \\
\{(uv, L) \mid u\delta \in \text{tr}(P), \ (v, L) \in \text{sfail}(R)\},
$$

$$
\text{stable}(P \gg R) = \text{stable}(P) \land \delta \notin \text{tr}(P).
$$

We have to consider two cases, because sequential composition is not commutative. First consider the processes $P \gg R$ and $Q \gg R$.

If $P \sqsubseteq_{\text{BKO ref}} Q$, then tr($P$) $\subseteq$ tr($Q$). By the formula for tr($P \gg R$), we see that tr($P \gg R$) $\subseteq$ tr($Q \gg R$). Suppose $u \in$ tr($Q \gg R$). Then either $u \in$ tr($Q$) or $u = v\delta$, where $v\delta \in$ tr($Q$) and $v \in$ tr($R$). In the former case, either $u \in$ tr($P$) or $u_1 \in$ partr($P$) for a prefix $u_1$ of $u$. It follows that in this case $u \in$ tr($P \gg R$) or $u_1 \in$ partr($P \gg R$). Consider now the latter case, $v\delta \in$ tr($Q$), $w \in$ tr($R$). The only possibility is that $(v\delta, L) \in$ sfail($Q$) for all action sets $L \subseteq A \setminus \{\tau\}$, where $A$ is the action set of $Q$. Now either $(v\delta, L) \in$ sfail($P$) or $v_1 \in$ partr($P$) for some prefix $v_1$ of $v\delta$. If $(v\delta, L) \in$ sfail($P$), then $v\delta \in$ tr($P \gg R$). If on the other hand $v_1 \in$ partr($P$), then also $v_1 \in$ partr($P \gg R$). We have proved the condition 1) in the definition of the BKO-refinement.

Conditions 2) and 4) for partr and sfail are proved in the same way as condition 1) for traces.

Consider condition 3) for stable. Suppose $(P \gg R) \downarrow$. It follows that $P \downarrow$ and if $P \xrightarrow{\delta} P'$, then also $R \downarrow$. Assume furthermore stable($P \gg R$). Now stable($P$) and $\delta \notin$ tr($P$). Because $P \sqsubseteq_{\text{BKO ref}} Q$ and $P \downarrow$, we know that also stable($Q$). If $\delta \in$ tr($Q$), then $\varepsilon \in$ partr($P$), because $\delta \notin$ tr($P$). Hence $P \uparrow$, a contradiction. Thus $\delta \notin$ tr($Q$) and we have stable($Q \gg R$).

Assume next $(P \gg R) \downarrow$ and $\sim$ stable($P \gg R$). In this case $\sim$ stable($P$) or $\delta \in$ tr($P$). If $\sim$ stable($P$), then also $\sim$ stable($Q$) and thus $\sim$ stable($Q \gg R$). If
5.2. BKO-Equivalence and Refinement Relations

δ ∈ tr(P), then also δ ∈ tr(Q) and we have \( ∼ \) stable(Q \( >> \) R). We have proved that in the case (P \( >> \) R) \( \Downarrow \), stable(P \( >> \) R) if and only if stable(Q \( >> \) R).

If (P \( >> \) R) \( \Uparrow \) and ∼ stable(P \( >> \) R), then ∼ stable(P) or δ ∈ tr(P). Suppose ∼ stable(P) and δ \∉ tr(P), because the other case is similar with the previous case. But now ∼ stable(Q) and hence ∼ stable(Q \( >> \) R). Condition 3) has been proved.

Secondly, consider the processes R \( >> \) P and R \( >> \) Q. This case is easier and all the conditions follow directly from the formulas and the property P \( \subseteq_{BKOref} \) Q.

4) Disabling. The starting point in the proof are the relations:

\[
\begin{align*}
\text{tr}(P \supset R) & = \text{tr}(P) \cup \{uv \mid u \in \text{tr}(P) \land \delta \notin \text{actions}(u) \land v \in \text{tr}(R)\}, \\
\text{partr}(P \supset R) & = \text{partr}(P) \cup \{uv \mid u \in \text{tr}(P) \land \delta \notin \text{actions}(u) \land v \in \text{partr}(R)\}, \\
\text{stable}(P \supset R) & = \text{stable}(P) \land \text{stable}(R), \\
\text{sfail}(P \supset R) & = \{ (u, L) \mid (u, L) \in \text{sfail}(P) \} \cup X,
\end{align*}
\]

where

\[
X = \{ (uv, L) \mid u \in \text{tr}(P) \land \delta \notin \text{actions}(u) \land (v, L) \in \text{sfail}(R) \},
\]

if ∼ stable(R), and

\[
X = \{ (uv, L) \mid u \in \text{tr}(P) \land \delta \notin \text{actions}(u) \land (v, L) \in \text{sfail}(R) \land v \neq \varepsilon \} \cup \{ (u, L) \mid (u, L) \in \text{sfail}(P) \land (\varepsilon, L) \in \text{sfail}(R) \},
\]

if stable(R).

Again, we have to consider two cases because of non-commutativity. First consider the processes P \( \supset R \) and Q \( \supset R \). It is clear that tr(P \( \supset R \)) \( \subseteq \) tr(Q \( \supset R \)) and sfail(P \( \supset R \)) \( \subseteq \) sfail(Q \( \supset R \)). Suppose u ∈ tr(Q \( \supset R \)). If u ∈ tr(Q), then the second part of the condition 1) in the definition of BKO-refinement is readily seen as valid. So assume that u = vuv, v ∈ tr(Q), δ \notin \text{actions}(v) and w ∈ tr(R).

We know that v ∈ tr(P) or v₁ ∈ partr(P) for a prefix v₁ of v. If v ∈ tr(P), then vuv ∈ partr(P \( \supset R \)). Thus condition 1) has been proved.

Condition 2) is proved in the same way.

Suppose (P \( \supset R \)) \( \Downarrow \). Then P \( \Downarrow \) and stable(P) if and only if stable(Q). It follows that stable(P \( \supset R \)) if and only if stable(Q \( \supset R \)). If (P \( \supset R \)) \( \Uparrow \) and ∼ stable(P \( \supset R \)), then either ∼ stable(P) or ∼ stable(R). In both cases it follows that Q \( \supset R \) is unstable.

The second part of condition 4) for sfail is proved in the same way as the second part for condition 1).

Secondly, we have the processes R \( \supset P \) and R \( \supset Q \). This case is simpler and all follows directly either from the formulas, the property P \( \subseteq_{BKOref} \) Q or similar reasoning as above.
5) Hiding. If $u$ is a string of actions and $h_B$ a hiding operator, we denote by $h_B(u)$ the string obtained from $u$ by deleting the actions in $B$. We have the relations:

\[
\begin{align*}
\text{tr}(h_B(P)) &= \{h_B(u) \mid u \in \text{tr}(P)\}, \\
\text{partr}(h_B(P)) &= \{h_B(u) \mid u \in \text{partr}(P)\}, \\
\text{sfail}(h_B(P)) &= \{(h_B(u), L) \mid (u, L \cup B) \in \text{sfail}(P)\}, \\
\text{stable}(h_B(P)) &= \text{stable}(P) \land B \cap \text{tr}(P) = \emptyset.
\end{align*}
\]

The conditions for $\text{Tr}$ and $\text{partr}$ are now clear.

Suppose $P \sqsubseteq_{\text{BKorf}} Q$ and $h_B(P) \downarrow$. Then $P \downarrow$ and also $Q \downarrow$. Thus $\text{stable}(P)$ if and only if $\text{stable}(Q)$. Because $P \sqsubseteq_{\text{BKorf}} Q$ and $P \downarrow$, $B \cap \text{tr}(P) = \emptyset$ if and only if $B \cap \text{tr}(Q) = \emptyset$. It follows that $\text{stable}(h_B(P))$ if and only if $\text{stable}(h_B(Q))$. If $h_B(P) \uparrow$ and \sim $\text{stable}(h_B(P))$, then either $\sim \text{stable}(P)$ or $B \cap \text{tr}(P) \neq \emptyset$. The former implies \sim $\text{stable}(Q)$. If the latter, then also $B \cap \text{tr}(Q) \neq \emptyset$. Thus \sim $\text{stable}(h_B(Q))$.

In condition 4) the requirement $\text{sfail}(h_B(P)) \subseteq \text{sfail}(h_B(Q))$ is clear by the third formula. Let $(h_B(u), L) \in \text{sfail}(h_B(Q))$. Then $(u, L \cup B) \in \text{sfail}(Q)$ and thus either $(u, L \cup B) \in \text{sfail}(P)$ or $u_1 \in \text{partr}(P)$ for a prefix $u_1$ of $u$. In the former case $(h_B(u), L) \in \text{sfail}(h_B(P))$ and in the latter $h_B(u_1) \in \text{partr}(P)$.

6) Renaming. If $f$ is a renaming function, we denote by $f(u)$ the string obtained from $u$ by applying $f$ to the actions in $u$. In addition, let $B$ be the set of those actions $a$ that are transformed by $f$, i.e. $f(a) \neq a$. We have the relations:

\[
\begin{align*}
\text{stable}(P[f]) &= \text{stable}(P), \\
\text{tr}(P[f]) &= \{f(u) \mid u \in \text{tr}(P)\}, \\
\text{partr}(P[f]) &= \{f(u) \mid u \in \text{partr}(P)\}, \\
\text{sfail}(P[f]) &= \{(f(u), L) \mid \exists(u, M) \in \text{sfail}(P) : L \subseteq C\},
\end{align*}
\]

where

\[
C = (M \cup B) \setminus \{f(a) \mid a \in B, a \notin M\}.
\]

We have to prove only condition 4); the others are obvious. The formula for $\text{sfail}$ becomes more understandable, if we examine the following example. Let $P$ be a process with an action set $A$ and let $Pi$ be a state in $P$ with the outgoing arcs $a_1, \cdots, a_k$, $a_i \neq \tau$, $i = 1, \cdots, k$:

\[
\begin{align*}
\text{Pi} &\xrightarrow{a_1} \cdots \xrightarrow{a_k} \tau
\end{align*}
\]

If $P \Rightarrow Pi$, then $(u, M) \in \text{sfail}(P)$, where

\[
M \subseteq A \setminus \{a_1, \cdots, a_k, \tau\}.
\]
The situation in $P[f]$ is as follows:

$$
\begin{array}{c}
f(a_1) \\
\vdots \\
Pi \\
\vdots \\
f(a_k)
\end{array}
$$

If

$$L \subseteq (M \cup \{a_1, \ldots, a_k\}) \setminus \{f(a_1), \ldots, f(a_k)\},$$

then $(f(u), L) \in \text{sfail}(P[f])$. But also

$$L \subseteq (M \cup B) \setminus \{f(a) \mid a \in B, a \notin M\}$$

and for every such $L$, $(f(u), L) \in \text{sfail}(P[f])$. More precisely, if $b \in L$, then $b \neq f(a)$, $a \in B$, $a \notin M$. Thus $b \neq f(a_i)$, $i = 1, \ldots, k$. All this is also valid, if there are infinitely many transitions from $Pi$.

Now it is clear that $\text{sfail}(P[f]) \subseteq \text{sfail}(Q[f])$. If on the other hand $(f(u), L) \in \text{sfail}(Q[f])$, then there is $(u, M) \in \text{sfail}(Q)$ such that $L \subseteq C$. Now either $(u, M) \in \text{sfail}(P)$ or $u_1 \in \text{partr}(P)$ for a prefix $u_1$ of $u$. In the former case it follows that $(f(u), L) \in \text{sfail}(P[f])$. In the latter case $f(u_1) \in \text{partr}(P[f])$.

7) Parallel composition. A state in $P \mid B \mid R$ can be represented in the form $(Pi, B, Rj)$, where $Pi$ is a state in $P$ and $Rj$ a state in $R$. We usually denote a state simply by $(Pi, Rj)$, if $B$ is known. If $(P \mid B \mid R) \xrightarrow{w} (Pi, Rj)$, then there are subsequences $v$ and $w$ of $u$ such that $P \xrightarrow{v} Pi$ and $R \xrightarrow{w} Rj$. If $u = a_1 a_2 \cdots a_n$ and

$$(P \mid B \mid R) \xrightarrow{a_1} (Pi_1, Rj_1) \xrightarrow{a_2} (Pi_2, Rj_2) \xrightarrow{a_3} \cdots \xrightarrow{a_n} (Pi_n, Rj_n) = (Pi, Rj), \quad (1)$$

then $a_k$ belongs to $v$, if $a_k \notin B$ and $Pi_{k-1} \xrightarrow{a_k} Pi_k$, and $a_k$ belongs to $w$, if $a_k \notin B$ and $R_{j_{k-1}} \xrightarrow{a_k} R_{j_k}$. The element $a_k$ belongs both to $v$ and $w$, if $a_k \in B$.

If $(P \mid B \mid R) \xrightarrow{w} (Pi, Rj)$, then we denote by $\text{parcom}(u; v, w)$ the situation, where $v$ and $w$ are derived from a path (1) using the rules presented above. We have the formulas:

$$
\begin{align*}
\text{stable}(P \mid B \mid R) & = \text{stable}(P) \land \text{stable}(R), \\
\text{tr}(P \mid B \mid R) & = \{u \mid \exists v \in \text{tr}(P), \exists w \in \text{tr}(R) : \text{parcom}(u; v, w)\}, \\
\text{partr}(P \mid B \mid R) & = \left\{u \mid \exists v \in \text{partr}(P), \exists w \in \text{tr}(R) : \text{parcom}(u; v, w)\right\} \cup \\
& \quad \left\{u \mid \exists v \in \text{tr}(P), \exists w \in \text{partr}(R) : \text{parcom}(u; v, w)\right\}, \\
\text{sfail}(P \mid B \mid R) & = \{(u, L) \mid \exists (v, M) \in \text{sfail}(P), \exists (w, N) \in \text{sfail}(R) : \text{parcom}(u; v, w) \land L \cap (B \cup \{\delta\}) \subseteq M \cup N \\
& \quad \land L \setminus (B \cup \{\delta\}) \subseteq M \cap N\}.
\end{align*}
$$

Suppose $P \subseteq BKOref \subseteq Q$. The claim is that $(P \mid B \mid R) \subseteq BKOref (Q \mid B \mid R)$. The inclusions $\text{tr}(P \mid B \mid R) \subseteq \text{tr}(Q \mid B \mid R)$ and $\text{sfail}(P \mid B \mid R) \subseteq \text{sfail}(Q \mid B \mid R)$ are
clear by the formulas. Also condition 3) is clear. Consider now the second part of condition 1).

If \( u \in \text{tr}(Q \mid B \mid R) \), then \( Q \xrightarrow{v} Qi, R \xrightarrow{w} Rj \), \( \text{parcom}(u; v, w) \). Now either
\( v \in \text{tr}(P) \) or \( v_1 \in \text{partr}(P) \) for a prefix \( v_1 \) of \( v \). Suppose \( v \in \text{tr}(P) \). Then \( P \xrightarrow{v} Pk \) for some \( Pk \) and thus \( (P \mid B \mid R) \xrightarrow{v} (Pk, Rj) \). Hence \( u \in \text{tr}(P \mid B \mid R) \). Suppose \( v_1 \in \text{partr}(P) \). Now there is a prefix \( w_1 \) of \( w \) such that \( \text{parcom}(u_1; v_1, w_1) \) for a prefix \( u_1 \) of \( u \). This means \( u_1 \in \text{partr}(P \mid B \mid R) \).

Condition 2) for partr is proved in the same way.

Suppose \( (u, L) \in \text{sfail}(Q \mid B \mid R) \). Then there are \( (v, M) \in \text{sfail}(Q) \) and \( (w, N) \in \text{sfail}(R) \) such that \( \text{parcom}(u; v, w) \). If \( (v, M) \in \text{sfail}(P) \), then \( (u, L) \in \text{sfail}(P \mid B \mid R) \). Otherwise there are prefixes \( v_1 \) of \( v \), \( w_1 \) of \( w \) and \( u_1 \) of \( u \) such that \( v_1 \in \text{partr}(P) \) and \( \text{parcom}(u_1; v_1, w_1) \). Thus \( u_1 \in \text{partr}(P \mid B \mid R) \).

The theorem has now been proved. \( \Box \)

**Proposition 5.2.3**

i) The BKO-refinement coincides with the BKO-equivalence on totally defined processes.

ii) \( \approx_{\text{BKO}} \subset (\subseteq_{\text{BKO ref}} \cap (\subseteq_{\text{BKO ref}})^{-1}) \) properly.

**Proof.**

i) This follows from the fact that \( \text{partr}(P) = \emptyset \) for a totally defined process \( P \).

ii) If \( P \approx_{\text{BKO}} Q \), then clearly \( P \subseteq_{\text{BKO ref}} Q \) and \( Q \subseteq_{\text{BKO ref}} P \). Conversely, consider the processes:

\[
P : \quad \begin{array}{c}
P_1 \xrightarrow{a} P_2 \uparrow \\
\quad \quad \quad \quad P_3 \end{array} \quad Q : \quad Q_1 \xrightarrow{a} Q_2
\]

Then \( \text{partr}(P) = \{\varepsilon, a\} \) and \( \text{partr}(Q) = \{\varepsilon\} \), and so \( P \not\approx_{\text{BKO}} Q \). Furthermore

\[
\text{sfail}(P) = \text{sfail}(Q) = \{(a, \emptyset), (a, \{a\})\};
\]

\[
\text{tr}(P) = \text{tr}(Q) = \{\varepsilon, a\}.
\]

It follows that \( P \subseteq_{\text{BKO ref}} Q \) and \( Q \subseteq_{\text{BKO ref}} P \). \( \Box \)

### 5.3 CFFD-equivalence and refinement relations

CFFD-equivalence (Chaos-Free Failures Divergences equivalence, [Valmari and Tienari 95]) is a restriction of BKO-equivalence in the sense that it also considers divergences. Before introducing the CFFD-relations we define some new concepts. Let \( P = (Q, A, T, q_0, \uparrow) \) be a partially defined labeled transition system.
5.3. CFFD-EQUIVALENCE AND REFINEMENT RELATIONS

1. The set of divergence traces in $P$ is

$$\text{divtr}(P) = \{u \in \text{tr}(P) | q_0 \xrightarrow{u} q \text{ and } \text{Div}(q)\}.$$ 

2. The set of infinite traces is

$$\text{infr}(P) = \{a_1a_2a_3\cdots | a_i \in A \backslash \{\tau\}, a_1a_2\cdots a_n \in \text{tr}(P) \text{ for all } n \geq 1\}.$$ 

**Definition 5.3.1** Processes $P_1$ and $P_2$ are CFFD-(specification) equivalent, $P_1 \approx_{\text{CFFD}} P_2$, if and only if

1. $\text{stable}(P_1)$ if and only if $\text{stable}(P_2)$,
2. $\text{partr}(P_1) = \text{partr}(P_2)$,
3. $\text{infr}(P_1) = \text{infr}(P_2)$,
4. $\text{divtr}(P_1) = \text{divtr}(P_2)$,
5. $\text{sfail}(P_1) = \text{sfail}(P_2)$.

In the case of totally defined processes the new definition of sfail coincides with the original definition presented in [Valmari and Tienari 95] and $\text{partr}(P_1) = \text{partr}(P_2) = \emptyset$. Thus for totally defined processes the extended definition of the CFFD-equivalence agrees with the original one. The condition for infinite traces is necessary, if we want that CFFD-equivalence is congruence with respect to the hiding operator for infinite processes (see [Valmari and Tienari 95] for further explanations).

**Proposition 5.3.1** If $P \approx_{\text{CFFD}} Q$, then $\text{tr}(P) = \text{tr}(Q)$.

*Proof.* Let $u \in \text{tr}(P), u \neq \varepsilon$. If $u \in \text{divtr}(P)$ or $(u, L) \in \text{sfail}(P)$ for some $L$, then $u \in \text{tr}(Q)$. Suppose therefore that $u \not\in \text{divtr}(P)$ and $(u, L) \not\in \text{sfail}(P)$ for all $L$. The only possibility is that $P \xrightarrow{u} s$ and $s \uparrow$ for some state $s$ in $P$. Hence $u \in \text{partr}(P)$ and also $u \in \text{partr}(Q)$. Thus $\text{tr}(P) \subseteq \text{tr}(Q)$. The other inclusion $\text{tr}(Q) \subseteq \text{tr}(P)$ is shown in the same way. $\Box$

**Definition 5.3.2** A process $Q$ is a CFFD-refinement of a process $P$, $P \sqsubseteq_{\text{CFFD-ref}} Q$, if

1. $\text{partr}(P) \subseteq \text{tr}(Q)$ and for every $u \in \text{partr}(Q)$ there is a prefix $u_1$ of $u$ such that $u_1 \in \text{partr}(P)$;
2. 

$$P \Downarrow \implies (\text{stable}(P) \iff \text{stable}(Q)),$$

$$P \Uparrow \implies (\neg \text{stable}(P) \implies \neg \text{stable}(Q)).$$
3. \( \text{infr}(P) \subseteq \text{infr}(Q) \) and for every \( u \in \text{infr}(Q) \), either \( u \in \text{infr}(P) \) or there is a finite prefix \( u_1 \) of \( u \) such that \( u_1 \in \text{partr}(P) \);

4. \( \text{divtr}(P) \subseteq \text{divtr}(Q) \) and for every \( u \in \text{divtr}(Q) \), either \( u \in \text{divtr}(P) \) or there is a prefix \( u_1 \) of \( u \) such that \( u_1 \in \text{partr}(P) \);

5. \( \text{sfail}(P) \subseteq \text{sfail}(Q) \) and for every \((u, L) \in \text{sfail}(Q)\), either \((u, L) \in \text{sfail}(P)\) or there is a prefix \( u_1 \) of \( u \) such that \( u_1 \in \text{partr}(P) \).

**Proposition 5.3.2** If \( P \sqsubseteq_{CFFD_{ref}} Q \), then \( \text{tr}(P) \subseteq \text{tr}(Q) \).

**Proof.** The claim is clear, because for every \( u \in \text{tr}(P) \), either \( u \in \text{partr}(P) \), \( u \in \text{divtr}(P) \) or \((u, L) \in \text{sfail}(P)\) for some action set \( L \). \( \square \)

**Proposition 5.3.3** The CFFD-refinement \( \sqsubseteq_{CFFD_{ref}} \) is a preorder.

**Proof.** Proceeds in the same way as the case of BKO-refinement. Infinite and divergence traces present no problems. \( \square \)

**Theorem 5.3.4** The CFFD-refinement is compositional with respect to all the Lotos operators.

**Proof.** The proof is analogous to the corresponding congruence proof in [Valmari and Tienari 95]. For the most part it has been proved already in the case of BKO-refinement. Here we enlist only the formulas for divergence and infinite traces and prove in detail the congruence with respect to the hiding operator.

