Basic Research in Computer Science

Checking for Open Bisimilarity in the π-Calculus

Ulrik Frendrup
Jesper Nyholm Jensen

BRICS Report Series **RS-01-8**

ISSN 0909-0878 **February 2001**

**See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:**

**BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk**

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/01/8/`

Checking for Open Bisimilarity in the π-Calculus

BRICS[1]
Ulrik Frendrup & Jesper Nyholm Jensen
Aalborg University, Department of Computer Science
Fredrik Bajers Vej 7E
9220 Aalborg Ø, Denmark

Email: ulrikf@cs.auc.dk, nyholm@gaztricjuice.dk

Abstract

This paper deals with algorithmic checking of open bisimilarity in the π-calculus. Most bisimulation checking algorithms are based on the partition refinement approach. Unfortunately the definition of open bisimulation does not permit us to use a partition refinement approach for open bisimulation checking directly, but in the paper *A Partition Refinement Algorithm for the π-Calculus* Marco Pistore and Davide Sangiogi present an iterative method that makes it possible to check for open bisimilarity using partition refinement. We have implemented the algorithm presented by Marco Pistore and Davide Sangiorgi. Furthermore, we have optimized this algorithm and implemented this optimized algorithm. The time-complexity of this algorithm is the same as the time-complexity for the first algorithm, but performance tests have shown that in many cases the running time of the optimized algorithm is shorter than the running time of the first algorithm.

Our implementation of the optimized open bisimulation checker algorithm and a user interface have been integrated in a system called the OBC Workbench. The source code and a manual for it is available from http://www.cs.auc.dk/research/FS/ny/PR-pi/.

[1]Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation

Contents

Chapter 1

Introduction

This paper deals with algorithmic checking of open bisimilarity in the π-calculus. We have implemented an algorithm for open bisimulation checking in the π-calculus. Most algorithms for bisimulation checking are based on partition refinement, but the definition of open bisimulation makes it difficult to use this strategy for an open bisimulation checking algorithm directly since the state spaces of the processes being checked for open bisimilarity cannot be constructed separately. We have implemented the open bisimulation checking algorithm presented in [10]. This algorithm was developed by introducing the notion of constrained processes, defining active names bisimulation on constrained processes, showing that there is a useful connection between open bisimilarity and active names bisimilarity, and developing an iterative method for checking active names bisimilarity.

We have optimized the algorithm for open bisimulation checking, implemented this algorithm, and carried out some performance tests of the first and the optimized version of the implementation. Finally, the implementation of the optimized algorithm and a user interface have been integrated in a system called the OBC Workbench.

This paper contains seven chapters and two appendices and is organized as follows. Chapter 2 describes the generalized partition refinement problem and shows how an algorithm for solving this problem can be used for checking bisimilarity. In chapter 3 we present the syntax and semantics of a subset of the π-calculus and define some traditional bisimilarities between π-processes. In chapter 4 we present another notion of bisimilarity called open bisimilarity defined by [10]. Chapter 4 also contains some results from [10] that shows that open bisimilarity can be checked algorithmically. Chapter 5 contains a description of our first implementation of the open bisimulation checking algorithm and chapter 6 describes how the algorithm is optimized with respect to running time. We conclude on the implementation in chapter 7. Pseudo code for the optimized implementation can be found in appendix A. Results of performance tests of the first and the optimized implementation and a comparison of these can be found in appendix B.

Chapter 2

Checking for Bisimilarity Using Partition Refinement

The algorithm for open bisimulation checking in the π-calculus developed by [10] is as many other bisimulation equivalence checking algorithms based on partition refinement. In this chapter we describe the generalized problem of partition refinement. Furthermore, we show how this can be used for bisimulation checking.

2.1 The Generalized Partition Refinement Problem

Before we can present the ***generalized partition refinement problem*** we need to define the notions of ***partitions*** and ***blocks***.

Definition 1 (Partitions and blocks)
Let U be a set. A partition $\mathcal{W}$ of U is a finite set of pairwise disjoint subsets $B_1, \ldots B_n$ of U where $\bigcup_{i \in \{1,\ldots,n\}} B_i = U$. The sets B_i of $\mathcal{W}$ are called blocks. ■

Definition 2 (Generalized Partition Refinement Problem)
Given a set U, a partition $\mathcal{W} = \{B_1, \ldots B_n\}$ of U, and k functions $f_l : U \to \mathcal{P}(U)$, $1 \leq l \leq k$, we want to refine $\mathcal{W}$ into a new partition $\mathcal{W}'$ of U, where $\mathcal{W}' = \{C_1, \ldots, C_m\}$ is the coarsest set (the set with fewest blocks) satisfying

(i) for each $i \in \{1, \ldots, m\}$ there exists a $j \in \{1, \ldots, n\}$ such that $C_i \subseteq B_j$ and

(ii) for any $a, b \in C_i$, any block C_j, $1 \leq i, j \leq m$, and any function f_l, $1 \leq l \leq k$ it holds that $f_l(a) \cap C_j \neq \emptyset$ if and only if $f_l(b) \cap C_j \neq \emptyset$.