1) **Action prefix.** If \( a = \tau \), then \( \text{infr}(a; P) = \text{infr}(P) \) and \( \text{divtr}(a; P) = \text{divtr}(P) \). If \( a \neq \tau \), then

\[
\text{infr}(a; P) = \{aw \mid w \in \text{infr}(P)\}, \\
\text{divtr}(a; P) = \{au \mid u \in \text{divtr}(P)\}.
\]

2) **Choice.**

\[
\text{infr}(P[\mid R]) = \text{infr}(P) \cup \text{infr}(R), \\
\text{divtr}(P[\mid R]) = \text{divtr}(P) \cup \text{divtr}(Q).
\]

3) **Sequential composition.**

\[
\text{infr}(P >> R) = \text{infr}(P) \cup \{uw \mid u \delta \in \text{tr}(P) \land w \in \text{infr}(R)\}, \\
\text{divtr}(P >> R) = \text{divtr}(P) \cup \{uv \mid u \delta \in \text{tr}(P) \land v \in \text{divtr}(R)\}.
\]
4) **Disabling.**

\[
\text{inftr}(P|> R) = \text{inftr}(P) \cup \{uw \mid u \in \text{tr}(P) \land \delta \notin \text{actions}(u) \land w \in \text{inftr}(R)\},
\]

\[
\text{divtr}(P|> R) = \text{divtr}(P) \cup \{uv \mid u \in \text{tr}(P) \land \delta \notin \text{actions}(u) \land v \in \text{divtr}(R)\}.
\]

5) **Hiding.** If \( u \) is a string of actions and \( h_B \) a hiding operator, we denote by \( h_B(u) \) the string obtained from \( u \) by deleting the actions in \( B \). We have the relations:

\[
\text{inftr}(h_B(P)) = \{h_B(u) \mid u \in \text{inftr}(P) \land |h_B(u)| = \infty\},
\]

\[
\text{divtr}(h_B(P)) = \{h_B(u) \mid u \in \text{divtr}(P)\} \cup \{h_B(u) \mid u \in \text{inftr}(P) \land |h_B(u)| < \infty\}.
\]

Condition 3 for \text{inftr} is similar but simpler than condition 4 and that is why we prove only the latter. To prove condition 4) we notice first that \text{inftr}(P) \subseteq \text{inftr}(Q). Thus \text{divtr}(h_B(P)) \subseteq \text{divtr}(h_B(Q)). We are left to prove the second part.

Let \( t \in \text{divtr}(h_B(Q)) \). We can assume that \( t = h_B(u), u \in \text{inftr}(Q), |h_B(u)| < \infty \), because the other alternative is obvious. Now either \( u \in \text{inftr}(P) \) or there is a finite prefix \( v_1 \) of \( u \) such that \( v_1 \in \text{partr}(P) \). Thus either \( h_B(u) \in \text{divtr}(h_B(P)) \) or the prefix \( h_B(v_1) \) of \( h_B(u) \) belongs to \( \text{partr}(h_B(P)) \). Condition 4) has been proved.

6) **Naming.**

\[
\text{inftr}(P[f]) = \{f(w) \mid w \in \text{inftr}(P)\},
\]

\[
\text{divtr}(P[f]) = \{f(u) \mid u \in \text{divtr}(P)\}.
\]

7) **Composition.**

\[
\text{inftr}(P \mid B \mid R) = \{u \mid \exists v \in \text{tr}(P) \cup \text{inftr}(P) : \exists w \in \text{tr}(R) \cup \text{inftr}(R) : \parcom(u;v,w) \land (v \in \text{inftr}(P) \lor w \in \text{inftr}(R))\},
\]

\[
\text{divtr}(P \mid B \mid R) = \{u \mid \exists v \in \text{divtr}(P), \exists w \in \text{tr}(R) : \parcom(u;v,w)\} \cup \{u \mid \exists v \in \text{tr}(P), \exists w \in \text{divtr}(R) : \parcom(u;v,w)\}.
\]

□

**Proposition 5.3.5 i)** The CFFD-refinement coincides with the CFFD-equivalence on totally defined processes.

\[ \approx_{\text{CFFD}} \subset (\subseteq_{\text{CFFD}_{\text{ref}}} \cap (\subseteq_{\text{CFFD}_{\text{ref}}}^{-1})) \text{ properly.} \]

**Proof.**

i) This follows from the fact that \( \text{partr}(P) = \emptyset \) for a totally defined process \( P \).
ii) If $P \approx_{\text{CFFD}} Q$, then clearly $P \subseteq_{\text{CFFD}_{ref}} Q$ and $Q \subseteq_{\text{CFFD}_{ref}} P$. Conversely, consider the processes:

$$
P : \quad \begin{array}{c}
P_1 \uparrow^a \quad P_2 \uparrow^a \\
P_3 \end{array} \quad Q : \quad \begin{array}{c}
Q_1 \uparrow^a \quad Q_2 \\
Q_3 \end{array}
$$

Then $\text{partr}(P) = \{\varepsilon, a\}$ and $\text{partr}(Q) = \{\varepsilon\}$, and so $P \not\approx_{\text{CFFD}} Q$. Furthermore

$$
\text{sfail}(P) = \text{sfail}(Q) = \{(a, \emptyset), (a, \{a\})\};
\text{divtr}(P) = \text{divtr}(Q) = \emptyset;
\text{tr}(P) = \text{tr}(Q) = \{\varepsilon, a\}.
$$

It follows that $P \subseteq_{\text{CFFD}_{ref}} Q$ and $Q \subseteq_{\text{CFFD}_{ref}} P$. □

### 5.4 Remarks

Our examples of failure equivalences have been BKO- and CFFD-equivalences. We have defined the corresponding refinement relations and analysed them in detail (see the next chapter, as well). There are other closely related equivalences which could be redefined for partially defined processes and for which one could also define corresponding refinement relations. Examples of these equivalences are trace, maximal trace and NDFD-equivalences ([Kaivola and Valmari 92], [Kaivola 96]). By using the same kind of technique as in the case of BKO- and CFFD-equivalences it is easy to also define corresponding refinement relations for these other equivalences. For example, the refinement relation corresponding with the trace equivalence is defined using the first two conditions of the BKO-refinement in the definition 5.2.2.
Chapter 6

Comparisons

6.1 Equivalences

We have had four equivalences, weak bisimilarity $\approx_{wbi}$, divergence bisimilarity $\approx_{div}$, BKO-equivalence $\approx_{BKO}$ and CFFD-equivalence $\approx_{CFFD}$. Divergence bisimilarity implies weak bisimilarity, because every divergence bisimulation is also an ordinary bisimulation. The CFFD-equivalence implies the BKO-equivalence. Otherwise, BKO- and CFFD-equivalences are independent of the other two. For example, the following processes are weakly and divergence bisimilar, but not BKO- or CFFD-equivalent:

$$\rightarrow P_1, \quad \rightarrow Q_1 \xrightarrow{\tau} Q_2.$$

The opposite case can be seen from the following diagram (Figure 3 in [Eloranta 94]):

The next example shows the behaviour of $\approx_{CFFD}$ in the presence of partially defined states:
Of course, it is not completely fair to compare the BKO- and CFFD-equivalences with bisimulation equivalences, if we also consider the predicate stable. Fairer comparisons could be achieved, if we dropped the stability condition from the definitions of BKO- and CFFD-equivalences. However, we proceed in the opposite way and modify bisimulation equivalences so that they also contain some kind of condition for stability. This is necessary, if the equivalences are wanted to be congruences with respect to choice and disable operators. In this way we get bisimulation equivalences that imply BKO- and CFFD-equivalences.

It is a well-known idea to use rooted processes and rooted versions of equivalences to make equivalences congruences; see for instance [Bergstra and Klop 85] and [Bergstra, Klop, Olderog 87]. In [Eloranta 94], rooted versions are also used to compare various equivalences. It turns out that rooted versions generalize without difficulties to partially defined labeled transition systems.

Suppose \( P \approx_{\text{wbi.s}} Q \) and let \( \mathcal{R} \) be a bisimulation between \( P \) and \( Q \). We say that a pair \( (P', Q') \) has been derived from \((P, Q)\), if there are chains

\[
P \xrightarrow{a_1} P_1 \xrightarrow{a_2} P_2 \xrightarrow{a_3} \cdots \xrightarrow{a_n} P_n = P',
Q \xrightarrow{a_1} Q_1 \xrightarrow{a_2} Q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_n} Q_n = Q',
\]

such that \( (P_i, Q_i) \in \mathcal{R} \) for all \( i = 1, \ldots, n \).

**Definition 6.1.1** Let \( P, Q \in \text{ALTS} \).

i) **Processes** \( P \) and \( Q \) are weakly rooted bisimilar, \( P \approx_{\text{wbi.s}} Q \), if there is a bisimulation \( \mathcal{R} \) such that \((P, Q) \in \mathcal{R} \) and the following so called root condition is satisfied: for every \((P', Q') \in \mathcal{R} \) derived from \((P, Q)\), either \( P' \neq P \) and \( Q' \neq Q \), or \( P' = P \) and \( Q' = Q \).
ii) Processes $P$ and $Q$ are rooted divergence bisimilar, $P \approx_{\text{rdiv}} Q$, if there is a divergence bisimulation $\mathcal{R}$ such that $(P, Q) \in \mathcal{R}$ and the root condition is satisfied.

We enlist some basic properties of the rooted equivalences defined above. Firstly, it is easy to see that the relations are equivalences. The next proposition clarifies their relationship with the non-rooted versions.

**Proposition 6.1.1** There are the following proper inclusions:

i) $\approx_{\text{rubis}} \subset \approx_{\text{ubis}}$, 
ii) $\approx_{\text{rdiv}} \subset \approx_{\text{div}}$.

**Proof.** The inclusions are valid because of the definitions. It is only necessary to prove that they are proper. The following example is sufficient for this purpose:

$P : \rightarrow P1 \xrightarrow{\tau} P2 \xrightarrow{a} P3, \quad Q : \rightarrow Q1 \xrightarrow{a} Q2.$

$\Box$

The rooted versions of weak and divergence bisimilarities are not congruences with respect to choice and disabling. The counter-example after the theorem 3.3.2 is also valid for the rooted versions in the case of disabling. The next example gives a counter-example for choice:

**Example 1.** Let $P$ and $Q$ be the processes:

Then we have the following relations, where $R$ is a non-trivial process in ALTS:

$P \approx_{\text{ubis}} Q$, \quad P \approx_{\text{div}} Q,$
$P \approx_{\text{rubis}} Q$, \quad P \approx_{\text{rdiv}} Q,$
$P || R \not\approx_{\text{ubis}} Q || R, \quad P || R \not\approx_{\text{div}} Q || R,$
$P || R \not\approx_{\text{rubis}} Q || R, \quad P || R \not\approx_{\text{rdiv}} Q || R.$
\[\square\]

It seems as if \(\approx_{\text{rubis}}\) and \(\approx_{\text{div}}\) are of no use, but as a matter of fact they are reasonable equivalences on a large subset of ALTS. Let RALTS consists of processes \(P \in \text{ALTS}\) having no incoming transitions to the initial state of \(P\). Now \(\approx_{\text{rubis}}\) and \(\approx_{\text{div}}\) are both congruences on RALTS with respect to all the Lotos operators. We don’t present the easy proof of this result, but show the inclusions instead.

**Theorem 6.1.2** We have the following proper inclusions in ALTS:

i) \(\approx_{\text{div}} \subset \approx_{\text{CFD}}\),

ii) \(\approx_{\text{div}} \subset \approx_{\text{wbis}}\),

iii) \(\approx_{\text{rubis}} \subset \approx_{\text{BKO}}\).

*Proof.*

i) Suppose \(P \approx_{\text{div}} Q\). We prove that also \(P \approx_{\text{CFD}} Q\). The conditions for stable, infr, and divtr have been proved in theorem 18 in [Eloranta 94]. We must check the claim for partr and sfail. Suppose \(u \in \text{partr}(P)\). Then for some \(P'\), \(P \xrightarrow{u} P'\) and \(P' \uparrow\). Because \(P \approx_{\text{div}} Q\), it follows by induction that there is a \(Q'\) such that \(Q \xrightarrow{u} Q'\) and \(P' \approx_{\text{div}} Q'\). Because \(P' \uparrow\), then also \(Q' \uparrow\). Thus there is a \(Q''\) such that \(Q \Rightarrow Q''\) and \(Q'' \uparrow\). Thus \(u \in \text{partr}(Q)\). By symmetry, we can also conclude that \(\text{partr}(Q) \subset \text{partr}(P)\). Hence \(\text{partr}(P) = \text{partr}(Q)\).

Suppose next that \((u, L) \in \text{sfail}(P)\). We can prove the property \((u, L) \in \text{sfail}(Q)\) in the same way as in the theorem 18 in [Eloranta 94] in spite of the differences in the definition of sfail.

ii) This is clear, because \(\approx_{\text{div}}\) implies \(\approx_{\text{wbis}}\) and \(\approx_{\text{rubis}} \approx_{\text{wbis}}\).

iii) Use appropriate parts from i). \(\square\)

Next we define the bisimulation congruence. It turns out that we can apply ideas from the ordinary bisimulation theory, see [Milner 89], ch. 7. The only difficulty is how to take the partiality predicate along with the definition. For example, the following processes are bisimilar with the root condition satisfied:
And the same is true for processes

\[
\begin{array}{c}
P_1 \\
\downarrow \\
P_2 \\
\downarrow \\
P_3^{\uparrow} \\
\downarrow \\
P_4
\end{array}
\begin{array}{c}
Q_1 \\
\downarrow \\
Q_2^{\uparrow} \\
\downarrow \\
Q_3
\end{array}
\]

It would be attractive, if bisimulation congruence were an extension of rooted bisimulation equivalence. In other words, bisimulation congruence would be defined on all of ALTS and it would agree with $\approx_{\text{wbis}}$ on RALTS. With these motivations in mind we give the following definition.

**Definition 6.1.2** Let $P, Q \in \text{ALTS}$.

a) Processes $P$ and $Q$ are weak bisimulation congruent, $P \approx_{\text{wbis}} Q$, if $P \uparrow$ if and only if $Q \uparrow$ and for every $a \in A$,

- $P \xrightarrow{a} P'$ implies that there is a $Q'$ such that $Q \xrightarrow{a} Q'$ and $P' \approx_{\text{wbis}} Q'$;
- $Q \xrightarrow{a} Q'$ implies that there is a $P'$ such that $P \xrightarrow{a} P'$ and $P' \approx_{\text{wbis}} Q'$.
b) Processes $P$ and $Q$ are divergence congruent, $P \approx_{\text{div}}^c Q$, if $P \uparrow$ if and only if $Q \uparrow$ and for every $a \in A$ the two properties in $a$ are valid with $\approx_{\text{bis}}$ replaced by $\approx_{\text{div}}$.

**Proposition 6.1.3** The relations $\approx_{\text{bis}}^c$ and $\approx_{\text{div}}^c$ are equivalences.

*Proof.* Essentially the proof of Lemma 77 in [Eloranta 94]. □

**Proposition 6.1.4** The equivalences $\approx_{\text{bis}}^c$ and $\approx_{\text{div}}^c$ are congruences with respect to the Lotos operators.

*Proof.* See Theorem 97 in [Eloranta 94]. We need only to check the properties of the partiality and convergence predicates, but that is quite straightforward. □

Next, we show the relationship of the new congruences to the original equivalences $\approx_{\text{bis}}$, $\approx_{\text{div}}$ and $\approx_{\text{CFFD}}$.

**Lemma 6.1.5** For all $a \in (A \setminus \{\tau\}) \cup \{\varepsilon\}$,

i) if $P \approx_{\text{bis}}^c Q$ or $P \approx_{\text{div}}^c Q$, then $P \uparrow a$ if and only if $Q \uparrow a$;

ii) if $P \approx_{\text{div}}^c Q$, then Conv$(P, a)$ if and only if Conv$(Q, a)$.

*Proof.* i) Suppose $P \uparrow a$. Then either $P \uparrow$ or there is a $P'$ such that $P \xrightarrow{a} P'$ and $P' \uparrow$. In the former case, also $Q \uparrow$ and thus $Q \uparrow a$. In the latter case there is a $Q'$ such that $Q \xrightarrow{a} Q'$ and $P' \approx_{\text{bis}} Q'$. Hence $Q' \uparrow$ and thus $Q \uparrow a$. By the symmetry of an equivalence relation, we have proved the part i).

ii) In this case we can proceed similarly as above. □

**Proposition 6.1.6** We have the following proper inclusions:

i) $\approx_{\text{bis}}^c \subset \approx_{\text{bis}}$,

ii) $\approx_{\text{div}}^c \subset \approx_{\text{div}}$.

*Proof.* To prove the inclusions, it is enough to show that if $P \approx_{\text{bis}}^c Q$ or $P \approx_{\text{div}}^c Q$, then the relations

$$\mathcal{R}_1 = \{(P, Q) \cup \{S, T \mid S \approx_{\text{bis}} T\}\},$$

$$\mathcal{R}_2 = \{(P, Q) \cup \{S, T \mid S \approx_{\text{div}} T\}\}$$

are bisimulation and divergence bisimulation, respectively. This is done for the most part in the proof of Lemma 76 in [Eloranta 94]. It is only necessary to check the conditions for $\uparrow$ and Conv, but with the help of the previous lemma this is straightforward. In addition, the inclusions are proper; the examples in the proof of Proposition 6.1.1 are valid in this context, as well. □
**Proposition 6.1.7** We have the following proper inclusions:

i) \( \cong_{\text{div}} \subset \cong_{CFFD} \),

ii) \( \cong_{\text{div}} \subset \cong_{\text{wbi}^*} \),

iii) \( \cong_{\text{wbi}^*} \subset \cong_{BKO} \).

**Proof.** i) See Lemma 89 in [Eloranta 94].

ii) This is evident, because every divergence bisimulation is also ordinary weak bisimulation.

iii) Use appropriate parts from i). □

The next diagram shows the relationship of all the equivalences. An arrow \( E_1 \rightarrow E_2 \) means that the equivalence \( E_1 \) implies \( E_2 \).

![Diagram](image)

Finally we present a simple fact which is important in verifications.

**Proposition 6.1.8** Let \( \cong \) be one of the previous equivalences. If \( P \cong Q \), then \( P \) is totally defined if and only if \( Q \) is.

**Proof.** Directly from the definitions. □

### 6.2 Refinement relations

We can add the root condition to prebisimulation, partial bisimulation, divergence prebisimulation and partial divergence bisimulation to get the rooted versions of the bisimulation-based refinement relations. These rooted versions are not compositional with respect to all the Lotos operators in ALTS, but in RALTS they are. The rooted refinement relations \( \sqsubseteq_{\text{bisref}}, \sqsubseteq_{\text{pbi}^*}, \sqsubseteq_{\text{divref}} \) and \( \sqsubseteq_{\text{pdivref}} \) behave in the
same way as their unrooted counterparts in the propositions 3.3.2, 3.4.1 and 5.1.3. Thus

(i) \( E_{\text{bisref}} \) and \( E_{\text{prbisref}} \) agree with \( r_{\text{rwbis}} \) on totally defined processes;

(ii) \( E_{\text{bisref}} \subseteq E_{\text{prbisref}} \) properly;

(iii) \( \approx_{\text{bisref}} \subseteq \approx_{\text{prbisref}} \) properly, where \( \approx_{\text{bisref}} = E_{\text{bisref}} \cap (E_{\text{bisref}})^{-1} \) and \( \approx_{\text{prbisref}} = E_{\text{prbisref}} \cap (E_{\text{prbisref}})^{-1} \);

(iv) \( r_{\text{rwbis}} \subseteq \approx_{\text{bisref}} \) properly;

and ditto for \( E_{\text{divref}} \) and \( E_{\text{prdivref}} \).

We can also develop refinement relations for the bisimulation congruence and divergence congruence.

**Definition 6.2.1** A process \( Q \) is congruent bisimulation refinement of a process \( P \), \( P \subseteq_{\text{wabisref}} Q \), if the following holds for all \( a \in A \):

i) \( P \xrightarrow{a} P' \) implies there is a \( Q' \) such that \( Q \xrightarrow{a} Q' \) and \( P' \subseteq_{\text{wabisref}} Q' \);

ii) If \( P \not\downarrow a \), then \( Q \not\downarrow a \) and \( Q \xrightarrow{a} Q' \) implies there is a \( P' \) such that \( P \xrightarrow{a} P' \) and \( P' \subseteq_{\text{wabisref}} Q' \).

A process \( Q \) is congruent divergence refinement of a process \( P \), \( P \approx_{\text{div}} Q \), if the previous conditions i) and ii) hold for every \( a \in A \) with \( E_{\text{wabisref}} \) replaced by \( E_{\text{divref}} \).

Again, the difficult part in the previous definition was the partiality predicate: should it be in the form \( \downarrow \) or \( \downarrow a \). Consider the following processes \( P \) and \( Q \):

\[
\begin{array}{c}
P_1 \downarrow \\
a \downarrow \\
P_2 \uparrow \\
b \downarrow \\
P_3 \\
\end{array}
\quad
\begin{array}{c}
Q_1 \xrightarrow{a} Q_4 \uparrow \\
a \downarrow b \\
Q_2 \downarrow \\
Q_3 \downarrow \\
P_3 \downarrow \\
\end{array}
\]

Now \( P \subseteq_{\text{wabisref}} Q \) and \( P \subseteq_{\text{bisref}} Q \). We would like it, if \( \subseteq_{\text{wabisref}} \) agreed with \( \subseteq_{\text{bisref}} \) in RALTS. But if there were \( \uparrow \) instead of \( \uparrow a \) in the definition, then \( P \downarrow \) and it would be necessary to find a counterpart in \( P \) for \( Q_2 \), which is impossible. On the other
hand, the example also shows some of the counter-intuitive aspects of bisimulation refinements!

The refinement relations just defined have the same properties with respect to bisimulation and divergence congruences as bisimulation and divergence refinements had with respect to weak bisimulation and divergence equivalences. First, the relations $\subseteq^e_{\text{wibis}}$ and $\subseteq^e_{\text{div}}$ are preorders. Furthermore, they are compositional with respect to all the Lotos operators. Basically, the proof is similar to the proof of the theorem 97 in [Eloranta 94]. It is necessary to check only the additional requirement for $\downarrow$. But this follows directly from the definitions of the operators and the predicate $\downarrow$ itself given in the section 2.1.

**Proposition 6.2.1** The relations $\subseteq^e_{\text{wibis}}$ and $\subseteq^e_{\text{div}}$ have the following properties:

i) $\subseteq^e_{\text{wibis}}$ and $\subseteq^e_{\text{div}}$ agree with $\approx_{\text{wibis}}$ and $\approx_{\text{div}}$, respectively, on totally defined systems.

ii) $\subseteq^e_{\text{wibis}} \subseteq \subseteq^e_{\text{wibis}}$ properly.

iii) $\subseteq^e_{\text{div}} \subseteq \subseteq^e_{\text{div}}$ properly.

**Proof.** i) This follows directly from the fact that $\subseteq^e_{\text{wibis}}$ agrees with $\approx_{\text{wibis}}$ and $\subseteq^e_{\text{div}}$ with $\approx_{\text{div}}$ on totally defined processes.

ii) Suppose $P \subseteq^e_{\text{wibis}} Q$ and $P \xrightarrow{a} P'$ for some $a \in (A \setminus \{\tau\}) \cup \{\varepsilon\}$. Then either $P \xrightarrow{\tau} P'' \xrightarrow{a} P'$ or $P \xrightarrow{a} P'' \xrightarrow{a} P'$ for some $P''$. By the definition of $\subseteq^e_{\text{wibis}}$, there is a $Q'$ such that $Q \xrightarrow{\tau} Q'$ or $Q \xrightarrow{a} Q'$ and $P'' \subseteq^e_{\text{wibis}} Q'$. It follows that there is also a $Q'$ such that $Q \xrightarrow{a} Q$ and $P' \subseteq^e_{\text{wibis}} Q'$. Suppose $P \downarrow a$. Then also $Q \downarrow a$. If now $Q \xrightarrow{a} Q'$, then we again have two possibilities: $Q \xrightarrow{a} Q' \xrightarrow{a} Q'$ or $Q \xrightarrow{a} Q' \xrightarrow{a} Q'$. Thus there is a $P''$ such that $P \xrightarrow{a} P''$ or $P \xrightarrow{a} P''$ and $P'' \subseteq^e_{\text{wibis}} Q''$. In the case $P \rightarrow P''$ we must have $P \downarrow a$ and in the case $P \rightarrow P''$ we have $P \downarrow a$. It follows that also $Q'' \downarrow a$ or $Q'' \downarrow a$. Hence there is a $P'$ such that $P \rightarrow P'$ and $P' \subseteq^e_{\text{wibis}} Q'$.

The subset relation between the refinement relations is proper, because $\subseteq^e_{\text{wibis}}$ agrees with $\approx_{\text{wibis}}$ on totally defined systems and $\approx_{\text{wibis}}$ is properly contained in $\approx_{\text{wibis}}$.

iii) The proof proceeds in the same way as in ii). The property for Conv follows directly from the definitions. □

Our main goal in this chapter is to classify the refinement relations. As every refinement relation agrees with the corresponding equivalence relation on totally defined systems, we can take the inclusion diagram for the equivalences as our starting point. Thus we obtain the following diagram:
Theorem 6.2.2  The inclusions in the diagram are valid. There are no other inclusions between the refinement relations in the diagram.

Proof. The inclusions 1 - 16 follow directly from the definitions. The proofs of the inclusions 19 and 20 are similar but simpler than the proofs of the inclusions 17 and 18. We prove the inclusions 17 and 18 in detail and start with the inclusion 17.

Suppose $P \sqsubseteq \text{rdioref} Q$ and $\mathcal{R}$ is a divergence prebisimulation between $P$ and $Q$ with the root condition satisfied. We prove the conditions for $P \sqsubseteq \text{CFDFref} Q$.

Condition 1. Let $u \in \text{partr}(P)$. Then $P \xrightarrow{u} P'$, where $P'$ is a partially defined state. By induction on the length of $u$ (see Lemma 17 in [Eloranta 94]) there is a $Q'$ such that $Q \xrightarrow{u} Q'$ and $(P', Q') \in \mathcal{R}$. Thus $u \in \text{tr}(Q)$. Conversely, let $v \in \text{partr}(Q)$. Suppose $v = \varepsilon$ or $v = v_1 \cdots v_n$, $n \geq 1$, $v_i \in \mathcal{A} \setminus \{\tau\}$. Because $v \in \text{partr}(Q)$, there is a $Q'$ such that $Q \xrightarrow{u} Q'$ and $Q' \uparrow$. If $P \uparrow v_1$ (or $P \uparrow \varepsilon$), then $v_1 \in \text{partr}(P)$ (or $\varepsilon \in \text{partr}(P)$) and the condition is satisfied. If $P \downarrow v_1$, then $Q \xrightarrow{v_1} Q_1$ implies there
is a $P_1$ such that $P \xrightarrow{\tau} P_1$ and $(P_1, Q_1) \in \mathcal{R}$. Proceeding in this way we find that either $P \xrightarrow{\tau} P'$ and $(P', Q') \in \mathcal{R}$ for some $P'$ or there is a $k$, $1 \leq k \leq n$, such that $P^n \xrightarrow{\tau} P'$ and $P' \uparrow$. In the former case $P' \uparrow$, because in the other alternative we should also have $Q \downarrow$, a contradiction. Thus in both cases $P' \uparrow$ and condition 1 is satisfied.

**Condition 2.** Suppose $P \downarrow$. If $P \rightarrow P'$ and $P \neq P'$, then there is a $Q'$ such that $Q \rightarrow Q'$ and $(P', Q') \in \mathcal{R}$. Because the root condition is satisfied, $Q \neq Q'$ and thus $\sim$ stable $(P)$ and $\sim$ stable $(Q)$. If $P \xrightarrow{\tau} P'$, then $\sim$ stable $(P)$ and Div $(P)$. It follows from condition 2 in the definition 5.1.2 of the divergence refinement that also Div $(Q)$. Thus also $\sim$ stable $(Q)$. Suppose now that $P$ is stable. If $Q \rightarrow Q'$ and $Q \neq Q'$, there should exist a $P''$ such that $P \rightarrow P''$ and $P \neq P'$, because $P \downarrow$ and the root condition is satisfied. But this would be a contradiction. Hence also stable $(Q)$. If $Q \xrightarrow{\tau} Q'$, then Div $(Q)$. But $P \downarrow$ and condition 3.b) in definition 5.1.2 of the divergence refinement implies Div $(P)$, a contradiction. Hence $Q$ is stable.

Assume $P \uparrow$ and $\sim$ stable $(P)$. If $P \rightarrow P'$ and $P \neq P'$, then it follows again from the properties of the prebisimulation $\mathcal{R}$ and the root condition that $\sim$ stable $(Q)$. If $P \xrightarrow{\tau} P'$, then Div $(P)$ and hence also Div $(Q)$ and thus $\sim$ stable $(Q)$. Condition 2 has been proved.

**Condition 3.** Similar to condition 4.

**Condition 4.** We have divtr $(P) \subset$ divtr $(Q)$, because of Lemma 17 in [Eloranta 94] and condition 2 in the definition of the divergence refinement. If $u \in$ divtr $(Q)$, then $Q \xrightarrow{\tau} Q'$ and Div $(Q')$ for some $Q'$. Reasoning in the same way as in the proof of condition 1 and taking condition 3.b) in the definition of the divergence refinement into account, we can conclude that either $u \in$ divtr $(P)$ or $u_1 \in$ partr $(P)$ for some prefix $u_1$ of $u$. Condition 3 has thus been proved.

**Condition 5.** Let $(u, L) \in$ sfail $(P)$. Then for some $P'$, $P \xrightarrow{\tau} P'$, $P' \downarrow$ and $P' \xrightarrow{a/} a$ for all $a \in L \cup \{\tau\}$. As in the previous conditions we can deduce that for some $Q'$, $Q \xrightarrow{\tau} Q'$ and $(P', Q') \in \mathcal{R}$. Now $P' \downarrow a$ for all $a \in L \cup \{\tau\}$. If $Q \xrightarrow{\tau} Q'$, $a \in L$, then by condition 3.c) in the definition of the divergence refinement we should also have $P' \xrightarrow{a/} P''$ for some $P''$, a contradiction. Thus $Q' \xrightarrow{\tau} a$ for all $a \in L$. Suppose $Q' \xrightarrow{\tau} a$. We have Conv $(P')$ and from this and condition 3.b) we can deduce that also Conv $(Q')$. Thus there can be only finite chains $Q' \xrightarrow{\tau} Q_1 \xrightarrow{\tau} Q_2 \xrightarrow{\tau} \cdots \xrightarrow{\tau} Q_n$. Take one such chain and call its final state $Q_n$. If $Q' \xrightarrow{\tau} Q''$ for some $a \in L$, then $Q' \xrightarrow{a/} Q''$. As earlier we should now have $P' \xrightarrow{\tau} P''$ for some $P''$, a contradiction. Thus $Q' \xrightarrow{\tau} a$ for all $a \in L \cup \{\tau\}$. We can conclude that there is a $Q''$ such that $Q \xrightarrow{\tau} Q''$ and $Q' \xrightarrow{\tau} a$ for all $a \in L \cup \{\tau\}$. If $Q' \uparrow$, then $Q \uparrow$, a contradiction. Thus $Q \downarrow$ and $(u, L) \in$ sfail $(Q)$. We have proved sfail $(P) \subset$ sfail $(Q)$.

If $(v, L) \in$ sfail $(Q)$, then $Q \xrightarrow{\tau} Q'$, $Q \downarrow$ and $Q' \xrightarrow{a/} a$ for all $a \in L \cup \{\tau\}$. Especially, we have $Q \downarrow a$ for all $a \in L \cup \{\tau\}$. Write $v = v_1 \cdots v_n$, if $v \neq \varepsilon$. As in the proof of condition 1, either $P \xrightarrow{\tau} P$ and $(P', Q') \in \mathcal{R}$ or $P^n \xrightarrow{\tau} P''$ and $P'' \uparrow$. In the latter case the second part of condition 4 is satisfied. Consider the former case. There can’t
be transitions $P' \xrightarrow{a} P$, $a \in L$, because $Q' \xrightarrow{a} P$ and $(P', Q') \in R$. Because $\text{Conv}(Q')$, we also have $\text{Conv}(P')$. Thus there are no infinite $\tau$-paths starting from $P'$. If $P' \uparrow$, condition 4 is satisfied. So assume $P' \downarrow$. Take one of the $\tau$-paths $P'' \Longrightarrow P''$ such that $P'' \xrightarrow{a} P$. Now $P'' \downarrow$ and $P'' \xrightarrow{a}$ for all $a \in L \cup \{\tau\}$. Thus $P \xrightarrow{a} P''$ and $(v, L) \in \text{sfail}(P)$. Condition 5 has been proved and consequently $P \subseteq \text{CFD}_{\text{ref}} Q$.

Next we prove inclusion 18. Suppose $P \subseteq^e \text{div}_{\text{ref}} Q$. We claim that also $P \subseteq \text{CFD}_{\text{ref}} Q$.

\textbf{Condition 1.} Let $u \in \text{partr}(P)$. As in the case of $\subseteq^e_{\text{div}_{\text{ref}}}$ it follows that $u \in \text{tr}(Q)$. Suppose conversely that $v \in \text{partr}(Q)$. Write $v = v_1 \cdots v_n$, $n \geq 1$, if $v \neq \varepsilon$. There is a $Q'$ such that $Q \xrightarrow{v} Q'$ and $Q' \uparrow$. If $v = \varepsilon$ and $Q = Q'$, then $Q \uparrow$. Now $P \downarrow$ would lead to a contradiction and consequently $P \uparrow$, i.e. $\varepsilon \in \text{partr}(P)$. If on the other hand $Q \neq Q'$, then the first transition from $Q$ can be either $Q \xrightarrow{\tau} Q''$ or $Q \xrightarrow{v} Q'$. Let $a$ be the first action, i.e. $a = v_1$ or $a = \tau$. Now either $P \uparrow a$ or $P \downarrow a$. If $P \uparrow a$, then condition 1 for $\subseteq \text{CFD}_{\text{ref}}$ is satisfied. If $P \downarrow a$, then there is a $P''$ such that $P \xrightarrow{a} P''$ and $P'' \subseteq \text{div}_{\text{ref}} Q''$. Now we can proceed exactly in the same way as in the proof of the inclusion $\subseteq^e_{\text{div}_{\text{ref}}} \subseteq \text{CFD}_{\text{ref}}$.

\textbf{Condition 2.} Suppose $P \downarrow$. If $P \xrightarrow{\tau} P'$, then $Q \Longrightarrow \xrightarrow{\tau} Q'$, i.e. $\sim \text{stable}(P)$ implies $\sim \text{stable}(Q)$. If stable($P$), then $P \downarrow$. Now $\sim \text{stable}(Q)$ would lead to a contradiction on account of condition ii) in the definition of the congruent divergence refinement. Thus stable($Q$) and the first part of condition 2 is valid.

Suppose next $P \uparrow$ and $\sim \text{stable}(P)$. It follows directly from the definition of $\subseteq^e_{\text{div}_{\text{ref}}}$ that also $\sim \text{stable}(Q)$. The condition 2 has been proved.

\textbf{Conditions 3.} and 4. The proof proceeds exactly as the proof of the condition 4 for $\subseteq^e_{\text{div}_{\text{ref}}}$ and $\subseteq \text{CFD}_{\text{ref}}$.

\textbf{Condition 5.} Let $(u, L) \in \text{sfail}(P)$. If $u \neq \varepsilon$, we can show $\text{sfail}(P) \subseteq \text{sfail}(Q)$ in the same way as in the proof of $\subseteq_{\text{div}_{\text{ref}}} \subseteq \text{CFD}_{\text{ref}}$. If $u = \varepsilon$, then for some $P'$, $P \Longrightarrow P'$, $P' \downarrow$, and $P' \xrightarrow{a} P''$ for all $a \in L \cup \{\tau\}$. The case $P \neq P'$ is essentially the same as the case $u \neq \varepsilon$. So suppose $P = P'$. We have then $P \downarrow a$ for all $a \in L \cup \{\tau\}$. Furthermore, $Q \downarrow a$ and for every transition $Q \xrightarrow{a} Q'$ there should be a derivation $P \xrightarrow{a} P''$, a contradiction. Thus $(u, L) \in \text{sfail}(Q)$.

Suppose $(v, L) \in \text{sfail}(Q)$. If $v \neq \varepsilon$, then we can proceed in the same way as in the proof of $\subseteq_{\text{div}_{\text{ref}}} \subseteq \text{CFD}_{\text{ref}}$. If $v = \varepsilon$ and $P \xrightarrow{a} P'$ for some $a \in L \cup \{\tau\}$, then we should have $Q \Longrightarrow \xrightarrow{a} Q'$, a contradiction. Thus $(v, L) \in \text{sfail}(P)$ and the whole claim has now been proved.

That there are no other other inclusions follows either from definitions or from the inclusion diagram for the equivalences. \(\Box\)
Proposition 6.2.3 In RALTS, $\sqsubseteq_{ubisref}^c = \sqsubseteq_{bisref}$ and $\sqsubseteq_{divef}^c = \sqsubseteq_{divref}.$

Proof. The conditions for transitions are proved in Theorem 81 in Eloranta [2]. The conditions for the partiality predicate are the same in both definitions. The same is true for the function Conv. \(\square\)

The next proposition generalizes a property which was already used in section 4.1.

Proposition 6.2.4 Let $Q$ be a partially defined labeled transition system interpreted as a graph. If $P$ is obtained from $Q$ by cutting off some arcs and marking the states left at the cut-off points partially defined, then $P \sqsubseteq Q$ for all the refinement relations $\sqsubseteq.$

Proof. Let $Q_1, \ldots, Q_n$ be the states in $P$ and set $\mathcal{R} = \{(Q_i, Q_i) \mid i = 1, \ldots, n\}.$ Then $\mathcal{R}$ is a relation between processes. It contains the initial states of $P$ and $Q$ and it is evident that $\mathcal{R}$ is prebismulation, divergence prebismulation, partial bismulation and partial divergence bismulation. Also the root condition and congruence condition are satisfied. Thus $P \sqsubseteq Q$ for all the bismulation-based refinement relations $\sqsubseteq.$ As for CFFD-refinement, the inclusions in the definition are clearly valid and the other requirements are fulfilled on account of the definition of $P.$ \(\square\)
Chapter 7

Algorithmic Aspects

The starting point for bisimulation-based relations is the fixpoint algorithm for strong bisimilarity. Its best source is [Fernandez 89], which is based on [Paige and Tarjan 87]. Earlier results are [Kanellakis and Smolka 83], [Bolognesi and Smolka 87] and [Bolognesi and Canev 88]. It is possible to use similar ideas also when computing bisimulation-based refinement relations.

The algorithm for determining the CFFD-equivalence, on the other hand, uses the methods developed for testing the trace equivalence of nondeterministic finite automata ([Hopcroft and Ullman 79], pp. 22-23, [Aho, Hopcroft and Ullman 74], pp. 157-162). In the case of CFFD-equivalence, we must also take divergences, stable failures and partial traces into account. This can be done by equipping the nodes with attributes that reserve the necessary information ([Hennessy 85], [Cleaveland and Hennessy 90], [Cleaveland, Parrow and Steffen 90], [Brinksma 89], [Valmari and Tienari 91]).

7.1 Bisimulation-based equivalences

We first present an algorithm to detect strongly bisimilar states in one finite process $P = (S, A, T, \uparrow, s_0)$. The algorithm computes the partition $\rho$ of the state set $S$ iteratively. At the end the partition $\rho$ represents the equivalence classes of the states. The computation starts by partitioning the initial partition $\rho_I = \{B_1, B_2\}$, where

$$B_1 = \{s \in S | s \uparrow\}, \quad B_2 = \{s \in S | s \downarrow\}.$$

The algorithm uses the notation

$$T^{-1}_a[B] = \{s \in S | (s, a, s') \in T, \ s' \in B\},$$

where $B \subset S$. 

83
Algorithm: Strong bisimilarity.

Input: A finite process \( P = (S, A, T, \uparrow, s_0) \in \text{ALTS}. \)

Output: Partition \( \rho \) of \( S; \rho \) represents equivalent states.

Method:

1. \textbf{begin}
2. \hspace{1em} \( W := \rho; \hspace{1em} \rho := \rho; \)
3. \hspace{1em} \textbf{repeat}
4. \hspace{2em} \text{choose and remove any } B \in W;
5. \hspace{2em} \textbf{for} each \( a \in A \) \textbf{do}
6. \hspace{3em} \( I_{a,B} := \{ X \in \rho \mid X \cap T_{a}^{-1}[B] \neq \emptyset, X \not\subseteq T_{a}^{-1}[B] \} ; \)
7. \hspace{3em} \( I_{a,B}^{1,2} := \{ X \cap T_{a}^{-1}[B] \mid X \in I_{a,B} \} \cup \{ X \setminus T_{a}^{-1}[B] \mid X \in I_{a,B} \} ; \)
8. \hspace{3em} \( \rho := (\rho \setminus I_{a,B}) \cup I_{a,B}^{1,2} ; \)
9. \hspace{3em} \( W := (W \setminus I_{a,B}) \cup I_{a,B}^{1,2} ; \)
10. \hspace{2em} \textbf{endfor}
11. \hspace{1em} \textbf{until} \( W = \emptyset ; \)
12. \hspace{1em} \textbf{end.}

For \( a \in A, B \subseteq S \) and a partition \( \rho \) of \( S \), define the operator \( \Phi_{a,B}(\rho) \) as follows:

\[
\Phi(\rho) = \{ X \cap T_{a}^{-1}[B] \mid X \in \rho \} \cup \{ X \setminus T_{a}^{-1}[B] \mid X \in \rho \} .
\]

Now the lines 4-9 compute the operator \( \Phi(\rho, \rho) \), which is defined as follows. Let \( \rho' = \{ B_1, \ldots, B_n \} \) be a family of subsets of the state set \( S \) and \( \rho \) a partition of \( S \). Then

\[
\Phi(\rho', \rho) = (\Phi_{B_1} \circ \cdots \circ \Phi_{B_n})(\rho),
\]

where \( \Phi_B = \Phi_{a_1,B} \circ \cdots \circ \Phi_{a_n,B} \), if \( a_1, \ldots, a_n \) are the elements of \( A \). The algorithm computes the maximal fixpoint of \( \Phi(\rho) = \Phi(\rho, \rho) \) and this turns out to be the maximal bisimulation. The correctness proof of the algorithm is long and tedious, but at the same time elegant, [Fernandez 89]. The algorithm can be implemented in \( O(nm) \) time, where \( n \) is the number of states and \( m \) the number of arcs. By carefully keeping track of how blocks of the partition are split into subblocks at each refinement step, it is possible to reduce the time to \( O(m \log n) \) ([Fernandez 89], [Paige and Tarjan 87]).
7.2. BISIMULATION REFINEMENTS

It is now straightforward to test for two finite processes $P$ and $Q$, whether $P \approx_{\text{wbi}} Q$ or not. This can be done by forming the process $R = \pi; P \| \pi; Q$ and testing, if the initial states of $P$ and $Q$ are equivalent.

The same algorithm can be used for computing the weak and divergence bisimulation equivalences. However, instead of $P = (S, A, T, \uparrow, s_0)$ we must consider the system $\hat{P} = (S, A, \hat{T}, \uparrow, s_0)$, where

$$(s_1, a, s_2) \in \hat{T} \text{ if and only if } s_1 \overset{a}{\rightarrow} s_2.$$  

For weak bisimilarity, the initial partition is $\rho_I = \{B_1, B_2\}$, where

$B_1 = \{s \in S | s \uparrow\}$,  $B_2 = \{s \in S | s \downarrow\}$.  

For divergence bisimilarity, the initial partition is $\rho = \{B_1, B_2, B_3, B_4\}$, where

$B_1 = \{s \in S | s \downarrow \text{ and } \text{Conv}(s)\}$,  

$B_2 = \{s \in S | s \downarrow \text{ and } \text{Div}(s)\}$,  

$B_3 = \{s \in S | s \uparrow \text{ and } \text{Conv}(s)\}$,  

$B_4 = \{s \in S | s \uparrow \text{ and } \text{Div}(s)\}$.  

The most-time consuming task is the computation of $\hat{T}$. In practice the most efficient way is not to compute $\hat{T}$ beforehand, but on the fly. In other words, when computing $\hat{T}_{\alpha}^{-1}[B]$, the best way seems to be the backward depth-first search in the original transition system $P$ to detect the states $s$ such that $s \overset{\alpha}{\rightarrow} s'$, $s' \in B$. Because of the time-consuming computations of the sets $\hat{T}_{\alpha}^{-1}[B]$, it is not necessary to use the most efficient version of the equivalence algorithm, but the basic version we have presented is sufficient for most purposes.

### 7.2 Bisimulation refinements

The computation of the relation $\sqsubseteq_{\text{wbi/ref}}$ is based on the same kind of principles as the computation of $\approx_{\text{wbi}}$. First we define an operator $\mathcal{F}$ on binary relations on ALTS. If $R \subseteq \text{ALTS} \times \text{ALTS}$ is a relation, then $\mathcal{F}(R)$ consists of those pairs $(P, Q) \in R$ such that

1. if $P \overset{a}{\rightarrow} P'$, $a \in (A \setminus \{\tau\}) \cup \{\varepsilon\}$, then there is a $Q'$ such that $Q \overset{a}{\rightarrow} Q'$ and $(P', Q') \in R$,

2. if $P \downarrow a$, then $Q \downarrow a$ and for every $Q'$, $Q \overset{a}{\rightarrow} Q'$ there is a $P'$ such that $P \overset{a}{\rightarrow} P'$ and $(P', Q') \in R$.

The algorithm is based on the following theorem ([Stirling 87]).

**Theorem 7.2.1** Let $P = (S_1, A_1, T_1, \uparrow, r_0)$, $Q = (S_2, A_2, T_2, \uparrow, s_0)$ and $\sqsubseteq_0 = S_1 \times S_2$, $\sqsubseteq_{k+1} = \mathcal{F}(\sqsubseteq_k)$. Then $P \sqsubseteq_{\text{wbi/ref}} Q$ if and only if for every $k$, $(r_0, s_0) \in \sqsubseteq_k$.  

The basic algorithm is now simple ([Celikkan and Cleaveland 95]).
Algorithm: Bisimulation refinement.

Input: Processes $P = (S_1, A_1, T_1, \uparrow, r_0)$ and $Q = (S_2, A_2, T_2, \uparrow, s_0)$.

Output: True, if $P \sqsubseteq_{\text{wbi}} Q$, false otherwise.

Method:

1. begin
2. \hspace{1em} $k := 1$;
3. \hspace{1em} $\sqsubseteq_0 := S_1 \times S_2$;
4. \hspace{1em} $\sqsubseteq_1 := \mathcal{F}(\sqsubseteq_0)$;
5. while $\sqsubseteq_k \neq \sqsubseteq_{k-1}$ do
6. \hspace{1em} $\sqsubseteq_{k+1} := \mathcal{F}(\sqsubseteq_k)$;
7. \hspace{1em} $k := k + 1$;
8. endwhile;
9. if $(r_0, s_0) \in \sqsubseteq_k$ then return true else return false;
10. end.

Conditions 1) and 2) in the definition of $\mathcal{F}$ are just the conditions in the definition of $\sqsubseteq_{\text{wbi}}$. By replacing the conditions for $\mathcal{F}$ with the conditions of the relation $\sqsubseteq_{\text{dref}}$ we get an algorithm for $\sqsubseteq_{\text{dref}}$.

The time complexity of the algorithm is quite high compared to the bisimulation equivalence algorithm. In the worst case the while loop executes $|S_1| \times |S_2|$ times. This would happen if each iteration removes exactly one pair from $\sqsubseteq_k$. The time spent on computing $\mathcal{F}$ is bounded by $|S_1| \times |S_2|$. Thus the total time is $O(|S_1|^2 \times |S_2|^2)$.

Celikkan and Cleaveland (1995) also give a more efficient algorithm. It is based on the ideas presented in [Bloom and Paige 92], [Cleaveland and Steffen 91] and [Cleaveland and Steffen 93]. The starting point is the observation that $R$ is a pre-bisimulation if and only if $R \subset \mathcal{F}(R)$. Thus, if $R \setminus \mathcal{F}(R) \neq \emptyset$ then $R$ is not a pre-bisimulation. The following algorithm computes the largest prebisimulation over $S_1 \times S_2$ and may thus be used to determine if $P \sqsubseteq_{\text{wbi}} Q$. 
Algorithm: Efficient bisimulation refinement.

Method:

1. begin
2. \( R := S_1 \times S_2 \); 
3. \( \text{ToDelete} := R \setminus \mathcal{F}(R) \); 
4. while \( \text{ToDelete} \neq \emptyset \) do 
5. \( \text{choose } (p, q) \in \text{ToDelete}; \)
6. \( R := R \setminus \{(p, q)\}; \)
7. \( \text{ToDelete} := R \setminus \mathcal{F}(R); \)
8. endwhile
9. end.

Our representation of the efficient algorithm is still too abstract for concrete implementation, but we don’t repeat the details, because there are plenty of them and they are explained thoroughly in [Celikkan and Cleaveland 95]. Let’s only notice that the implementation takes \( \mathcal{O}(|S_2| \times |P| + |Q| \times |S_1|) \) time, where \( |P| = |S_1| + |T_1| + |\uparrow_P| \).

7.3 BKO- and CFFD-equivalence

Testing the property \( P \approx_{CFFD} Q \) for two finite processes \( P \) and \( Q \) is algorithmically more difficult than testing \( P \approx_{wbis} Q \). There seems to be more than one candidate for efficient implementation. Our algorithm is nearly the same as presented in [Valmari and Tienari 91] and that’s why we introduce only the basic ideas. An efficient implementation needs more concentration on details, but these have been explained in [Valmari and Tienari 91]. We present only an algorithm for CFFD-equivalence, because BKO-equivalence can be computed in the same way with evident modifications.

Our aim is to attach a unique structure \( \mathcal{N}(P) \) to every finite process \( P \). This structure should be such that \( \mathcal{N}(P) = \mathcal{N}(Q) \), if \( P \approx_{CFFD} Q \), and \( \mathcal{N}(P) \neq \mathcal{N}(Q) \), if \( P \not\approx_{CFFD} Q \). Furthermore, it should be relatively easy to compute \( \mathcal{N}(P) \) algorithmically, at least on average. Our \( \mathcal{N} \) resembles the well-known determinization and minimization process of finite nondeterministic automata.

Let \( P = (S, A, T, \uparrow, s_0) \in \text{ALTS} \) be finite. Instead of \( P \), consider a process \( \hat{P} = (S, A, \hat{T}, \uparrow, s_0) \), where \( \hat{T} \) is as in the weak bisimulation algorithm, i.e. \( (s, a, s') \in \hat{T} \) if
\[ s \xrightarrow{a} s' \]. Interpret \( \hat{P} \) as a nondeterministic automaton with every state final. To compute \( \mathcal{N}(P) \), transform \( \hat{P} \) first into a deterministic automaton \( P_{\text{det}} = (S_d, A, T_d, \nu_0) \), where every state is final. This transformation is well-known, see for example [Hopcroft and Ullman 79]. Every state in \( S_d \) is a subset \( R \subset S \). Equip every \( R = \{s_1, \ldots, s_n\} \in S_d \) with information about the states \( s_1, \ldots, s_n \) as follows.

- A boolean variable \( d \) is attached to \( R \) with the following values:
  \[
  d = \begin{cases} 
  \text{true}, & \text{if Div}(s_i) \text{ for some } i = 1, \ldots, n, \\
  \text{false}, & \text{if Conv}(s_i) \text{, for all } i = 1, \ldots, n.
  \end{cases}
  \]

- A subset collection \( \text{Ref}(R) \) consists of subsets \( B \subset A \) such that there is a \( s \in R \), \( s \downarrow \), \( s \xrightarrow{a} B \) for all \( a \in B \) and there is no \( B' \) such that \( B' \) satisfies the previous conditions of \( B \) and \( B \subset B' \) properly. In other words, every \( B \) is maximal. We also use the notation \( \text{Ref}(s) \) for a single state in a state graph with the obvious meaning.

- A boolean variable \( p \) attached to \( R \) has the values:
  \[
  p = \begin{cases} 
  \text{true}, & \text{if } s_i \uparrow \text{ for some } i = 0, \ldots, n, \\
  \text{false}, & \text{if } s_i \downarrow \text{ for all } i = 1, \ldots n.
  \end{cases}
  \]

- Let \( R_0 \) be the initial state of \( P_{\text{det}} \). Then stable\((R_0)\) if and only if stable\((s_0)\).

We use refusal sets \( \text{Ref}(R) \) in our algorithms. In real implementations it is often more efficient to use acceptance sets. These are complements of refusal sets and contain less elements in many practical situations than refusal sets.

We denote the structure \( P_{\text{det}} \) with the attributes \( d, p, \text{Ref} \) and stable by \( \mathcal{N}_0(P) \) and call it an acceptance graph. Next \( \mathcal{N}_0(P) \) will be minimized. It turns out that the usual minimization algorithm for automata ([Aho, Hopcroft and Ullman 74]) can also be applied in our case. The only modification is the initial partition of the state set \( S_d \). The states in \( S_d \) are partitioned into initial classes on the basis of the attributes. If \( R \) is a state in \( \mathcal{N}_0(P) \), the equivalence class \( d(R) \) of \( R \) is defined by

\[
    d(R) = \{ R' \mid \text{Ref}(R) = \text{Ref}(R'), d(R) = d(R'), p(R) = p(R') \}.
\]

These classes constitute a partitioning of the state set, which is used as the initial partition in the minimization.

After the initial partition we no longer need the attributes in the algorithm. The minimized \( \mathcal{N}_0(P) \) with the attributes attached to the states is the structure \( \mathcal{N}(P) \), the normalization of \( P \). It is unique with the exception of the names of the states. The check \( P \approx_{\text{CFFD}} Q \) can now be done by comparing \( \mathcal{N}(P) \) and \( \mathcal{N}(Q) \). If they are the same modulo the state names, then \( P \approx_{\text{CFFD}} Q \), otherwise \( P \not\approx_{\text{CFFD}} Q \).

As it is known, the determinization step can take exponential time in the worst case. But in many situations the determinization proceeds quickly and thus the CFFD-equivalence is algorithmically relevant in applications.
7.4 BKO- and CFFD-refinement

To test whether $P \subseteq_{\text{CFFD}_{\text{ref}}} Q$ or not is more time-consuming than testing $P \approx_{\text{CFFD}} Q$. The reason is that we must check a lot of trace inclusions. Our algorithm is based on the idea to examine every trace with a length less than a certain limit. This limit depends on the number of states in the labeled transition systems $P$ and $Q$. It is not necessary to examine longer traces, because they must contain loops and these we can bypass.

In the articles [Cleaveland and Hennessy 93] and [Celikkkan and Cleaveland 92] the bisimulation algorithm is also used to compute the testing preorders. We have not considered this possibility in the case of the BKO- and CFFD-refinements, although it seems to be a plausible alternative. In fact, in our algorithm we have to compare explicitly refusal (or acceptance) sets, whereas in the above method these comparisons happen implicitly. So using the bisimulation algorithm may be more efficient than our method. However, our aim has been to demonstrate that the BKO- and CFFD-refinements can be computed efficiently enough. The question of the most efficient algorithm is left outside this thesis. It is an interesting research topic.

First we construct $\mathcal{N}(P)$ and $\mathcal{N}(Q)$. We test with the help of the sets examine and simulation, if every trace in $\mathcal{N}(P)$ belongs to $\mathcal{N}(Q)$. The set simulation will consist of pairs $(R, R')$ such that a trace leading to $R$ in $\mathcal{N}(P)$ also leads to $R'$ in $\mathcal{N}(Q)$. If $R$ can be reached with different traces, then we may have several pairs $(R, R'_1), \ldots, (R, R'_k)$ in simulation. The set examination will contain new pairs $(R, R')$ that should still be examined. We construct simulation step by step and at the end it will be maximal in the sense that it contains all possible pairs $(R, R')$. Notice that it is not necessary for our purposes to collect information about the actions in traces, but only about the states reachable by the traces.

The set simulation is constructed step by step with the help of the set examination. First the pair of the initial states is added to both sets. After that we take a pair from examination, say $(R, R')$, and collect the states reachable with one arc from $R$. The we check the corresponding transitions from $R'$. In this way we get new pairs. For every new pair we check, whether it is already in simulation. If it is not, we add it to both sets, otherwise it is not necessary to consider that pair anymore.

At the same time we construct new pairs we can also check for the attributes of partially defined states, divergences and stable failures. After all the states and paths in $\mathcal{N}(P)$ have been examined, we do the same for $\mathcal{N}(Q)$ with the exception that we study the attributes in a different way. Especially, if a trace $t$ in $\mathcal{N}(Q)$ contains a prefix to a partially defined state in $\mathcal{N}(P)$, all the conditions of the CFFD-refinement are satisfied. For this reason and because $\mathcal{N}(P)$ and $\mathcal{N}(Q)$ are deterministic, it is no more necessary to examine longer traces, which contain $t$ as a prefix.

If $R$ is a state in $\mathcal{N}(P)$, then we denote by $d(R)$ the value of the attribute $d$ in
$R$ and by $p(R)$ the value of the attribute $p$ in $R$. We must test that $\text{tr}(P) \subseteq \text{tr}(Q)$ and the traces or prefixes of the traces of $Q$ belong to $\text{tr}(P)$. In both cases we can apply the same kind of technique that we present as a procedure:

**procedure** Test-traces($R$, $\mathcal{N}$, $R'$, $\mathcal{N}'$, simulation, examine)

where

- $R$ is a state in $\mathcal{N}$,
- $\mathcal{N}$ is a normalized state graph,
- $R'$ is a state in $\mathcal{N}'$,
- $\mathcal{N}'$ is a normalized state graph,
- simulation is a set of state pairs $(R, R')$, where $R$ is in $\mathcal{N}$ and $R'$ in $\mathcal{N}'$,
- examine consists of the same kind of pairs as simulation.

1. begin
2. for every $(R, a, S)$ in $\mathcal{N}$ loop
3. if there is $(R', a, S')$ in $\mathcal{N}'$ then
4. if $(S, S') \notin \text{simulation}$ then
5. add $(S, S')$ to simulation;
6. add $(S, S')$ to examine;
7. end if;
8. else return 'no'; end if;
9. end for;
10. end.

Furthermore, for a pair $(R, R')$ in the simulation set we must check the refusal sets. We use the notation $\text{Ref}(R) \subseteq \text{Ref}(R')$, if for every set $U$ in $\text{Ref}(R)$ there is a set $U'$ in $\text{Ref}(R')$ such that $U \subseteq U'$. 
Algorithm: CFFD-refinement test
Input: Finite state graphs of processes $P$ and $Q$.
Output: 'Yes', if $P \subseteq_{CFFD_{ref}} Q$, 'no' otherwise.
Method:
1. begin
2. check the stable predicates of $P$ and $Q$;
3. if OK, continue, otherwise return 'no';
4. construct $\mathcal{N}(P)$ and $\mathcal{N}(Q)$ with initial states $R_P$ and $R_Q$, respectively;
   (* Phase 1: check the traces of $P$ *)
5. simulation := $\{(R_P, R_Q)\}$;
6. examine := simulation;
7. while examine $\neq \emptyset$ loop
8. take $(R, R')$ from examine;
9. if $(d(R) \text{ and } (\text{not } d(R'))) \text{ or } (\text{Ref}(R) \subset \text{Ref}(R'))$ then return 'no';
10. end if;
11. Test-traces($R$, $\mathcal{N}(P)$, $R'$, $\mathcal{N}(Q)$, simulation, examine);
12. end while;
   (* Phase 2: check the traces of $Q$ *)
13. simulation := $\{(R_Q, R_P)\}$;
14. examine := simulation;
15. while examine $\neq \emptyset$ loop
16. take $(R, R')$ from examine;
17. if not $p(R')$ then
18. if $p(R) \text{ or } (d(R) \text{ and } (\text{not } d(R'))) \text{ or } (\text{Ref}(R) \subset \text{Ref}(R'))$ then
return 'no';
19. end if;
20. Test-traces($R$, $\mathcal{N}(Q)$, $R'$, $\mathcal{N}(P)$, simulation, examine);
21. end if;
22. end while;
23. \ return 'yes';

24. \ end.

Next we prove that the previous algorithm is correct. The proof consists of a long series of lemmas.

\textbf{Lemma 7.4.1} The algorithm stops after a finite number of steps.

\textit{Proof.} The algorithm stops, because in the while-loops one pair of states is taken every time from \textit{examine}. A pair is added to \textit{examine} only if it has not been there already. The procedure \texttt{Test-traces} takes care of this with the help of the set \textit{simulation}. Thus the upper bound for the while loops is $m \times n$, where $m$ is the number of states in $\mathcal{N}(P)$ and $n$ is the number of states in $\mathcal{N}(Q)$. □

\textbf{Lemma 7.4.2} Let $t \in \text{tr}(P) \subseteq \text{tr}(Q)$ and consider the first part of the algorithm without the conditions for divergences and stable failures. Then there will be an element $(R, S)$ in simulation such that $R_p \xrightarrow{\tau} R$ and $R_Q \xrightarrow{\tau} S$.

\textit{Proof.} We use induction on the length of $t$. In this case the induction principle is quite powerful and it not necessary to consider for example loops in the path accepting $t$.

Suppose $t = \varepsilon$, i.e. $|t| = 0$. Then $R = R_p$ and $S = R_Q$, because $\mathcal{N}(P)$ and $\mathcal{N}(Q)$ are deterministic without $\tau$-transitions. The pair $(R_p, R_Q)$ is in simulation already at the beginning of the algorithm.

Suppose $|t| = k > 0$ and the claim is true for all traces shorter than $k$. Suppose $t = u \cdot a$, $|u| = k - 1$. Let $R_p \xrightarrow{u} R_i \xrightarrow{a} R$ and $R_Q \xrightarrow{u} R_j \xrightarrow{a} S$. By the induction hypothesis, the pair $(R_i, R_j)$ will be in simulation. Hence it will also be in \textit{examine}. Thus in the while-loop 7-12 the pair $(R_i, R_j)$ will be picked at some time from \textit{examine} and the transitions from $R_i$ and $R_j$ will be examined. Especially, the procedure \texttt{Test-traces} will find $R$ and $S$ and the pair $(R, S)$ will be added to \textit{simulation} and \textit{examine}. It is also possible that the pair has been added earlier, when another trace has been examined. □

\textbf{Lemma 7.4.3} Let $(R, S)$ be an element added into simulation at the $k$:th round of the first while loop. Then there is a trace $t$ of length $\leq k$ such that $R_p \xrightarrow{t} R$ and $R_Q \xrightarrow{t} S$.

\textit{Proof.} We use induction on $k$. If $k = 0$, we must have $R = R_p$, $S = R_Q$ and $t = \varepsilon$. Thus the claim is valid.

Suppose $k > 0$ and the claim is true for all rounds $1, \cdots, k - 1$. Assume the pair $(R, S)$ is added into \textit{simulation} and \textit{examine} at the $k$:th round. This addition is done in the procedure \texttt{Test-traces} and it shows that $R$ is reached with an action $a$ from some $R'$ in $\mathcal{N}(P)$ and $S$ is reached with $a$ from some $S'$ in $\mathcal{N}(Q)$. Furthermore,
the pair \((R', S')\) has been taken from examine, so it is also in simulation. The pair
\((R', S')\) has been added into simulation and examine at an earlier stage and hence
we can apply the induction hypothesis to \((R', S')\). Thus there is a trace \(u\) such that
\(R_P \xrightarrow{u} R', \ R_Q \xrightarrow{t} S'\) and \(|u| < k - 1\). If we now choose \(t = ua\), we have \(R_P \xrightarrow{t} R, \ R_Q \xrightarrow{u} S\) and \(|t| \leq k\).

**Lemma 7.4.4** Suppose \(t \in \text{tr}(P)\), but \(t \notin \text{tr}(Q)\). Consider the first phase of the
algorithm without conditions for divergences and stable failures. The algorithm outputs 'no'.

**Proof.** We again use induction on the length of \(t\). If \(|t| = 1\), then \(t = a\) for some
action \(a\). There is a transition with \(a\) from \(R_P\), but no transition with \(a\) from \(R_Q\).
Because \((R_P, R_Q)\) is in examine, the procedure test-traces notices in step 3 that
\(t \notin \text{tr}(Q)\) and returns 'no'. Of course, the output 'no' could also occur earlier.

Suppose \(|t| = k > 1\) and the claim is valid for all traces of length less than \(k\). Let
\(t = ua\). If \(u \notin \text{tr}(Q)\), then by the induction hypothesis the algorithm outputs 'no'.
So suppose \(u \in \text{tr}(Q)\). If the algorithm does not output 'no', we can conclude by
following the proof of Lemma 7.4.2 that there are \(R'\) in \(\mathcal{N}(P)\) and \(S'\) in \(\mathcal{N}(Q)\) such that
\(R_P \xrightarrow{u} R', \ R_Q \xrightarrow{t} S'\) and \((R', S')\) will be added into examine and simulation.
Now the procedure test-traces finds that \(t = ua\) is not in \(\text{tr}(Q)\) and outputs 'no'.

**Lemma 7.4.5** Suppose \(t \in \text{divtr}(P)\), but \(t \notin \text{divtr}(Q)\). Then the first phase of the
algorithm outputs 'no'.

**Proof.** If \(t \notin \text{tr}(Q)\), then by Lemma 7.4.4 the algorithm outputs 'no'. So suppose
\(t \in \text{tr}(Q)\). There are \(R\) and \(S\) such that \(R_P \xrightarrow{t} R, \ R_Q \xrightarrow{u} S\) and \((R, S)\) will be added into simulation and examine by Lemma 7.4.2, if the algorithm does not output 'no'
before that.

Suppose the algorithm picks \((R, S)\) at some round of the while-loop 7-12. In
line 9 the algorithm checks conditions \(d(R)\) and \(d(S)\). Because \(t \in \text{divtr}(P)\) and
\(R\) is unique, \(d(R)\) must be true on the basis of the construction of \(\mathcal{N}(P)\). Because
\(t \notin \text{divtr}(Q)\) and because \(S\) is also unique, \(d(S)\) must be false. Thus the algorithm
returns 'no'.

**Lemma 7.4.6** Suppose \((t, L) \in \text{sfail}(P)\), but \((t, L) \notin \text{sfail}(Q)\). Then the first phase
of the algorithm outputs 'no'.

**Proof.** In the same way as the proof of Lemma 7.4.5.

**Lemma 7.4.7** If the first phase of the algorithm outputs 'no', then \(P \notin \text{CFFD-rel} Q\).

**Proof.** Suppose the first phase outputs 'no' after taking a pair \((R, S)\) from examine.
By Lemma 7.4.3 there is a trace \(t\) such that \(R_P \xrightarrow{t} R\) and \(R_Q \xrightarrow{u} S\). Suppose the
algorithm outputs 'no' when at line 9. The output is the result of the tests for divergences and stable failures. For example, if \( d(R) \) is true, but \( d(S) \) is false, this means that there is a state \( r \) in \( R \) such that a trace leading to \( r \) diverges, but the same trace does not belong to \( \text{divtr}(Q) \). So the output is correct. In the same way the output is correct, if it is the result of the condition for stable failures. □

**Lemma 7.4.8** If the algorithm outputs 'no', \( P \not\subseteq_{\text{CFFD}_{\text{ref}}} Q \).

**Proof.** We have already proved the result for the first phase of the algorithm, so consider the second phase. The lemmas 7.4.2-7.4.3 are valid also for the second while loop with the following exception: Suppose \( t \in \text{tr}(Q) \cap \text{tr}(P) \) and consider the second while loop without conditions for divergences and stable failures. Then there is a pair \( (S, R) \) in simulation such that \( R_Q \overset{t}{\rightarrow} S \) and \( R_P \overset{t}{\rightarrow} R \) or \( R_Q \overset{u}{\rightarrow} S \), \( R_P \overset{u}{\rightarrow} R \), where \( u \) is a prefix of \( t \) and \( u \in \text{partr}(P) \).

We can now proceed in the same way as with the first phase and conclude that the claim is valid. □

**Lemma 7.4.9** If the algorithm outputs 'yes', \( P \subseteq_{\text{CFFD}_{\text{ref}}} Q \).

**Proof.** The algorithm outputs 'yes', if it has not output 'no' before. We have proved with the previous lemmas that if one of the conditions for cffd-refinement is violated, the algorithm detects this and outputs 'no'. So if 'no' has not been output, it means that all the conditions are valid. □

**Theorem 7.4.10** The algorithm decides correctly, if \( P \subseteq_{\text{CFFD}_{\text{ref}}} Q \) or not.

□

It is possible to make the algorithm faster. The construction of the normalized versions \( \mathcal{N}(P) \) and \( \mathcal{N}(Q) \) may be time-consuming and it turns out that it is not necessary to normalize the whole of \( P \).

In the first phase, we can use \( P \) instead of the normalized version. If \( s \) is a state in \( P \), we can use the predicates \( d \) and \( \text{Ref} \) with \( s \) in the same way as with a state of the normalized graph. So it is not necessary to change the first phase of the algorithm in any way.

In the second phase we need the normalized version of \( P \), but only up to partially defined states. So we first construct \( P' \) from \( P \) by leaving all the transitions from partially defined states out. Then we construct \( \mathcal{N}(P') \) and use it in the second phase of the algorithm instead of \( \mathcal{N}(P) \). No other changes in the second phase are necessary.

So if \( P \) contains partially defined states near the initial state, we can probably leave most states and transitions out and thus the task of the second loop is more easily executed.
The most time consuming task is the determinization. It is exponential in the worst case. Otherwise, the while-loops in the algorithm are performed at the most \( m \times n \) times, where \( m \) and \( n \) are the number of states in \( \mathcal{N}(P) \) and \( \mathcal{N}(Q) \), respectively. In every round of the loops the algorithm checks the conditions for divergences and stable failures. The divergences can be checked in a constant time in the loops, because they have been calculated beforehand after the determinization. If \( a \) is the number of actions in the systems, there can be \( 2^a \) refusal (or acceptance) sets and each set has at most \( a \) elements. So the comparison of stable failures is at worst \( \mathcal{O}(2^a) \), but in practice very often less, nearly constant. It seems that if the number of states in the determinization is reasonably small, the number of refusal sets also is small. These two are related, at least in practice. So the complexity of the algorithm after the determinization is \( \mathcal{O}(m \times n \times 2^a) \) in the worst case, but \( \mathcal{O}(m \times n) \) in many practical situations.

7.5 Practical implementation of CFFD

I have implemented BKO- and CFFD-refinements in the presence of partially defined states. If one is going to use existing programs to generate labeled transition systems from Lotos specifications, a short way to implement BKO- and CFFD-relations is as follows.

Define `udef` as an ordinary Lotos process. Generate a labeled transition system into a file using an existing software. Determinize the system in a normal way. Add attributes to the determinized system. This can be done by visiting every node using every possible path in the original system with the exception of the nodes reachable only via \( \eta \)-transitions and collecting the attributes for a node. Search the nodes in the determinized system that are reachable with the traces and add the attribute information to these nodes. Do the same for the other Lotos specification. After this it is possible to check the BKO- and CFFD-relations by comparing the determinized systems and their attributes.

The determinization step can usually be done with existing Lotos software. By proceeding this way it is necessary to write only about 1500 lines of code in order to calculate BKO- and CFFD-relations. This code includes a simple user interface, but no efficient memory management.
Chapter 8

Abstract versus concrete notions of refinement

8.1 An Introduction

We have explained in section 2.2 how the refinement is done in practice. It was conjectured that this intuitive concept of refinement is valid in all the situations. Let $P$ and $Q$ be processes, $\approx$ an equivalence and $\subset$ a refinement relation related to the equivalence. Then our conjecture is defined as follows.

**Structure Conjecture.** We have $P \subset Q$ if and only if there are $P'$ and $Q'$ such that $P \approx P'$, $Q \approx Q'$, $P' \subset Q'$ and $Q'$ is obtained from $P'$ by adding transitions from partially defined states, i.e. states $p$ of $P$ such that $p \uparrow$, to arbitrary augmented labelled transition systems or changing partially defined states to totally defined.

We have now enough machinery to study the conjecture in bisimulation and CFFD-semantics. It turns out that the conjecture is not true in the bisimulation semantics, but a modified version of the conjecture can be shown to be valid. However, the conjecture is true in the CFFD-semantics.

8.2 The conjecture in the bisimulation semantics

8.2.1 The counterexamples

The definition of the weak bisimulation refinement does not correspond completely with the intuitive concepts, and the conjecture turns out to be false in its original form. As a counterexample, consider the processes $P$ and $Q$ defined by the diagrams
\[
P : \rightarrow P1 \xrightarrow{a} P2 \uparrow \xrightarrow{b} P3 \\
\rightarrow Q1 \xrightarrow{a} Q4 \xrightarrow{c} Q5 \\
\]

\[Q : \]
\[Q2 \]
\[\]
\[Q3 \]

We have \(P \sqsubseteq_{\text{bisim}} Q\), because \(R = \{(P1, Q1), (P2, Q2), (P3, Q3)\}\) is a prebisimulation. The process \(Q\) is not obtained from \(P\) by adding transitions to partially defined states. Moreover, there is no such \(P'\) and \(Q'\) as the conjecture demands. For example, the processes \(P'\) and \(P''\), defined by the diagrams

\[
P' : \rightarrow P1 \uparrow \xrightarrow{a} P2 \uparrow \xrightarrow{b} P3 \\
P'' : \xrightarrow{b} P2 \uparrow \xrightarrow{b} P3 \\
\]

are not weakly bisimilar with \(P\). On the other hand, the process \(Q\) is already minimal and there is no larger process satisfying the conditions of the the conjecture. We can conclude that the conjecture is not correct with respect to \(P\) and \(Q\).

It might be thought that the reason for this behaviour is the difference in the definitions of the weak bisimulation equivalence and refinement. Two states \(P\) and \(Q\) are equivalent, only if \(P \uparrow \iff Q \uparrow\). Two states \(P\) and \(Q\) correspond with each other in the prebisimulation sense only if \(P \downarrow a\) implies \(Q \downarrow a\). So in the equivalence we use the condition \(\uparrow\) without an action and in the refinement with an action. If we changed the definition of the prebisimulation so that \(\uparrow\) were used instead of \(\uparrow a\), the conjecture would remain false as the following example shows:

\[
P : \]
\[\]
\[\]
\[\]
\[\]

\[
\]

\[
Q : \]
\[\]
\[\]
\[\]
\[\]
The situation is the same as before so that \( P \sqsubseteq_{\text{wbi-ref}} Q \), but there are \( P' \) and \( Q' \) as the conjecture demands.

As we have seen in chapter 3, the partiality condition of the bisimulation refinement is necessary, if the relation is wanted to be a congruence with respect to action prefix. On the other hand, the partiality condition of the bisimulation equivalence is very natural and corresponds with the intuitive concept of equivalence better than the condition with \( \uparrow a \). Moreover, the conjecture would remain false even if we changed the definition of the prebisimulation.

### 8.2.2 The modified conjecture in the bisimulation semantics

We show that in the case \( P \sqsubseteq_{\text{wbi-ref}} Q \) the process \( P \) can be considered to be a subgraph of \( Q \). This means the following. Let \( P \) be a labelled transition system with a state (node) set \( V_P \) and an arc set \( E_P \). Thus \( E_P \subseteq V_P \times A \times V_Q \), where \( A \) is the set of actions in \( P \). Similarly, let \( Q \) consist of a state set \( V_Q \) and an arc set \( E_Q \). We say that \( P \) is a subgraph of \( Q \), if there is an injection \( f: V_P \rightarrow V_Q \) such that the initial state of \( P \) maps to the initial state of \( Q \) and \( (v_1, a, v_2) \in E_P \) implies \( (f(v_1), a, f(v_2)) \in E_Q \).

**Proposition 8.2.1** Let \( P \sqsubseteq_{\text{wbi-ref}} Q \). There are processes \( P' \) and \( Q' \) such that \( P \approx_{\text{wbi}} P' \), \( Q \approx_{\text{wbi}} Q' \) and \( P' \) is a subgraph of \( Q' \).

**Proof.** Let \( \mathcal{R} \) be the prebisimulation between \( P \) and \( Q \). Let \( P_0 \) and \( Q_0 \) be the initial states of \( P \) and \( Q \), respectively. Construct two trees according to the rules given below. Every node in the trees is of the form \((p, q)\) or \((X, q)\), where \( p \) and \( q \) are states of \( P \) and \( Q \) and \( X \) is a special symbol.

The rule are as follows:

a) If \((p, q)\) is a node in the tree and \( p \neq X \), add a child \((p', q')\) with a transition \( a \) for every \( p', q' \) and \( a \) such that \( p \xrightarrow{a} p' \), \( q \xrightarrow{a} q' \) and \((p', q') \in \mathcal{R} \). Note that \( a \) may be \( \varepsilon \).

b) If \((p, q)\) is a node in the tree and \( p \neq X \), then add a child \((X, q')\) with a transition \( a \) for every \( q' \) such that \( q \xrightarrow{a} q' \) and there is no \( p' \) with \( p \xrightarrow{a} p' \), \((p', q') \in \mathcal{R} \).

c) If \((X, q)\) is a node in the tree and \( q \xrightarrow{a} q' \), then add a child \((X, q')\) with a transition \( a \).

Define \( T_1 \) to be a tree which is constructed by starting from the root \((p_0, q_0)\) and applying rule a) as long as possible. Define \( T_2 \) to be a tree which is constructed by starting from the same initial node and applying rules a), b) and c) as long as possible.

A node \((p, q)\) in \( T_1 \) is partially defined, if \( p \) is partially defined. A node \((p, q)\) in \( T_2 \), where \( p \) may be \( X \), is partially defined, if \( q \) is partially defined.
If we define $\mathcal{E}_1 = \{ (p, (p, q)) \}$, where $p$ is a state in $P$ and $(p, q)$ a state in $T_1$, then $\mathcal{E}_1$ is a weak bisimulation between $P$ and $T_1$. Similarly, $\mathcal{E}_2 = \{ (q, (p, q)) \}$, where $p$ is a state in $P$ or $X$, is a weak bisimulation between $Q$ and $T_2$. Moreover, $T_1 \sqsubseteq_{\text{wibis}} T_2$, because $\mathcal{R}' = \{ ((p, q), (p, q)) \}$, where $(p, q)$ is a state in $T_1$, is a pre-bisimulation between $T_1$ and $T_2$.

By the construction, $T_1$ is a subgraph of $T_2$. The graphs may be infinite, but we can stop to add new nodes, if we reach a node which already is on the path from the initial node to the current node. In this case we just draw the transition to the earlier node on the path. In this way we get two graphs and the graph corresponding with $T_1$ is a subgraph of the graph corresponding with $T_2$.

$\Box$

The converse of the previous proposition also is valid, if we put further restrictions on the subgraphs. Let a labelled transition system $P$ be a subgraph of a labelled transition system $Q$ with the following restriction. Let $f$ be an injection between the state sets of $P$ and $Q$. If $v$ is a state in $P$ and $f(v)$ has out-transitions not present in $v$, then $v$ is partially defined. With these restrictions it is seen directly from the definitions that $P \sqsubseteq_{\text{wibis}} Q$.

Next, we consider how $T_2$ differs from $T_1$. Take a node $(p, q)$ which is in $T_1$ and $T_2$ and suppose that $(p, q)$ contains a transition in $T_2$ not present in $T_1$. By the construction, this means that the transition is of the form $(p, q) \xrightarrow{a} (X, q')$. Because $T_1 \sqsubseteq_{\text{wibis}} T_2$, it is necessary that $p \uparrow a$. This means that $p$ or some state reachable by $\xrightarrow{a}$ is partially defined. We can state

**Theorem 8.2.2** $P \sqsubseteq_{\text{wibis}} Q$ if and only if there are labeled transition systems $P'$ and $Q'$ such that $P \approx_{\text{wibis}} P'$, $Q \approx_{\text{wibis}} Q'$, $P' \sqsubseteq_{\text{wibis}} Q'$, and $Q'$ is obtained from $P'$ by adding $a$-transitions from states of $P'$ such that $p \uparrow a$, to arbitrary augmented labelled transition systems, or by changing partially defined states of $P'$ to totally defined.

**Proof.** We have proved the if-part already. To prove the converse, if $P \approx_{\text{wibis}} P'$, $Q \approx_{\text{wibis}} Q'$ and $P' \sqsubseteq_{\text{wibis}} Q'$, then $P \sqsubseteq_{\text{wibis}} Q$, because the equivalence $\approx_{\text{wibis}}$ is also the preorder $\sqsubseteq_{\text{wibis}}$ and $\sqsubseteq_{\text{wibis}}$ is transitive. $\Box$

Reasoning in the same way it is easy to see that if we replace $\downarrow a$ in the definition of bisimulation refinement by $\downarrow$, the previous result also is valid with the exception that $\uparrow$ is used instead of $\uparrow a$.

This result is the best possible. Thus it is not possible to describe a refinement in the bisimulation semantics using only the simple concept of a subgraph. This seems to be an inherent property of the bisimulation refinement.
8.3 The conjecture in the CFFD-semantics

8.3.1 The plan of the proof in CFFD-semantics

The proof is based on the concepts of the acceptance graph and normalization presented in the previous chapter. So the first thing is to construct $\mathcal{N}(P)$ and $\mathcal{N}(Q)$. After this it is possible to apply the converse construction that builds an ordinary transition system from an acceptance graph. This construction is presented in the paper [Valmari and Tienari 91] and we can follow it with three modifications. The first modification is that we must take the partially defined states into account. The second modification is that it is necessary to handle the two acceptance graphs at the same time, because the other must be a subgraph of the other. If done separately, the subgraph property is not satisfied completely. The third modification is that the structure of the failure sets in the two processes must be taken into account in a new way in order to preserve the subgraph property.

It is necessary to do one more trick before the proof is complete. In order that the construction yields the labeled transition system $Q'$ and its subgraph $P'$, it is necessary that $\mathcal{N}(P)$ is a subgraph of $\mathcal{N}(Q)$ as well. However, this is not the case in general. Consider the following example:

Both $P$ and $Q$ are deterministic as ordinary transition systems. So as graphs $\mathcal{N}_0(P)$ is $P$ and $\mathcal{N}_0(Q)$ is $Q$. Moreover, $\mathcal{N}_0(P)$ already is minimal and $\mathcal{N}_0(Q)$ is minimal after combining the states $Q_5$ and $Q_8$. Anyway, $P \subseteq_{CFFD_{rel}} Q$ and it follows that $\mathcal{N}(P)$ is not necessarily a subgraph of $\mathcal{N}(Q)$.

It is possible to transform $\mathcal{N}(P)$ into an equivalent acceptance graph $\mathcal{N}(P')$, which is a subgraph of $\mathcal{N}(Q)$. The acceptance graph $\mathcal{N}(P')$ is not minimal, but this is not necessary. We shall show how to do this in the general case.

**Definition 8.3.1** Two acceptance graphs $\mathcal{N}(P)$ and $\mathcal{N}(P')$ with initial states $p$ and $p'$ are CFFD-equivalent, if the following conditions are valid:
• \( \text{stable}(p) = \text{stable}(p') \).

• \( \text{sfail}(\mathcal{N}(P)) = \text{sfail}(\mathcal{N}(P')) \), where \( \text{sfail}(\mathcal{N}(P)) \) is the set

\[
\{(u, A) \mid p \xrightarrow{u} \nu, \ A \subset B, \ B \in \text{Ref}(\nu)\},
\]

\( \nu \) is a state in \( \mathcal{N}(P) \).

• \( \text{div}(\mathcal{N}(P)) = \text{div}(\mathcal{N}(P')) \), where \( \text{div}(\mathcal{N}(P)) \) is the set

\[
\{u \mid p \xrightarrow{u} \nu, \ d(\nu)\},
\]

where \( \nu \) is a state in \( \mathcal{N}(P) \).

It is known that when generating labeled transition systems from acceptance graphs, CFFD-equivalent acceptance graphs yield CFFD-equivalent labeled transition systems.

8.3.2 The proof of the structure conjecture in CFFD-semantics

The conjecture is proved with the following two theorems. We only consider the case, where the processes are finite.

**Theorem 8.3.1** Suppose \( P \sqsubseteq_{\text{CFFD}_{\text{ref}}} Q \). Then there is an acceptance graph \( N \) such that as a graph \( N \) is a subgraph of \( \mathcal{N}(Q) \) and as an acceptance graph \( N \) is CFFD-equivalent with \( \mathcal{N}(P) \). More precisely, if \( \nu_N \) is a state in \( N \) and \( \nu_Q \) is the corresponding state in \( \mathcal{N}(Q) \), then \( \text{Ref}(\nu_N) \subseteq \text{Ref}(\nu_Q) \).

**Proof.** As automata, \( P \) is finite and \( \mathcal{N}(Q) \) is finite and deterministic. Because \( P \sqsubseteq_{\text{CFFD}_{\text{ref}}} Q \), every trace of \( P \) is a trace of \( \mathcal{N}(Q) \) as well. So take a trace \( t \) of \( P \) leading to a state \( p \) and consider a state \( \nu_Q \) in \( \mathcal{N}(Q) \) reachable by the trace \( t \) from the initial state of \( \mathcal{N}(Q) \). Add attributes to \( \nu_Q \). Initially, partiality and divergence attributes are false and the refusal set is empty. If \( p \) is partially defined or divergent, then the corresponding attributes in \( \nu_Q \) become true. Add a possible refusal set to \( \text{Ref}(\nu_Q) \). Combine the refusal sets so that only the maximal ones remain.

Go through all the traces of \( P \) using all the possible paths in the same way. In fact, because \( P \) is finite, it is necessary to examine only a finite number of paths. The graph consisting of the visited nodes of \( \mathcal{N}(Q) \) with the added attributes is the acceptance graph \( N \) in the theorem. □

Before proceeding to the main theorem, we introduce some notational concepts. Let \( \mathcal{N}(P) \) and \( \mathcal{N}(Q) \) be two acceptance graphs such that \( \mathcal{N}(P) \) is isomorphic to a subgraph of \( \mathcal{N}(Q) \) as given in the previous theorem. We will present in what follows how to construct labeled transition systems \( P' \) and \( Q' \) from \( \mathcal{N}(P) \) and \( \mathcal{N}(Q) \), respectively, such that the structure conjecture is valid. For this purpose we use the following notations.
8.3. THE CONJECTURE IN THE CFFD-SEMANTICS

- Denote by \( \nu_P \) a state in \( \mathcal{N}(P) \) and by \( \nu_Q \) the corresponding state in \( \mathcal{N}(Q) \) under the assumed isomorphism.

- If \( \nu_Q \) is a state in \( \mathcal{N}(Q) \), then \( \text{fwd}(\nu_Q) \) denotes the acceptance graph consisting of all the nodes, with the attributes, in \( \mathcal{N}(Q) \) reachable from \( \nu_Q \). The node \( \nu_Q \) itself belongs to \( \text{fwd}(\nu_Q) \). As a matter of fact, we denote the node \( \nu_Q \) in \( \text{fwd}(\nu_Q) \) with \( \text{fwd}(\nu_Q) \), too. In what follows, we consider \( \text{fwd}(\nu_Q) \) as a separate graph from \( \mathcal{N}(Q) \).

Then \( \nu_Q \) in \( \text{fwd}(\nu_Q) \) will be denoted by \( \nu_Q^{\text{fwd}} \).

- The labeled transition system \( P' \) to be constructed from the acceptance graph \( \mathcal{N}(P) \) will contain every \( \nu_P \) and some extra states related to \( \nu_P \). We will denote by \( s(\nu_P) \) the state in \( P' \) corresponding to \( \nu_P \). Furthermore, \( S(\nu_P) \) denotes the state set of \( P' \) related to \( \nu_P \). The complete state set of \( P' \) will be \( \bigcup_{\nu \in \mathcal{N}(P)} S(\nu) \).

The same conventions are valid for \( Q' \).

- If \( \text{Ref}(\nu_P) = \{ A_1, \ldots, A_n \} \) and \( A \) is the set of all actions in \( \mathcal{N}(P) \), excluding \( \tau \), then \( \text{Acc}(\nu_P) \) denotes the set \( \{ \bar{A}_1, \ldots, \bar{A}_n \} \), where \( \bar{A}_i = A \setminus A_i \), \( i = 1, \ldots, n \). In other words, \( \bar{A}_i \) is a set such that there is a state \( s \) in \( P, s \in \nu_P \) and \( s \xrightarrow{a}, s \xrightarrow{\tau} \), for every \( a \in \bar{A}_i \). The same notations are used for \( \mathcal{N}(Q) \) with the subscript \( P \) replaced by \( Q \).

- \( \delta_P \) is the transition relation in \( \mathcal{N}(P) \). We write \( \delta_P(\nu_P, a) = w \), if there is a transition from \( \nu \) to \( w \) with \( a \). Similar for \( \mathcal{N}(Q) \).

- \( \text{out}(\nu) \) is the set of outgoing actions in node \( \nu \).

Our construction will not be the most compact. It would be possible to reduce the amount of \( \tau \)-transitions, but the rules would be more complicated. The purpose of the following theorem is only to show the validity of the conjecture using a minimal number of rules in the construction. The example after the following theorem will show how to minimize \( \tau \)-transitions.

**Theorem 8.3.2** Suppose \( P \sqsubseteq_{\text{CFFD}_{el}} Q \) such that \( \mathcal{N}(P) \) is a subgraph of \( \mathcal{N}(Q) \). Then the conjecture is true in CFFD-semantics.

**Proof.** The proof is constructive. The transition systems \( P' \) and \( Q' \) are constructed step by step in parallel. First we take a state \( \nu_P \) in \( \mathcal{N}(P) \) and its corresponding state \( \nu_Q \) in \( \mathcal{N}(Q) \). We show how to define the state sets \( S(\nu_P) \) and \( S(\nu_Q) \) and to draw the outgoing transitions from the states in these sets. We go through all the states in \( \mathcal{N}(P) \) and the corresponding states in \( \mathcal{N}(Q) \) one by one. After this, it is shown how to deal with the other states in \( \mathcal{N}(Q) \). All the states of \( \mathcal{N}(P) \) have been dealt with already.

Let \( \nu_P \in \mathcal{N}(P) \) and \( \nu_Q \) the corresponding state in \( \mathcal{N}(Q) \). We have the following two cases to consider:
1. $\sim p(\nu_P)$,
2. $p(\nu_P)$.

Now we present the rules how to define the state sets $S(\nu_P)$ and $S(\nu_Q)$ and how to draw the transitions.

**Case 1.**

- **States:** Let $\text{Acc}(\nu_P) = \{A_1, \ldots, A_n\}$. Note that $\text{Acc}(\nu_P)$ may be empty.
  - $S(\nu_P) = \{s(\nu_P), s_1(\nu_P), \ldots, s_n(\nu_P)\}$,
  - $S(\nu_Q) = \{s(\nu_Q), s_1(\nu_Q), \ldots, s_n(\nu_Q)\}$.
Note that we use the sets in $\text{Acc}(\nu_P)$ in $Q'$ too and do not care of the sets in $\text{Acc}(\nu_Q)$ for the moment. These sets are taken into account in the next case.

- **Transitions:**
  - $(s(\nu_P), \tau, s_i(\nu_P)), i = 1, \ldots, n,$
  - $(s(\nu_Q), \tau, s_i(\nu_Q)), i = 1, \ldots, n,$
  - if $d(\nu_P)$, then $(s(\nu_P), \tau, s(\nu_P))$ and likewise for $\nu_Q$,
  - $(s_i(\nu_P), a, s(\delta_P(\nu_P, a)))$ for each $a \in A_i$, $i = 1, \ldots, n$, and likewise for $\nu_Q$.

- **Example (again dropping the subscripts $P$ or $Q$):**

\[
\begin{align*}
\nu^{(2)} \overset{b}{\leftarrow} & \nu \quad \{a\}, \{a, b\} \quad \Rightarrow \quad s(\nu) \overset{\tau}{\rightarrow} s(\nu_2) \overset{a}{\rightarrow} s(\nu^{(2)}) \\
\nu^{(1)} \quad & \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad
\end{align*}
\]

**Case 2.** This is the case where the two constructions differ in details.

- **States:** As in case 1) with the additions explained with transitions.

- **Transitions:**
  - For every $A \in \text{Acc}(\nu_P)$ follow the rules given in case 1).
  - For every $b \in \text{out}(\nu_P) \cap \text{out}(\nu_Q)$ such that $b$ does not belong to any set in $\text{Acc}(\nu_P)$, follow the rule given in case 1).
  - Set $s(\nu_P) \uparrow$. If $p(\nu_Q)$, set $s(\nu_Q) \uparrow$, too.
  - In $Q'$, make the following additions. Draw the transition

\[
(s(\nu_Q), \tau, \text{inj}(\nu_P)^{\text{fwd}}),
\]

where $\text{fwd}(\nu_Q)$ denotes the node $\nu_Q$ in the acceptance graph $\text{fwd}(\nu_Q)$ as explained before the theorem. (This notational confusion seems to be
well-motivated in this case.) So during the construction, $Q'$ contains both parts of the labeled transition system and acceptance graph. After all the nodes in $\mathcal{N}(P)$ and the corresponding nodes in $\mathcal{N}(Q)$ have been gone through, return to the acceptance graphs $\text{fwd}(\nu_Q)$ and apply the case 1 to their nodes. The acceptance graph $\mathcal{N}(P)$ is no more used in this phase.

- Example: See the example after the proof.

It follows directly from the construction that $P'$ is a subgraph of $Q'$ and $P' \subseteq_{\text{CFFD-ref}} Q'$. No new traces or stable failures have been added to $P'$ and $Q'$ compared to $P$ and $Q$. The transition system $Q'$ may contain same traces many times, i.e. it may be highly non-deterministic because of the subgraphs $\text{fwd}(\nu_Q)$, but this does not disturb the CFFD-property. Thus it is straightforward to show that $P \approx_{\text{CFFD}} P'$ and $Q \approx_{\text{CFFD}} Q'$. We only show that stable failures are same both in $Q$ and $Q'$. The same method can be followed in the other cases, too.

So suppose $(u, L)$ is a stable failure of $Q$. Instead of stable failures, consider $(u, A)$, where $A$ is an acceptance set, a complement of $L$. We know that there is a state $\nu_Q$ in $\mathcal{N}(Q)$ such that $u$ leads to $\nu_Q$. Furthermore, there is $B \in \text{Acc}(\nu_Q)$ such that $B \subseteq A$. The construction is applied to every node in $\mathcal{N}(Q)$. By examining the cases 1-2 we see that there is stable state $s$ in $Q'$, reached by $u$, such that $s \xrightarrow{a}$ for all $a \in B$, but $s \xrightarrow{b}$ for every $b \notin B$. So $Q'$ contains a stable failure $(u, \bar{B})$. By the definition of a stable failure, if $C \subseteq \bar{B}$, then $(u, C)$ is a stable failure of $Q'$ as well. Now $B \subseteq A$ implies $L = \bar{A} \subseteq \bar{B}$. Thus $(u, L)$ is a stable failure of $Q'$.

Conversely, let $(u, L)$ be a stable failure of $Q'$. There is a node $s$ in $Q'$, reached by $u$, such that $s \xrightarrow{a}$ for all $a \in L \cup \{i\}$. Let $A$ consist of all the actions $b$ such that $s \xrightarrow{b}$. We know that $L \subseteq \bar{A}$. In addition, we know the node $\nu_Q$ in $\mathcal{N}(Q)$ which has generated $s$. By examining the rules given in cases 1 and 2, we can conclude that there is a set $B$ in $\text{Acc}(\nu_Q)$ such that $B \subseteq A$. It follows that $(u, B)$ is stable failure in $Q$. But $L \subseteq \bar{A} \subseteq \bar{B}$ and thus $(u, L)$ is a stable failure of $Q$, too.

The only-if part follows directly from the structures of $P'$ and $Q'$. □

The two previous theorems imply the conjecture in the CFFD-semantics.

In order to clarify the construction, we present a larger example, where both rules are applied at the same time. So let $P$ and $Q$ be the following transition systems:
The next pair of figures shows the acceptance graphs of $P$ and $Q$. The sets in $\text{Acc}(\nu)$ are presented beside a node. If no acceptance set has been written, it means that $\text{Acc}(\nu)$ is empty.
The corresponding states in the two acceptance graphs above have been marked with the same number.

The third figure pair represents the result of the construction. Because $Q$ will be quite large, we present first $P'$ and after it $Q'$. We apply case 1 with the states 1, 3, 4, 5, 6 and 7. Case 2 is applied with the states 2. The states marked with a comma originate in the subgraph $\text{fwd}(s(2))$. We have used $i$ instead of $\tau$. We use an optimization. If a node contains only one acceptance set and only transitions with the action in that set as the situation is with the states 1 and 3, for example, then we do not draw an $i$-transition $s(1) \xrightarrow{i} s_1(1) \xrightarrow{a} s(2)$, but leave it out drawing only $s(1) \xrightarrow{a} s(2)$. As a matter of fact, the example gives a hint how to make the construction more minimal.
\[
s(1) \quad a \quad \downarrow \quad b \quad \downarrow \quad c \quad \downarrow \quad i \quad i \quad \downarrow \quad d \quad e \quad f \quad \downarrow \quad s(4) \quad s(5) \quad s(6) \quad s(7)
\]
Chapter 9

Action refinement in full Lotos

9.1 About full Lotos

In full Lotos, processes can send various type of data to each other and to an environment. A process can decide its next action on the basis of received data. A process can manipulate data in a similar way as in traditional programming languages. The essential difference from traditional languages is that data types are defined using the algebraic specification language ACT ONE (see for example [Ehrig and Mahr 85]). Data manipulating operators are defined by the same formalism.

An action in full Lotos consists of a gate name and zero or more input and output operations. A general action is presented by

\[ ge_1 \cdots e_n, \]

where \( n \geq 0 \), and if \( n > 0 \), \( 1 \leq i \leq n \), then \( e_i \) is either of the form \(?x:T\) or \(!E\). Here \(?x:T\) means that \( x \) is a variable of type \( T \) and a process is expecting a value for \( x \) through the gate \( g \). The notation \(!E\) means that a process is sending a value of the expression \( E \) through the gate \( g \) to other processes. We call \( e_1 e_2 \cdots e_n \) a gate list.

If \( ge_1 \cdots e_n \) and \( hd_1 \cdots d_m \) are general actions with gate names \( g \) and \( h \), then they can synchronize with each other if and only if \( g = h \), \( m = n \) and \( e_i \) can synchronize with \( d_i \) as in the following table:

<table>
<thead>
<tr>
<th>( e_i )</th>
<th>( d_i )</th>
<th>Synchronizing condition</th>
<th>Type of interaction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>( g!E )</td>
<td>( g!D )</td>
<td>( \text{value}(E) = \text{value}(D) )</td>
<td>value matching</td>
<td>E1</td>
</tr>
<tr>
<td>( g!E )</td>
<td>( g?x:T )</td>
<td>( \text{value}(E) ) is of sort ( T )</td>
<td>value passing</td>
<td>E2</td>
</tr>
<tr>
<td>( g?x:T )</td>
<td>( g?y:U )</td>
<td>( T = U )</td>
<td>value generation</td>
<td>E3</td>
</tr>
</tbody>
</table>

The effect E1 is a synchronization without side effects, E2 means synchronization with a substitution \( x = \text{value}(E) \) and E3 synchronization with the substitutions \( x = y = v \), where \( v \) is an arbitrary element in \( T \).
A labeled transition system can be generated from the full Lotos specification. In principle, the transitions are deduced in the same way as in the case of basic Lotos, but there are many additional details which affect their generation. Here we present only the most important rules.

Suppose $P$ is a process and $P \xrightarrow{ge} P'$ is a possible transition from $P$. If $P$ represents a state in a transition system, then the state $P$ has one of the following transitions:

1. If $ge$ is $g!E$, then the transition is $P \xrightarrow{a} P'$, where $a = g <!v>$ and $v$ is the value of the expression $E$.

2. If $ge$ is $g?x:T$, then the state $P$ has all the transitions $P^{g<\text{value}(x)>} P'_v$, where $v$ runs through the values in $T$ and $P'_v$ is $P'$ with $x$ having the value $v$.

These rules easily generalize in the case, where there are more than one input or output operation in the action.

**Example.** Let $B$ be the Boolean type with values true and false. Consider the following full Lotos process:

$$P := a!true; \ b?x:B; \ stop \ [] \ a!false; \ c?y:B;z:B; \ stop$$

Denote by $t$ the boolean value true and by $f$ the boolean value false. Then the Lotos expression $P$ generates the transition system

```
  P
 / \                   / \                      / \      / \       / \   / \    
 a!t | a!f             c?f!f | c?f!t stop     c?f!f | c?f!t stop
      / \               / \      / \     / \    / \    / \    
 b ?x:B; stop | b !t | b !f | stop | b !t | b !f | stop | stop | stop
```

The parallel operator is very important in our approach and that’s why we still show in detail how to generate the transition system in presence of that operator. So suppose $P = Q.B|R$ and $Q^{ge} Q', R^{ge} R'$, $g \in B$.

1. If $e = !E$, $d = !D$ and $\text{value}(E) = \text{value}(D) = v$, then there is the transition $P^{g<\text{value}(E)>} P'$ from the state $P$.

2. If $e = !E$, $d = ?x:T$, $v = \text{value}(E)$ and $v$ belongs to the type $T$, then there is the transition $P^{g<\text{value}(E)>} P'_v$ from $P$.

3. If $e = ?x:T$, $d = ?y:T$, then there are all the transitions $P^{g<\text{value}(x)>} P'_v$ from $P$, where $v$ runs through the type $T$ and $P'_v$ is $Q'|B|R'$ such that $x$ and $y$ have the value $v$ in $Q'$ and $R'$.
9.2. PARTIALLY DEFINED ACTIONS

The generalization to the case where there are several input or output operations in one action is again straightforward. Thus in a transition system all the transitions are of the form

\[ g !v_1, !v_2, \ldots, !v_n, \]

where \( g \) is a gate name and \( v_i, i = 1, \ldots, n \) are values of declared types.

9.2 Partially defined actions

Consider a full Lotos action

\[ ge_1 \cdots e_n, \]

with a gate name \( g \) and gate list \( e_1 \cdots e_n \). The whole action is atomic in the interleaving semantics. Thus if the action is being executed, all the events in the gate list are performed before any other action can proceed. This property makes it possible also to refine actions in a restricted form in full Lotos. In large specifications with the process `udfn` we probably do not know all the items in every gate list before we have reduced the number of partially defined states in several refinement steps. Unknown or undefined \( e_i \)'s will be used in partially defined states, but other \( e_i \) may be essential in the totally defined parts of the specification.

Suppose we start to design a system and we know that it consists of several components

\[ S = P_1 | B_1 | P_2 | B_2 | P_3 \cdots P_n | B_n | P_{n+1}. \]

In the start all or nearly all the processes \( P_i, i = 1, \ldots, n + 1 \), are partially defined. In this chapter we study if it is possible to keep gate lists partially defined as well. So the most general situation is as follows.

Processes \( P_i \) are partially defined. They execute actions of the form \( ge_1 \cdots e_n \), where the gate list is only partially defined. By this we mean that some \( e_i \) are missing and we mark that something is missing. It would be too far-reaching in this thesis to consider the situation, where a single \( e_i \) is only partially defined (for example, the type of a variable is left partially open). Two processes may synchronize with each other by performing partially or totally defined actions. Lotos allows multisynchronization so that several processes can synchronize at the same time using the same gate. A flexible refinement is such that one can refine one state or one action in one process or independently several states and several actions in various processes.

First we must decide what kind of partially defined actions we have and how partially defined actions can synchronize with each other. As for partially defined actions, we have at least three possibilities:

1. A partially defined action is of the form

\[ ge_1 * e_3 e_4 * * e_7 *. \]
Here we know that the action contains eight operations. The operations 2, 5, 6 and 8 are missing, \( e_1, e_3, e_4 \) and \( e_7 \) are complete.

2. Or the action can be

\[ ge_1e_2e_3 \star. \]

Here the operations \( e_1, e_2 \) and \( e_3 \) are complete, but the action perhaps contains something else after these operations. We don’t know how many operations there will be when the action is totally defined, but the new operations will be added after \( e_3 \).

3. Combination of the previous two:

\[ ge_1 \star e_3e_4 \star e_6 \perp. \]

Here the operations 2 and 5 are missing. Moreover, one or more operations may be added after \( e_6 \).

The first alternative seems restrictive. When processes are partially defined, we do not yet know exactly the structure of the gate list. That is why the second alternative seems more realistic whereas the third would be most flexible. However, in this thesis, we consider only the second alternative and examine how far it is sufficient.

So a partially defined action is an action of the form

\[ ge_1e_2 \cdots e_n \star, \quad n \geq 0, \]

where the star represents that something is still missing in the gate list. Instead of the star \( \star \), we could use our old symbol \( \downarrow \), but we have chosen the star, because \( \star \) and \( \downarrow \) are used in different ways in what follows.

The next question is what kind of partially defined actions can synchronize with each other. Let

\[ g_1d_1 \cdots d_m \star, \quad g_2e_1 \cdots e_n \star \]

be two partially defined actions. We have two possibilities:

1. The two actions can synchronize, if \( g_1 = g_2 \), \( m = n \) and \( d_i \) match with \( e_i \) in the usual sense, \( i = 1, \cdots, n \).

2. The actions can synchronize, if \( g_1 = g_2 \), \( m \leq n \) and \( d_i \) match with \( e_i \), \( i = 1, \cdots, m \) or \( n \leq m \) and \( e_i \) match with \( d_i \), \( i = 1, \cdots, n \).

The following example is instructive. Let \( P, Q \) and \( S \) be the following processes:

\[
P: \quad \rightarrow P1^{g<1>} \rightarrow P2
\]

\[
Q1 \xrightleftharpoons{g<12>} Q2
\]

\[
\begin{array}{c}
\xrightarrow[g<13>]{Q3}
\end{array}
\]

\[
S: \quad S1^{g<12>} \rightarrow S2
\]
Suppose $S$ is a service description and the system under construction is $P[\langle g \rangle] Q$. The process $Q$ is totally defined both in respect with actions and states. On the other hand, $P$ is under construction and in this phase only partially defined. Evidently, $P$ can be completed so that $P[\langle g \rangle] Q \approx S$ for most equivalences $\approx$.

If we apply the first alternative in the synchronization of partially defined actions, the system $P[\langle g \rangle] Q$ is the same as stop and this is not in a refinement relation with the service. In the case of the second alternative, the system would be

\[
PQ1 \xrightarrow{g^{<12>} >} PQ2
\]

\[
PQ2 \xrightarrow{g^{<13>} >} PQ3
\]

if we make the natural assumption that the resulting transitions after the synchronization of $g^{<12>}$ and $g^{<*>}$, $g^{<13>}$ and $g^{<*>}$ are as depicted. So in this case too the system is not in any relation with the service. All this seems to suggest that the system cannot be completed so that it is equivalent with $S$, a contradiction with reality.

As we have seen, both approaches are insufficient without some restrictions. The solution we shall follow is based on the first principle. In order to the actions $ge_1 \cdots e_m*$ and $gd_1 \cdots d_n*$ to be able to synchronize, we must have $m = n$ and $e_i$ matches with $d_i$, $i = 1, \cdots, n$. We pose an additional condition that synchronizing actions must be in the same phase in every refinement step. This means the following. Suppose $P$ is a partially defined system, $P \sqsubseteq S$ and we are going to refine $P$ so that at the end $P \approx S$. If $g$ is a gate somewhere in $P$ and we refine the gate list of this $g$, then we must refine every other gate list of $g$ in $P$ in the same way before we check the condition $P \sqsubseteq S$ again.

This requirement can be checked during the parsing phase already and a warning can be given, if this rule is not followed. This restriction also solves problems with compositional property of the parallel operator. Without the restriction, the action refinement cannot happen compositionally with respect to the parallel operator, if we apply the first principle in the synchronization. For instance, if $P$, $Q$ and $R$ are processes, $P \sqsubseteq Q$ for some intuitively relevant refinement relation $\sqsubseteq$ and action refinement is in a different phase in $P$ and $Q$, it may occur that $P|B|R \not\sqsubseteq Q|B|R$ as the following example shows [Kai 97]:

\[
P : \quad \rightarrow \quad P1 \xrightarrow{g^{<12,13>} >} P2
\]

\[
P : \quad \rightarrow \quad P3
\]

\[
Q : \quad \rightarrow \quad Q1 \xrightarrow{g^{<12,13>} >} Q2
\]

\[
Q : \quad \rightarrow \quad Q3
\]

\[
R : \quad \rightarrow \quad R1 \xrightarrow{g^{<12,14>} >} R2
\]
Now \( P \subseteq Q \), but \( P || g || R = \text{stop} \) and \( Q || g || R \) is \( g!2!3; \text{stop} \). So \( P || g || R \nsubseteq Q || g || R \) for any intuitively relevant refinement relation \( \subseteq \) and thus \( \subseteq \) is not compositional with respect to the parallel operator in presence of partially defined actions in different phases.

These kinds of refinements are, however, now forbidden, because actions are in a different phase. Theoretically, the solution is not completely satisfactory, but we apply the method in a concrete situation and show that in practice it has merits. Moreover, this solution makes it possible to use existing software packages for Lotos, because transition systems are generated now in the same way as without partially defined actions.

In Lotos specifications we use \(!*\) to denote a partially defined part in an action. For example, if \( g \) is a gate name and \( T \) a data type, we can write \( g!* \), \( g!2!* \), \( g?x:T!* \), \( g!2?x:T!* \) etc. In diagrams, theory and explanations we may use only the star \(*\) without the exclamation mark \(!\), but in complete Lotos specifications we must use both, because standard Lotos demands either \(!\) or \(?\). By using \(!\) our specifications differ minimally from the standard syntax.

### 9.3 Generalized BKO-refinement

Our aim in this section is to develop a BKO-refinement relation which allows partially defined states and actions, and which coincides with the earlier BKO-refinement, if all the actions are totally defined. Consider some examples first.

**Example 1.** Let \( P \) and \( Q \) be processes

\[
P : \quad \rightarrow P1 \xrightarrow{g<12>} P2, \quad \quad Q : \quad \rightarrow Q1 \xrightarrow{g<12,13>} Q2.
\]

We should have \( P \subseteq_{\text{BKO-ref}} Q \), because the action \( g<12> \) in \( P \) can be completed to the action \( g<12,13> \), after which \( P \) is exactly \( Q \). \( \square \)

**Example 2.** Let \( P \) and \( Q \) be processes

\[
\begin{array}{c}
\rightarrow P1 \xrightarrow{g<12>} P2 \uparrow \\
\quad \quad \quad \quad P3 \xrightarrow{a} P4
\end{array} \quad \rightarrow \quad Q1 \quad \quad \quad \quad Q2 \xrightarrow{a} Q3
\]

Again it should be \( P \subseteq_{\text{BKO-ref}} Q \), because \( P \) can be completed so that it will be equivalent with \( Q \). \( \square \)

**Example 3.** Let \( P \) and \( Q \) now be
9.3. GENERALIZED BKO-REFINEMENT

\[ \rightarrow P_1 \xrightarrow{\delta^{\langle i \rangle}} P_2, \longrightarrow Q_1 \xrightarrow{g^{\langle 12 \rangle}} Q_2 \]
\[ \quad \downarrow g^{\langle 13 \rangle} \]
\[ \quad Q_3 \]

Now \( P \not\models_{BKO_{ref}} \ Q \) should hold, because \( P \) cannot be completed to \( Q \). \( \square \)

These examples show that we must compare traces in a new way. Especially the example 3 shows that we must be careful, if we are going to get an intuitive refinement relation. The next definition shows the extra concepts we need in order to develop the BKO-refinement in the presence of partially defined actions. We formulate the concepts for transition systems. That is why it is enough to consider only the actions of the form \( g!e_1!, \ldots, !e_n! \). Actions with \( ? \) are not needed.

**Definition 9.3.1** Let \( u, v \in \text{tr}(P) \) for some full Lotos process \( P \). The trace \( v \) is a supertrace of \( u \), or \( u \) is a subtrace of \( v \), written \( u \leq v \), if

1. \( u = a_1a_2 \cdots a_n, \ v = b_1b_2 \cdots b_n \),
2. \( a_i \) and \( b_i \) are totally or partially defined actions in full Lotos, \( i = 1, \ldots, m, \)
3. for every \( i, \ 1 \leq i \leq n, \)
   a) either \( a_i = b_i \), or
   b) \begin{align*}
   a_i & = g^{<v_1, v_2, \ldots, v_k, >,} \\
   b_i & = g^{<v_1, v_2, \ldots, v_l, >,} \\
   k & \leq l,
   \end{align*}
   or
   c) \begin{align*}
   a_i & = g^{<v_1, v_2, \ldots, v_k, >,} \\
   b_i & = g^{<v_1, v_2, \ldots, v_l, >,} \\
   k & < l.
   \end{align*}

We say that an action \( a \) is a subaction of an action \( b \) or \( b \) is a superaction of \( a \), if \( a \) is a subtrace of \( b \).

**Definition 9.3.2** Let \( L \) and \( M \) be finite sets of partially or totally defined full Lotos actions. The sets \( L \) and \( M \) are action equivalent, \( L \approx_a M \), if \( L \) and \( M \) have the same number of elements, every action \( a \in L \) is a subaction of an action \( b \in M \) and every action \( b \in M \) is a superaction of an action \( a \in L \).

So instead of traces we must consider super- or subtraces. In addition, the example 3 on page 116 shows that it is necessary also to compare the outgoing
transitions of two different states. Before we did this by comparing failure sets, but now this is not sufficient. Instead of failure sets, we consider acceptance sets, which are complements of failure sets. If \((u, L)\) is a stable failure, we denote the complement of \(L\) by \(\bar{L}\). The complement \(\bar{L}\) represents those actions that could be executed after the trace \(u\).

**Definition 9.3.3** A finite full Lotos process \(Q\) is \(\text{BKO-refinement of a finite full Lotos process} P, P \sqsubseteq_{\text{BKO-ref}} Q, \) if

1. for every \(u \in \text{tr}(P)\) there is a supertrace \(v \in \text{tr}(Q)\); for every \(v \in \text{tr}(Q)\) there is a subtrace \(u\) such that either \(u \in \text{tr}(P)\) or some prefix \(u_1\) of \(\bar{u}\) belongs to \(\text{partr}(P)\);
2. for every \(v \in \text{partr}(Q)\) there is a subtrace \(u\) of \(v\) such that some prefix of \(u\) belongs to \(\text{partr}(P)\);
3.

\[
\begin{align*}
P \Downarrow & \implies \text{stable}(P) \iff \text{stable}(Q)), \\
P \Uparrow & \implies (\sim \text{stable}(P) \iff \sim \text{stable}(Q));
\end{align*}
\]

4. for every \((u, L) \in \text{sfail}(P)\) there is \((v, M) \in \text{sfail}(Q)\) such that \(v\) is a supertrace of \(u\) and \(\bar{L} \approx_a \bar{M}\); for every \((v, M) \in \text{sfail}(Q)\) either there is \((u, L) \in \text{sfail}(P)\) such that \(u\) is a subtrace of \(v\) and \(\bar{L} \approx_a \bar{M}\) or there is a prefix \(u_1\) of \(v\) and a subtrace \(u_1\) of \(v_1\) such that \(u_1 \in \text{partr}(P)\).

The condition for sfail is more complicated than previously, because we want to exclude relations \(P \sqsubseteq_{\text{BKO-ref}} Q\) for such processes as in example 3 on page 116. The finiteness requirement is essential, because it is not possible to compare infinite sets with respect to action equivalence precisely enough.

**Proposition 9.3.1** The modified \(\text{BKO-refinement} is a preorder.

**Proof.** The proof proceeds in the same way as with the original \(\text{BKO-refinement}. The only addition is the observation that a subtrace of a subtrace is a subtrace of the original trace. \(\square\)

The modified \(\text{BKO-reflation} is evidently compositional with respect to the Lotos operators, if all the synchronizing actions are in the same refinement phase.

### 9.4 About modified bisimulation refinement

Surprisingly, the definition of the bisimulation refinement is much more difficult to extend to partially defined actions. The reason can be seen from the following example.

Let \(P\) and \(Q\) be the processes

\[
\begin{align*}
9.5 IMPLEMENTATION OF BKO- AND CFFD-REFINEMENT RELATIONS WITH ACTION REFINEMENT

The first condition in the definition of the prebisimulation is clear, but the second condition is unsatisfactory in the presence of partially defined actions. Namely, \( P \downarrow g < v_1, * > \), but which of the actions \( g < v_1, v_2, v_3 > \) and \( g < v_1, v_2, v_4 > \) in \( Q \) to choose? For example, the following definition has drawbacks:

**Definition 9.4.1** For two processes \( P \) and \( Q \), \( P \sqsubseteq_{\text{wbieref}} Q \), if there is a set \( \mathcal{R} \) of process pairs such that \((P, Q) \in \mathcal{R}\) and for every \((P_1, Q_1) \in \mathcal{R}\):

- if \( P_1 \xrightarrow{a} P_2 \) then there is a \( Q_2 \) and a superaction \( \tilde{a} \) of \( a \) such that \( Q_1 \xrightarrow{\tilde{a}} Q_2 \) and \((P_2, Q_2) \in \mathcal{R}\);
- if \( Q_1 \uparrow a \) then \( Q_1 \uparrow \tilde{a} \) for some subaction \( a \) of \( \tilde{a} \) or there is a \( P_2 \) and a subaction \( a \) such that \( P_1 \xrightarrow{a} P_2 \) and \((P_2, Q_2) \in \mathcal{R}\);
- if \( Q_1 \uparrow \tilde{a} \) then \( P_1 \uparrow a \) for some subaction \( a \) of \( \tilde{a} \).

However, the following example shows that the definition does not correspond with the intuition:

\[
P : \rightarrow P1^{g<e_1,e_1>}P2 \uparrow \quad Q : \quad \rightarrow Q1^{g<e_1,e_2>}Q2 \uparrow
\]

Now \( P \sqsubseteq_{\text{wbieref}} Q \), but \( P \) cannot be completed so that it is equivalent with \( Q \). It seems that there is no straightforward, intuitive and flexible solution to this problem. The only way would be to equip the states with the information about acceptance sets, but this would destroy the pure structure of the bisimulation relation. That is why we do not develop this further in this study.

9.5 Implementation of BKO- and CFFD-refinement relations with action refinement

The action refinement does not influence the BKO- and CFFD-refinement algorithms much. The only differences are that we must consider sub- and supertraces instead of ordinary traces and we must compare action sets when checking conditions for stable failures. Sub- and supertraces can be taken care of in the procedure Test-traces. When considering a transition \((R, a, S)\) in line 2, we must find a sub- or supertrace \((R', a', S')\). It is an easy programming task to check, if \( a' \) is a sub- or superaction of \( a \). Thus, we can find sub- and supertraces by examining single actions in Test-traces.
The other modification in the algorithm is that we must compare, if two action sets satisfies the conditions of Definition 9.3.2. This is a bit time-consuming, but also easily programmed.
Chapter 10

Bounded Retransmission Protocol

10.1 Description of the protocol

The bounded retransmission protocol, or shortly BRP, is used by Philips in one of its products. It is a sufficiently complicated protocol to illustrate practical refinement-based design well. It has been studied using different approaches, for example in [Groote and van de Pol 93], [Helmink, Sellink and Vaandrager 94], [Havelund and Shankar 96] and [D’Argénio, Katoen, Ruys and Tretmans 96]. We follow the Lotos-based approach presented in [Mateescu 96]. The general description of the protocol is, however, from [D’Argenio, Katoen, Ruys and Tretmans 96].

Our purpose is to apply our BKO-theory and software. So we develop the protocol in steps and verify every step. The protocol is modelled using full Lotos. First we apply only partially defined states. After that we start all over again and design the protocol in steps using action refinements, too. The transition systems are generated with the help of CÆSAR/ALDEBARAN Development package [CADP 96].

Our method of refinement is not restricted to particular equivalences. The design could be done in the same way with for example bisimilarity. A designer must only to choose the equivalence and related refinement relation before starting to refine. Which one of the equivalences to choose is a matter of taste or sometimes the application dictates the choice.

10.1.1 Informal protocol specification

The bounded retransmission protocol consists of a sender $S$ equipped with a timer $T_1$, and a receiver $R$ equipped with a timer $T_2$. The sender and receiver exchange messages via two unreliable (lossy) channels, $K$ and $L$. Sender $S$ reads a list to be transmitted and sets the retry counter to 0. Then it starts to
send the elements in the list one by one over \( K \) to \( R \). Timer \( T_1 \) is set and a frame is sent into channel \( K \). This frame consists of bits and a datum (=chunk). The first bit indicates whether the datum is the first element in the list. The second bit indicates whether the element is the last item in the list. The third bit is the so-called alternating bit that is used to guarantee that data is not duplicated. After having sent the frame, the sender waits for an acknowledgement from the receiver, or for a timeout. In case an acknowledgement arrives, the timer \( T_1 \) is reset and (depending on whether this was the last element of the list) the sending client is informed of correct transmission, or the next element of the list is sent.

If timer \( T_1 \) times out, the frame is resent (after the counter for the number of retries is incremented and the timer is set again), or the transmission of the list is broken off. The latter occurs if the retry counter exceeds its maximum value \( \text{MAX} \).

Receiver \( R \) waits for a first frame to arrive. This frame is delivered at the receiving client, timer \( T_2 \) is started and an acknowledgement is sent over \( L \) to \( S \). Then the receiver simply waits for more frames to arrive.

The receiver remembers whether the previous frame was the last element of the list and the expected value of the alternating bit. Each frame is acknowledged, but it is handed over to the receiving client only if the alternating bit indicates that it is new. In this case timer \( T_2 \) is reset. Note that (only) if the previous frame was the last of the list, then a fresh frame will be the first of the subsequent list and a repeated frame will still be the last of the old list.

This goes on until \( T_2 \) times out. This happens if, for a long time, no new frame is received, indicating that transmission of the list has been given up. The receiving client is informed, provided the last element of the list has not just been delivered. Note that if transmission of the next list starts before timer \( T_2 \) expires, the alternating bit scheme is simply continued. This scheme is only interrupted after a failure.

Timer \( T_1 \) times out if an acknowledgement does not arrive "on time" at the sender. It is set when a frame is sent and reset after this frame has been acknowledged. (We assume that premature timeouts are not possible, i.e. a message must not come after the timer has expired.)

Timer \( T_2 \) is (re)set by the receiver at the arrival of each new (nonduplicate) frame. It times out if the transmission of a list has been interrupted by the sender. To decide the size of the timeslice of \( T_2 \) in a real implementation, one must take into account the timeslice of \( T_1 \), the number of retransmissions, the communication speed and the processing times at the sender and receiver. However, in Lotos we don't have explicit time at our disposal, so we must try to model the above situation in another way. How to do this is left to the step, where we design the timers. We assume that the sender does not start reading and transmitting the next list before the receiver has properly reacted to the failure. This is necessary, because the receiver has not yet switched its
alternating bit, so a new frame would be interpreted as a repetition.

10.1.2 Informal service specification

As we have seen, the input is a large data packet that is modeled as a list \( L = (d_1, \ldots, d_n) \). The input \( L \) is read on the "input" port. Ideally, each \( d_i \) is delivered onto the output port. Each chunk \( d_i \) is accompanied by an indication. This indication can be \( I\_\text{FST} \), \( I\_\text{INC} \) or \( I\_\text{OK} \). \( I\_\text{OK} \) is used if \( d_i \) is the last element of the list. \( I\_\text{FST} \) is used if \( d_i \) is the first element of the list and more will follow. All other chunks are accompanied by \( I\_\text{INC} \).

However, when something goes wrong, a "not OK" indication \( I\_\text{NOK} \) is delivered without datum. Note that the receiving client does not need a "not OK" indication before delivery of the first chunk nor after delivery of the last one.

The sending client is informed after transmission of the whole list, or when the protocol gives up. An indication is sent out on the input port. This indication can be \( I\_\text{OK} \), \( I\_\text{NOK} \) or \( I\_\text{DK} \). After an \( I\_\text{OK} \) or \( I\_\text{NOK} \) indication, the sender can be sure that the receiver has the corresponding indication. A "don't know" indication \( I\_\text{DK} \) may occur after delivery of the last-but-one chunk \( d_{n-1} \). This situation arises, because no realistic implementation can be certain in all cases whether the last chunk was lost. The reason is that information about a successful delivery has to be transported back somehow over the same unreliable medium. In case the last acknowledgement fails to come after the maximal number of retransmissions, there is no way to know whether the last chunk \( d_n \) has been delivered or not. This is indicated by sending \( I\_\text{DK} \), after which the protocol is ready to transmit a subsequent list.

This completes the original informal description of the file transfer service. We remark that it is unclear from this service description which indication the sending client receives in case the receiving client does not receive any chunk. Since something went wrong an \( I\_\text{NOK} \) indication is required, but from this indication the sending client may not deduce that the receiving client has the corresponding indication. This is because the receiving client does not receive an \( I\_\text{NOK} \) indication before delivery of the first chunk. In the Lotos specification we follow this somewhat incomplete description. So, if the sending client receives an \( I\_\text{NOK} \), either the receiving client received the same or did not receive anything at all.

10.2 Lotos description of BRP service

10.2.1 Data structures

The data packets produced by the sending client are modeled by the Lotos data type \texttt{PACKET}. It is a list type with typical list operations \texttt{cons}, \texttt{head}
and tail. In addition, the operation \texttt{len} gives the length of a list, the operation \texttt{cons\_pack} constructs a data packet with the help of the operation \texttt{cons\_packet}. Moreover, the maximum number of repetitions \texttt{max} is included in the data type \texttt{PACKET}.

The data type \texttt{INDICATION} contains the elements \texttt{I\_FST}, \texttt{I\_INC}, \texttt{I\_OK}, \texttt{I\_NOK} and \texttt{I\_DK} which were explained in the informal service specification. The auxiliary functions \texttt{ind} and \texttt{conf} are used to calculate the indication for the receiving client and the confirmation for the sending client.

The types \texttt{Boolean} and \texttt{NaturalNumber} are taken from the Lotos libraries given in the ISO standard [ISO 88]. The special comment (** constructor *) is used to explicitly indicate the constructor operators to the CAESAR.ADT compiler. Constructor operations are basic elements in a data type. Other elements are constructed from constructors with the help of other operations. No equations are allowed between constructors.

type DATA is NaturalNumber
    sorts Data
    opns data (**constructor*) : Nat \to Data
endtype

type PACKET is Boolean, NaturalNumber, DATA
    sorts Packet
    opns nil (**constructor*) : \to Packet
    cons (**constructor*) : Data, Packet \to Packet
    head : Packet \to Data
    tail : Packet \to Packet
    len : Packet \to Nat
    max : \to Nat
    cons\_packet : Nat \to Packet
    cons\_pack : Nat, Packet \to Packet

    eqns forall P:Packet, D:Data, N:Nat
    ofsort Data
        head (cons (D, P)) = D;
    ofsort Packet
        tail ( cons (D, P)) = P;
    ofsort Nat
        len (nil) = 0;
        len (cons (D, P)) = len (P) + 1;
    ofsort Nat
        max = succ (9);
10.2. LOTOS DESCRIPTION OF BRP SERVICE

ofsort Packet
   cons_packet (N) = cons_pack (N, nil);
   cons_pack (0, P) = P;
   cons_pack (succ (N), P) =
      cons_pack (N, cons (data (succ (N)), P));
endtype

A data packet received from the sending client is for example
   cons(data(1), cons(data(2), cons(data(3)))).

Here the packet contains three chunks, but in principle the number of chunks
 can be arbitrary.

type INDICATION is PACKET

sorts Ind

ops I_FST (*! constructor *),
    I_INC (*! constructor *),
    I_OK (*! constructor *),
    I_NOK (*! constructor *),
    I_DK (*! constructor *): -> Ind
    conf : Packet -> Ind
    ind : Bool, Bool -> Ind

eqns forall P:Packet, B: Bool
   ofsort Ind
      len (P) == 1 => conf (P) = I_DK;
      conf (P) = I_NOK;
      ofsort Ind
      ind (true, false) = I_FST;
      ind (false, false) = I_INC;
      ind (B, true) = I_OK;
endtype

The indication is computed cleverly from the length of the current list ([Matesescu 96]). For example, if the length of the list is greater than one and the
 first chunk is sent, the indication will be
   ind(len(P) == L, len(P) == 1) = ind(true, false) = I_FST.

We present in this section also the data type SIGNAL which is used in the
 protocol to synchronize behaviour with the timers. Its elements are explained
 in the protocol description.
type SIGNAL is
  sorts Sig
  opns START (⋆! constructor ⋆),
      RESET (⋆! constructor ⋆),
      TIMEOUT (⋆! constructor ⋆),
      S_READY (⋆! constructor ⋆),
      R_READY (⋆! constructor ⋆) : -> Sig
endtype

10.2.2 BRP service

The data packets produced by the sending client are read at the INPUT gate. After reading a packet, a decision is made either to reject the packet if it is empty and to go back to the initial state or to accept it and begin its transmission. The transmission is modeled by a call to the SERVICE_1 process.

process SERVICE[INPUT, OUTPUT] : noexit :=
  INPUT?P:Packet;
  ( [len(P) == 0] -> SERVICE[INPUT, OUTPUT]
   []
   [len(P) > 0] -> SERVICE_1[INPUT, OUTPUT](P, len(P))
  )
endproc

The SERVICE_1 process has two value parameters: P, which represents the portion of the packet that remains to be transmitted (initially the whole packet), and L, which stands for the length (i.e. number of chunks) of the whole packet. The service follows closely the informal service specification. Notice only the technical detail in dealing with nondeterminism. The i action is used to prevent the sending client from forcing a successful result of a transmission. For example, the choice between the rendezvous "INPUT !I_OK" and "INPUT !I_DK" below cannot be influenced by the sending client, since the protocol can always perform non-deterministically one of the silent actions i.

Also note the role of the parameters L and P in S_1. The parameter L is the original length of the packet and P is the remaining list of chunks. Indications can be computed with the help of these two parameters.

process SERVICE_1[INPUT, OUTPUT] (P:packet, L:Nat) : noexit :=
  i; OUTPUT !head(P) !ind(len(P) == L, len(P) == 1);
  ( [len(P) == 1] ->
    ( (* last chunk delivered *)
  )
i; INPUT !I_OK; SERVICE[INPUT, OUTPUT]  
[]  
i; INPUT !I_DK; SERVICE[INPUT, OUTPUT]  
[]  
[len(P) > 1] ->  
(* more chunks to deliver *)  
i; SERVICE_1[INPUT, OUTPUT](tail(P), L)  
[]  
i; INPUT !I_NOK; OUTPUT !I_NOK; SERVICE[INPUT, OUTPUT]  
)  
[]  
i;  
[len(P) == L] ->  
(* failure at the first chunk *)  
INPUT !conf(P); SERVICE[INPUT, OUTPUT]  
[]  
[len(P) < L] ->  
(* failure at an intermediate chunk *)  
INPUT !conf(P); OUTPUT !I_NOK; SERVICE[INPUT, OUTPUT]  
)  
enproc

10.3 Design of BRP

10.3.1 The first step

The main parts of the protocol are sender S and receiver R. The sender reads a data packet from the sending client via the INPUT gate. Processes S and R are loosely coupled, so we must design channel processes K and L. The sender sends packets into K and the receiver sends acknowledgements into L. At first the receiver just waits for a data packet. Moreover, two timers are needed and we model them as processes. Thus the overall structure of the system is as follows.

\[
\text{BRP1[INPUT]} := ((S[INPUT] || R) ||| (K ||| L)) |||  
(T1 ||| T2)
\]

where

\[
S[INPUT] := \text{INPUT ?P: Packet; undef}
\]
and all the other processes are udef. We have used the pure interleaving operator $||$ in our first design, because the structure of the processes is unclear in this phase and we do not know yet the synchronizing gates. In later stages, when we have refined the processes more, the parallel operators will contain a synchronizing gate set. We want to show that the design satisfies the condition

$$BRP1[INPUT] \sqsubseteq_{BKoref} SERVICE[INPUT, OUTPUT].$$

BRP1 contains the process $INPUT \, ?P: \text{Packet}; \, udef$. The data type Packet is infinite and there is no process sending concrete packets. So our system is infinite. In order to be able to analyse the system automatically, we need the Lotos description of the sending client that sends a finite amount of finite packets. We use the following sending client:

```plaintext
process SENDING_CLIENT [ INPUT ] : noexit :=
    INPUT !cons_packet (N);
    INPUT ?IO: Ind;
    SENDING_CLIENT [ INPUT ]
endproc
```

The auxiliary function $cons\_packet(N)$ constructs a data packet of $N$ chunks respectively numbered from 1 to $N$. In refinements it is enough to use short lists. We have chosen $N$ to be 4. Later, when analysing the complete protocol, the length of the data list may be varied.

The global state graph of the service with $N = 4$ is small. It consists of 36 states and 48 transitions. The minimal determinized graph of the service consists of 7 states and 13 transitions. The graph of the first version of BRP-protocol consists of 3 states and 18 transitions (including $\eta$-transitions). The minimal determinized version consists of 2 states and 3 transitions.

Now we can show with the help of CÆSAR/Aldebaran-system and our own software that

$$SENDING\_CLIENT[INPUT] \sqsubseteq_{BKoref} BRP1[INPUT] \sqsubseteq_{BKoref} SERVICE[INPUT, OUTPUT]$$

### 10.3.2 The second step

We first design the behaviour of the protocol in case where the channels are perfect, i.e. no packets are lost in the channels. We mark those points as undefined that take care of errors in the channels. The sender is as follows:

```plaintext
process S[INPUT, SEND_K, REC_L, T1](ALT: Bool): noexit :=
    INPUT ?P: Packet;
```

[len(P) == 0] -> S[INPUT, SEND_K, REC_L, T1](ALT)
  (* empty packets rejected *)
  □

[ len(P) > 0 ] -> S_1[INPUT, SEND_K, REC_L, T1]
  (ALT, P, len(P), 0)
)
endproc

process S_1[INPUT, SEND_K, REC_L, T1](ALT: Bool, P: Packet, L):
  noexit :=
  SEND_K !(len(P) == L) !(len(P) == 1) !ALT !head(P);
  T1 !START;
  ( REC_L;
    T1 !RESET;
    ( [len(P) == 1] ->
      INPUT !I_OK;
      S [INPUT, SEND_K, REC_L, T1] (not (ALT))
    □
    [ len(P) > 1 ] ->
      S_1 [ INPUT, SEND_K, REC_L, T1 ]
      (not (ALT), tail (P), L, 0)
    □
    undef;
  )
)
endproc

Process S_1 scans the data packet P and sends the data chunks, one by one, into channel K via the SEND_K gate. Each chunk is accompanied by three bits which are modeled as boolean values: the first two bits indicate whether the chunk is the first and/or last of the packet, and the third is the alternating bit. The value parameter L stands for the initial length of the packet. The timer T1 is started via the gate T1. If the sender receives an acknowledgement, the timer is reset and the next chunk is sent. If the previous chunk was the last one, the sending client is informed about the successful delivery with I_OK message and the sender is ready to accept the next packet from the client.

Channel K is simple. It receives data via the gate SEND_K and transmits it at REC_K gate. After reading a data chunk, the channel K may lose it (i.e. do not transmit it at the REC_K gate). We leave for the moment open how to model a loss of a data packet exactly.

process K[SEND_K, REC_K]: noexit :=
  SEND_K ?FST, LST, ALT: Bool ?D: Data;
(i; REC.K !FST !LST !ALT !D; K[SEND.K, REC.K]
   []
i; undef
)
endproc

Now we can start to design the receiver. First it reads a data chunk from K, transmits it to the receiving client, starts timer T2 and sends an acknowledgement into channel L. After this it waits for a new chunk. The next chunk may be a really new one or a resending of the previous chunk. The third alternative is that no message arrives and timer T2 times out, but we leave this undefined in this step.

process R[OUTPUT, REC.K, T2, SEND.L]: noexit :=
   REC.K ?FST, LST, ALT: Bool ?D: Data [FST];
   OUTPUT !D !ind(FST, LST);
   T2 !START;
   SEND.L;
   R_1 [OUTPUT, REC.K, SEND.L, T2] (LST, not (ALT))
endproc

process R_1 [OUTPUT, REC.K, SEND.L, T2] (END, ALT: Bool): noexit :=
   REC.K ?FST, LST: Bool !ALT ?D: DATA;
   (* new chunk *)
   T2 !RESET;
   OUTPUT !D !ind (FST, LST);
   T2 !START;
   SEND.L;
   R_1 [OUTPUT, REC.K, SEND.L, T2] (LST, not (ALT))
[]
   REC.K ?FST, LST: Bool !not(ALT) ?D: DATA;
   (* duplicated chunk *)
   SEND.L;
   R_1 [OUTPUT, REC.K, SEND.L, T2] (LST, ALT)
[]
   undef
endproc

Channel L resembles channel K:

process L[SEND.L, REC.L]: noexit :=
   SEND.L;
   (
i; (* correct transmission *)
   )
10.3. DESIGN OF BRP

RECL;
   L [ SEND_L, RECL]
[]
i; (* loss indication *)
udef
endproc

The structures of the timers are still open for the most part. In Bounded Transmission Protocol, the timers play an essential role and we postpone their design to later steps. We now write only:

process T1[T1]: noexit :=
   T1 !START;
   ( T! !RESET;
      T1 [T1]
      []
      udef
   )
endproc

process T2[T2]: noexit :=
   T2 !START;
   ( T2 !RESET;
      T2 [T2]
      []
      udef
   )
endproc

The whole system is now

BRP2[INPUT, OUTPUT] :=
hide T1, T2, SEND_K, REC_K, SEND_L, RECL in
( ( S[INPUT, SEND_K, RECL, T1] (false) |||
   R[OUTPUT, REC_K, SEND_L, T2] )
   || [SEND_K, REC_K, SEND_L, RECL]|||
   ( K[SEND_K, REC_K] ||| L[SEND_L, RECL] )
)
|| [T1, T2]|
( T1[T1] ||| T2[T2] )
The global state graph of the partially defined BRP-protocol consists of 716 states and 3302 transitions. The minimal determinized version consists of 14 states and 26 transitions. We verify that

\[
\begin{align*}
&\text{SENDING\_CLIENT[INPUT]} \parallel \text{[INPUT]} \parallel \text{BRP1[INPUT]} \\
\subseteq_{BKoref} &\text{SENDING\_CLIENT [INPUT]} \parallel \text{BRP2[INPUT, OUTPUT]} \\
\subseteq_{BKoref} &\text{SENDING\_CLIENT[INPUT]} \parallel \text{SERVICE[INPUT, OUTPUT]}
\end{align*}
\]

### 10.3.3 The third step

Now, we complete the protocol. The packets may be lost in the channels. If the transmission of a chunk fails, no acknowledgement arrives and timer \(T1\) times out. In this case the previous chunk is sent again, if the number of retransmissions is less than \(\max\). After \(\max\) failed attempts \(I\_OK\) or \(I\_DK\) is sent to the sending client.

If the receiver receives no messages for a while, timer \(T2\) may time out. If the last received chunk was not the last in the list, the receiver sends \(I\_NCK\) to the receiving client, resets the timer and waits for a new packet. If the last chunk of the packet has been received, the receiver only resets timer \(T2\) and waits for a new packet.

The process \(S\_1\) becomes:

```
process S_1 [INPUT, SEND_K, REC_L, T1]
    (ALT: Bool, P: Packet, L, RN: Nat) : noexit :=
    SEND_K !(len(P) == L) !(len(P) == 1) !ALT !head(P);
    T1 !START;
    (REC_L;
     T1 !RESET;
     (len(P) == 1] ->
      INPUT !I_OK;
      S [INPUT, SEND_K, REC_L, T1] (not (ALT))
    []
     [len(P) > 1] ->
      S_1 [INPUT, SEND_K, REC_L, T1]
      (not(ALT), tail(P), L, 0)
    )
    []
    T1 !TIMEOUT;
    (RN < max] ->
    S_1[INPUT, SEND_K, REC_L, T1] (ALT, P, L, RN+1)
```

```
[]
[RN == max] ->
    INPUT !conf(P);
    S [INPUT, SEND_K, REC_L, T1] (not(ALT))
)
)

The channels are as follows:

process K[SEND_K, REC_K] : noexit :=
    SEND_K ?FST, LST, ALT: Bool ?D: Data;
    (i; REC_K !FST !LST !ALT !D; K[SEND_K, REC_K]
        []
        i; K [SEND_K, REC_K]
    )
endproc

process L [SEND_L, REC_L] : noexit :=
    SEND_L;
    (i; REC_L; L [SEND_L, REC_L]
        []
        i; L [SEND_L, REC_L]
    )
endproc

Next we complete the receiver:

process R_1 [OUTPUT, REC_K, SEND_L, T2] (END, ALT: Bool) : noexit :=
    REC_K ?FST, LST: Bool !ALT ?D: Data;
    T2 !RESET;
    OUTPUT !D !ind (FST, LST);
    T2 !START;
    SEND_L;
    R_1 [OUTPUT, REC_K, SEND_L, T2] (LST, not (ALT))
        []
    REC_K ?FST, LST: Bool !not (ALT) ?D:Data;
    (* duplicated chunk *)
    SEND_L;
    R_1 [OUTPUT, REC_K, SEND_L, T2] (LST, ALT)
        []
T2 !TIMEOUT;
    (not(END)] ->
(* more chunks to follow *)
OUTPUT !I_NOK;
R [OUTPUT, REC_K, SEND_L, T2]
[]
[END] ->
(* last chunk already delivered *)
R [OUTPUT, REC_K, SEND_L, T2]
)
endproc

The timers are simple:

process T1 [T1] : noexit :=
T1 !START;
(
  T1 !RESET;
  T1 [T1]
[]
  T1 !TIMEOUT;
  T1 [T1]
endproc

process T2 [T2] : noexit :=
T2 !START;
(
  T2 !RESET;
  T2 [T2]
[]
  T2 !TIMEOUT
  T2 [T2]
)
endproc

But now the system is not BKO-equivalent with the service. There are deadlocks in the system and because of this failure sets are different. The system is quite large with 29251 states and 90123 transitions, although we used a list of length one and retransmission number 1. The first deadlocked state is 649, so it is not easy to see the exact reason for the deadlock. However, it is possible to minimize the system. We determinized the system and the result shows that some received chunks are not sent to the receiving client. The obvious reason for this is that the timers are too simple. In fact, the informal specification of the protocol states that T1 should not timeout too hastily and T2 should timeout only after packets of a list are no more coming.
10.3.4 The fourth step

We design timer $T_1$ completely, but timer $T_2$ is left partially undefined. Timer $T_1$ should time out only if a chunk has been lost in the channel. Because standard Lotos has no timing constraints, we model time with an auxiliary gate LOST. The channels $K$ and $L$ signal timer $T_1$ that a chunk or an acknowledgement has been lost and after this $T_1$ can timeout. We also leave those parts in $S$ and $R$ undefined which depend on $T_2$.

process K [SEND_K, REC_K, LOST] : nexit :=
    SEND_K ?FST, LST, ALT: Bool ?D: Data;
    i;
    REC_K !FST !LST !ALT !D;
    K [SEND_K, REC_K, LOST]
[]
i;
LOST;
    K [SEND_K, REC_K, LOST]
) endproc

process L [SEND_L, REC_L, LOST] : nexit :=
    SEND_L;
    (i;
     REC_L;
     L [SEND_L, REC_L, LOST]
[]
i;
LOST;
     L [SEND_L, REC_L, LOST]
) endproc

process T1 [T1, LOST] : nexit :=
T1 !START;
( T1 !RESET;
    T1 [T1, LOST]
[]
LOST;
    T1 !TIMEOUT;
T1 [T1, LOST]
)
endproc

process T2 [T2] : noexit :=
T2 !START;
(
T2 !RESET;
T2 [T2]
[]
udef
)
endproc

process S_1 [INPUT, SEND_K, REC_L, T1]
  (ALT: Bool, P: Packet, L, RN: Nat) : noexit :=
SEND_K !(len(P) == L) !(len(P) == 1) !ALT !head(P);
T1 !START;
(
REC_L;
T1 !RESET;
(
[len(P) == 1] ->
INPUT !I_OK;
S [INPUT, SEND_K, REC_L, T1] (not (ALT))
[]
[len(P) > 1] ->
S_1 [INPUT, SEND_K, REC_L, T1]
  (not(ALT), tail(P), L, 0)
)
[]
T1 !TIMEOUT;
(
[RN < max] ->
S_1[INPUT, SEND_K, REC_L, T1] (ALT, P, L, RN+1)
[]
[RN == max] ->
INPUT !conf(P);
udef
)
process R_1 [OUTPUT, REC_K, SEND, T2] (END, ALT: Bool): noexit :=
REC_K ?FST, LST: Bool !ALT ?D: Data;
T2 !RESET;
OUTPUT !D !ind (FST, LST);
T2 !START;
SEND_L;
R_1 [OUTPUT, REC_K, SEND_L, T2] (LST, not (ALT))
[]
REC_K ?FST, LST: Bool !not (ALT) ?D: Data;
(* duplicated chunk *)
SEND_L;
R_1 [OUTPUT, REC_K, SEND_L, T2] (LST, ALT)
[]
T2 !TIMEOUT;
( [not(END)] ->
(* more chunks to follow *)
OUTPUT !I_NCK;
undef
[]
[END] ->
(* last chunk already delivered *)
undef
)
endproc

We tested the system with the list length two and retransmission number one. Otherwise the system would be quite large and it would take a long time to construct the determinized version. The global state graph now consists of 383 states and 785 transitions. The determinized version consists of 12 states and 28 transitions. We tested the conditions

\[
\begin{align*}
\subseteq_{BK\text{or}f} & \quad SENDING_{-}CLIENT[INPUT] | [INPUT] BRP2[INPUT, OUTPUT] \\
\subseteq_{BK\text{or}f} & \quad SENDING_{-}CLIENT[INPUT] | [INPUT] BRP4[INPUT, OUTPUT] \\
\subseteq_{BK\text{or}f} & \quad SENDING_{-}CLIENT[INPUT] | [INPUT] SERVICE[INPUT, OUTPUT]
\end{align*}
\]

These are now valid and we can continue.

### 10.3.5 The fifth step

We can now try to complete the specification of the BRP-protocol. The processes S_1, R_1 and T2 become as follows:
process S_1 [INPUT, SEND_K, REC_L, T1]
  (ALT: Bool, P: Packet, L, RN: Nat) : noexit :=
  SEND_K !(len(P) == L) !(len(P) == 1) !ALT !head(P);
  T1 !START;
  (REC_L;
   T1 !RESET;
   (len(P) == 1) ->
     INPUT !I_OK;
     S [INPUT, SEND_K, REC_L, T1] (not (ALT)) []
   (len(P) > 1) ->
     S_1 [INPUT, SEND_K, REC_L, T1]
     (not(ALT), tail(P), L, 0)
  )
[]
T1 !TIMEOUT;
(RN < max) ->
  S_1[INPUT, SEND_K, REC_L, T1] (ALT, P, L, RN+1) []
(RN == max) ->
  INPUT !conf(P);
  S [INPUT, SEND_K, REC_L, T1]
)
endproc

process R_1 [OUTPUT, REC_K, SEND, T2] (END, ALT: Bool) : noexit :=
  REC_K ?FST, LST: Bool !ALT ?D: Data;
  T2 !RESET;
  OUTPUT !D !ind (FST, LST);
  T2 !START;
  SEND_L;
  R_1 [OUTPUT, REC_K, SEND_L, T2] (LST, not (ALT))
[]
  REC_K ?FST, LST: Bool !not (ALT) ?D: Data;
  (* duplicated chunk *)
  SEND_L;
  R_1 [OUTPUT, REC_K, SEND_L, T2] (LST, ALT)
[]
  T2 !TIMEOUT;
10.3. DESIGN OF BRP

( [not(END)] ->
  (* more chunks to follow *)
  OUTPUT !I_NOK;
  R [OUTPUT, REC_K, SEND_L, T2]
[]
[END] ->
  (* last chunk already delivered *)
  R [OUTPUT, REC_K, SEND_L, T2]
)
endproc

process T2 [T2] : noexit :=
  T2 !START;
  ( T2 !RESET;
    T2 [T2]
  []
    T2 !TIMEOUT;
    T2 [T2]
  )
endproc

The global state graph was now quite large even with the packet size 2 and retransmission number 1. It consisted of 6181 states and 11395 transitions. The graph contained deadlocks and that is why it cannot be equivalent with the service. One trace leading to a deadlock is

INPUT !CONS(DATA(1), CONS(DATA(2), NIL)) ; OUTPUT !DATA(1) !I_FST;
OUTPUT !I_NOK.

This trace is not contained in the service. In the service, the receiver cannot inform the client with I_NOK before the sender has informed its client. It can be concluded that the second timer is too simple.

In this protocol the timers have a delicate structure. Before accepting a new packet from the client, the sender must ensure that the receiver has also detected this situation in order to properly restart the whole protocol. Because Lotos contains no explicit time concepts, we model this causal constraint with the help of an auxiliary gate SYNC ([Mateescu 96]). The sender issues "sender ready" or S_READY signal on the SYNC gate, enabling the T2 timer to time out, waits for a "receiver ready" or R_READY reply on the same gate and then return to its initial state by calling the S process.
If the receiver receives a timeout signal from $T_2$, it means that the contact has been lost with the sender. If the current packet has not been completely delivered, an $I\_\text{NCK}$ indication is issued to the receiving client. An $R\_\text{READY}$ signal is sent to $T_2$ in response to the timeout, and the receiver returns to its initial state by calling the process $R$.

We have to complete $S_1$, $R_1$ and $T_2$:

```plaintext
process T2 [T2, SYNC] : noexit :=
    T2 !START;
    (T2 !RESET;
     T2 [T2, SYNC]
    )
    SYNC !S\_READY;
    T2 !TIMEOUT;
    T2 !R\_READY;
    SYNC !READY;
    T2 [T2, SYNC]
)
endproc

process S_1 [INPUT, SEND\_K, REC\_L, T1, SYNC]
    (ALT: Bool, P: Packet, L, RN: Nat) : noexit :=
    SEND\_K !(len(P) == L) !(len(P) == 1) !ALT !head(P);
    T1 !START;
    (REC\_L;
     T1 !RESET;
     (len(P) == 1) =>
      INPUT !I\_OK;
      S [INPUT, SEND\_K, REC\_L, T1] (not (ALT))
     []
     [len(P) > 1] =>
      S_1 [INPUT, SEND\_K, REC\_L, T1]
      (not(ALT), tail(P), L, 0)
    )
    []
    T1 !TIMEOUT;
    (RN < max) =>
      S_1[INPUT, SEND\_K, REC\_L, T1] (ALT, P, L, RN+1)
    []
    [RN == max] =>
```
10.3. DESIGN OF BRP

```
INPUT !conf(P);
SYNC !S_READY;
SYNC !R_READY;
    S [INPUT, SEND_K, REC_L, T1, SYNC] (not(ALT))
)
endproc
```

```
process R_1 [OUTPUT, REC_K, SEND, T2] (END, ALT: Bool) : noexit :=
    REC_K ?FST, LST: Bool !ALT ?D: Data;
    T2 !RESET;
    OUTPUT !D !ind (FST, LST);
    T2 !START;
    SEND_L;
    R_1 [OUTPUT, REC_K, SEND_L, T2] (LST, not (ALT))
[]
REC_K ?FST, LST: Bool !not (ALT) ?D:Data;
(* duplicated chunk *)
SEND_L;
R_1 [OUTPUT, REC_K, SEND_L, T2] (LST, ALT)
[]
T2 !TIMEOUT;

( [not(END)] ->
    (* more chunks to follow *)
    OUTPUT !I_NCK;
    T2 !R_READY;
    R [OUTPUT, REC_K, SEND_L, T2]
[]
[END] ->
    (* last chunk already delivered *)
    T2 !R_READY;
    R [OUTPUT, REC_K, SEND_L, T2]
)
endproc
```

Now we can show the relation

$\text{SENDING\_CLIENT[INPUT]} \parallel [\text{INPUT}] \triangleright \text{BRP5[INPUT, OUTPUT]} \sqsubseteq_{\text{BKOref}}$

$\text{SENDING\_CLIENT[INPUT]} \parallel [\text{INPUT}] \triangleright \text{SERVICE[INPUT, OUTPUT]}$

Because all the processes are now completely defined, the relation is in fact BKO-equivalence. We have thus completed the design of the Bounded Retransmission Protocol and shown that it functions according to the service
specification. We can now test the correctness of the design by varying the length of the data list and the retransmission number, see [Mateescu 96] for example.
10.4 Derivation of BRP using action refinement

We have developed Bounded Retransmission Protocol stepwise using state refinement. This enabled us to proceed gradually and at every step to verify the correctness of the construction. This approach has, however, one negative aspect. It was necessary to define nearly all the abstract data types beforehand and completely. It can be argued that one seldom knows at the beginning what kind of data types are necessary or useful. It would be better if data types could also be introduced step by step along with the state refinement.

We now try to develop BRP using action refinement as well. In this protocol the actions are not very long and the different parts in the action are not independent, but must be used together. That is why we use only one step in action refinement. First, we leave the whole inside structure in the action open, defining only the gate name, for example $g!*$. In a refinement step for the action we complete the action, for example $g!*2$. Even this kind of simple refinement makes it easier to proceed gradually, because it is more natural to define data types in steps when needed. Notice that there can be many state refinement steps before an action is completed.

Because BRP is now familiar we use larger steps than in the state refinement approach. For example, now we do not consider the difficult timing constraints in detail.

10.4.1 First step

At the beginning we need the data type Packet that the protocol uses when it communicates with the sending client. In other parts of the protocol we use only gates without data. In this way we first test the overall structure of the protocol before starting to develop data structures in detail. The overall structure of the specification is the same as before. So we have the processes SENDING_CLIENT, sender S, receiver R, channels K and L, timers T1 and T2. In the first step, the above processes are as follows:

- **SENDING_CLIENT** is completely defined and the same as before.
- process S[INPUT, SEND_K, REC_L, T1]: noexit :=
  INPUT ?P:Packet;
  (
    [len(P) == 0] -> S[INPUT, SEND_K, REC_L, T1]
    []
    [len(P) > 0] -> S_1[SEND_K, REC_L, T1]
  )
  endproc
- process S_1[SEND_K, REC_L, T1]: noexit :=
  T1 !*;
  SEND_K !*;
(  
  REC_L;  
  udef  
)
[]
  udef
endproc

• process R[OUTPUT, REC_K, SEND_L, T2]: noexit :=  
  REC_K !*;  
  OUTPUT !*;  
  T2 !*;  
  SEND_L;  
  R_1[REC_K]
endproc

• process R_1[REC_K]: noexit :=  
  REC_K !*;  
  udef
endproc

• process K[SEND_K, REC_K]: noexit :=  
  SEND_K !*;  
  (  
    i;  
    REC_K !*;  
    K[SEND_K, REC_K]  
  )  
  udef
endproc

• process L[SEND_L, REC_L]: noexit :=  
  SEND_L;  
  (  
    i;  
    REC_L;  
    L[SEND_L, REC_L]  
  )  
  udef
endproc

• process T1[T1]: noexit :=  
  T1 !*;  
  udef
endproc
10.4. DERIVATION OF BRP USING ACTION REFINEMENT

- process T2[T2]: noexit :=
  T2 !*;
  udef
endproc

The whole system is as before. If we denote it by $\text{BRP}_1[\text{INPUT, OUTPUT}]$, we can show with the help of our software that

$$\text{SENDING\_CLIENT}[\text{INPUT}] \parallel \text{[INPUT]} \parallel \text{BRP}_1[\text{INPUT, OUTPUT}] \equiv_{\text{BKoref}}$$

$$\text{SENDING\_CLIENT}[\text{INPUT}] \parallel \text{[INPUT]} \parallel \text{SERVICE}[\text{INPUT, OUTPUT}].$$

10.4.2 Second step

This is a large step. We complete the specifications of channels K and L and timer T1. Thus we must define the data type for START, RESET and TIMEOUT. We also define all the operations needed in sending the chunks. The data of the OUTPUT gate is still left open in order to test our software.

In this step, we leave open the synchronization of S, T1 and T2 as well as the synchronization of R and T2. The processes K, L and T1 are now completely defined and we do not repeat their code here. We refine the processes S_1, R_1 and T2 as follows:

- process S_1[INPUT, SEND_K, REC_L, T1]
  (ALT: Bool, P: Packet, L, RN: Nat): noexit :=
  T1 !START;
  SEND_K !(len(P) == L) !(len(P) == 1) !ALT !head(P);
  ( REC_L;
    T1 !RESET;
    ( [len(P) == 1] ->
      INPUT !I_OK;
      S[INPUT, SEND_K, REC_L, T1](not(ALT)) []
    [len(P) > 1] ->
      S_1[INPUT, SEND_K, REC_L, T1]
      (not(ALT), tail(P), L, 0)
    )
  )
  []
  T1 !TIMEOUT;
  ( [RN < max] -> S_1[INPUT, SEND_K, REC_L, T1]
    (ALT, P, L, RN+1)
  )
  [RN == max] ->
\begin{verbatim}
INPUT !conf(P);
   udef
)
)
endproc

• process R_1[OUTPUT, REC_K, SEND_L, T2](END, ALT: Bool): noexit :=
   REC_K ?FST, LST: Bool !ALT ?D: Data;
   T2 !RESET;
   OUTPUT !*;
   T2 !START;
   SEND_L;
   R_1[OUTPUT, REC_K, SEND_L, T2](LST, not(ALT))

   REC_K ?FST, LST: Bool !not(ALT) ?D: Data;
   SEND_L;
   R_1[OUTPUT, REC_K, SEND_L, T2](LST, ALT)

   T2 !TIMEOUT;
   ( [not(END)] ->
     OUTPUT !I_NCK;
     udef
   )
   [END] ->
   udef
)
endproc

• process T2[T2]: noexit :=
   T2 !START;
   ( T2 !RESET;
     T2[T2]
   )
   udef
   )
   udef
endproc
\end{verbatim}

The system has 614 states and 1291 transitions. The length of the data packets is one and no retransmissions are allowed. The system is in BKQ-preorder with the service. Thus we can complete the design in the next step.
10.4.3 Third step

We do not try various forms of the timers any longer, because those details were already considered in the state refinement example. Thus, we complete the designs of S, R and T2, but still leave the OUTPUT action partially defined in order to test our software. As expected, the Bounded Retransmission Protocol specification and the service specification are in BKO-refinement relation. Thus everything has succeeded well and in the final step we can define the output action completely.
Chapter 11

Conclusions and Further Research

The new results have been explained in the section 1.2. The most important conclusions are that failure semantics also allows partially defined specifications and it is difficult to define bisimulation relations so that they allow partially defined actions in full Lotos. Surprisingly, failure semantics, in our case BKO- and CFFD-semantics, allow restricted action refinement, although it does not behave in the best possible way. In addition, CFFD-semantics seems to correspond with the intuition better than bisimulation semantics.

The stepwise refinement of specifications using processes with partially defined states seems a valuable idea and it is worthwhile to apply in practice. I have applied the method with the design of Bounded Retransmission Protocol, which is a real protocol and fairly complicated. The method worked well with this protocol. I also tried to apply action refinement in the design, but it turned out that in this example the action refinement was not as important as the state refinement. In fact, it was not very obvious that the action refinement would be of any use in any of the other examples I had at my disposal. This does not necessarily mean that the action refinement is useless in practice. It means only that we need a different type of protocols on which to apply the action refinement method. Especially, we need examples which contain several data items in one action and possibly also multisynchronization. These kind of examples are now rare. The possibility to develop specifications by refining states and actions may alter the style in which specifications are written.

As an application of partially defined states, I have presented a tentative distributed algorithm to detect deadlocks. Its main aim was not to speed up the generation of global states, but to distribute the states into many central memories. It is suitable to be used with very large systems, with over a million states.

As for further research, one possibility is to restrict equivalences and refinement relations. As mentioned in [Cleveland and Steffen 90], one could equip every

149
partial state with a set of possible actions or, in the case of full Lotos, a set of gate names. When refining a state, only actions in the corresponding set can be used. It should not be hard to define equivalences and refinement relations according to these rules. This approach could also work in action refinement although there are many ways to define the possible actions.

Then there are different semantics. We have studied refinement in bisimulation and failure semantics. It is not very interesting to examine refinement on closely related semantics, but true concurrency semantics is different. There are many versions of true concurrency ([Langerak 94]), but not all are suitable for Lotos. For example, in the article [da Costa and Courtiat 92] a suitable true concurrency semantics is constructed for Lotos. It is well known that action refinement works better in true concurrency semantics ([Aceto 92], [Castellano et al 87]) and it would be very interesting to examine action refinement in full Lotos in true concurrency semantics. This has already been done in [Courtiat and Saidouni 93], but their approach is different compared to our method.

Lastly, implementations are a never ending task. I have implemented BKORelations with partial states and actions and CFFD-relations with partial states. The algorithms work well in practice, although they seem a bit time-consuming in the worst case. It would not require a great effort to implement the bisimulation relations with partial states. This has already been done with respect to CCS ([Celikkan and Cleaveland 93]). It is an interesting research topic to study, if the bisimulation refinement algorithm can be applied to check the BKO and CFFD-refinements as has been done in the case of the testing preorders in [Celikkan and Cleaveland 92] and [Cleaveland and Hennessy 93]. Distributed algorithms are also possible. The realization of distributed detection of deadlocks needs a lot of distributed programming, if one is going to use the method in practice.

Thus it seems that the stepwise refinement of specifications, based on undefined processes, offers a wide area for further work. This refinement method is concrete, intuitive and easy to use. In many other refinement techniques, refinement points are not marked explicitly and it is not particularly easy for a large group to work on such specifications. In our method, those points, both states and actions, that need refinement are clearly marked. It can concluded that further work to develop the method is well-motivated.
Bibliography


[Cleaveland and Steffen 93] R. Cleaveland and B. Steffen, A linear-time model-checking algorithm for the


[Kaivola and Valmari 92] R. Kaivola and A. Valmari, The weakest compositional semantic equivalence preserving nexttime-less linear temporal logic,


[Tienari 89] M. Tienari, Johdanto tilakoneformal- 
ismin käyttöön tietoliikenneprotokollien 
spesificoinnissa ja analyysissä. Department of 
Computer Science, University of 


[Valmari & al 93] A. Valmari, J. Kemppainen, M. Clegg and 
M. Levanto, Putting Advanced Reacha-
bility Analysis Techniques Together: the 
'ARA' Tool. Proceedings of FME'93: 
Industrial-Strength Formal Methods, First 
International Symposium of Formal Meth-
ods Europe, Odense, Denmark, April 1993, 
Proceedings, pp 597-616.

[Valmari and Tienari 91] A. Valmari and M. Tienari, An Improved 
Failures Equivalence for Finite-State Sys-
tems with a Reduction Algorithm. 11th 
International IFIP WG6.1 Symposium on 
Protocol Specification, Testing and Verifi-

[Valmari and Tienari 95] A. Valmari and M.Tienari, Compositional 
Failure-Based Semantic Models for Basic 
LOTOS. Formal Aspects of Computing 7, 

Information and Computation, 85:202-