■

The new partition $\mathcal{W}'$ always exists[3] and is the unique greatest fixed point of the function $\psi_{\mathcal{W}}$ that maps partitions of U to partitions of U and is defined by

$C \in \psi_{\mathcal{W}}(\mathcal{W}')$ if and only if

(i) $\exists B \in \mathcal{W}.(C \subseteq B)$ and

(ii) for any $a, b \in C$, any block $C' \in \mathcal{W}'$, and any function f_l, $1 \leq l \leq k$ it holds that $f_l(a) \cap C' \neq \emptyset$ if and only if $f_l(b) \cap C' \neq \emptyset$.

The partition $\mathcal{W}'$ induces an equivalence $\mathcal{W}'_{\sim}$ defined by $\mathcal{W}'_{\sim} \stackrel{def}{=} \{(a,b) \mid \exists B \in \mathcal{W}.(a, b \in B)\}$. So the generalized partition refinement problem consists of computing the equivalence classes of U for an equivalence defined as a greatest fixed point.

In section 5.3 we will present an algorithm to solve the partition refinement problem for a finite set U and $k = 1$. This can be used to solve the generalized partition refinement problem in an iterative process where the refinement algorithm is applied once for each function in each iteration until the partition stabilizes.

In the following section we show how a partition refinement algorithm can be used for bisimulation checking.

2.2 Bisimulation Checking

Let $\mathcal{P}$ be a set of states and $\mathcal{A}ct$ a set of labels. We let P and Q range over $\mathcal{P}$ and α over $\mathcal{A}ct$. For a given labeled transition system ($\mathcal{P}$, $\mathcal{A}ct$, $\longrightarrow$) we define bisimulation as follows.

Definition 3 (Bisimulation)
A relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a bisimulation if $(P, Q) \in \mathcal{R}$ implies,

(i) if $P \stackrel{\alpha}{\longrightarrow} P'$ then for some $Q' \in \mathcal{P}$, $Q \stackrel{\alpha}{\longrightarrow} Q'$ and $(P', Q') \in \mathcal{R}$, and

(ii) if $Q \stackrel{\alpha}{\longrightarrow} Q'$ then for some $P' \in \mathcal{P}$, $P \stackrel{\alpha}{\longrightarrow} P'$ and $(P', Q') \in \mathcal{R}$.

■

We will let $\sim$ denote the greatest bisimulation and say that P and Q are equivalent or bisimilar if $P \sim Q$. We will use the shorthand notation $P \stackrel{s}{\longrightarrow} P'$ if $s \stackrel{def}{=} a_1 a_2 \cdots a_n$, $a_i \in \mathcal{A}ct$ and there exists a sequence of states $P_1, P_2, \ldots, P_n \in \mathcal{P}$ such that $P \stackrel{a_1}{\longrightarrow} P_1 \stackrel{a_2}{\longrightarrow} \cdots \stackrel{a_n}{\longrightarrow} P_n$ and $P' = P_n$. Furthermore, we define the labels contained in a state as $\mathrm{n}(P) = \{\alpha \in \mathcal{A}ct \mid \exists s_1, s_2 \in \mathcal{A}ct^*.(\exists P' \in \mathcal{P}.P \stackrel{s_1 \alpha s_2}{\longrightarrow} P')\}$.

Now we show how a partition refinement algorithm can be used for bisimulation checking. If refine$(U, \mathcal{W}, f_1, \ldots, f_k)$ is a function which returns a solution of the generalized partition refinement problem for the instance $U, \mathcal{W}, f_1, \ldots, f_k$ then it can be used for checking for

bisimilarity between the states P and Q as follows. Let S_P denote the state space of P, i.e. $S_P = \{P' \mid \exists s \in \mathcal{A}ct^*.P \xrightarrow{s} P'\}$. Let U be the set $S_P \cup S_Q$ and $\mathcal{W}$ the partition of U containing only one block which is the set U. Furthermore, let the functions $f_\alpha : U \to \mathcal{P}(U)$, $\alpha \in \mathrm{n}(P) \cup \mathrm{n}(Q)$ be defined as $f_\alpha(R) = \{R' \in U \mid R \xrightarrow{\alpha} R'\}$. P and Q are bisimilar if and only if they are in the same block of the refined partition $\mathcal{W}'$ returned from refine$(U, \mathcal{W}, \{f_\alpha\}_{(\alpha \in \mathrm{n}(P) \cup \mathrm{n}(Q))})$. This is true since $\sim$ gives rise to a partition $\mathcal{W}''$ of U where the blocks are the equivalence classes of $\sim \cap (U \times U)$. $\mathcal{W}''$ is clearly a fixed point of $\psi_{\mathcal{W}}$. If $\mathcal{W}''$ is not the same as $\mathcal{W}'$, i.e. the greatest fixed point of $\psi_{\mathcal{W}}$, there exists a bisimulation $\sim \cup \mathcal{W}'_\sim$ larger than $\sim$, which is a contradiction.

Chapter 3

The π-Calculus

In this chapter we present the notion of late and early bisimulation for a subset of the π-calculus and describe some of the problems associated with algorithmic checking for late and early bisimilarity. First, we give the syntax and semantics of a subset of the π-calculus.

3.1 Syntax

We begin by giving the syntax for a subset of the π-calculus. The syntactic categories for the π-calculus are: an infinite set of ***names***, $\mathcal{N}$, a set of ***actions***, $\mathcal{A}ct$, an infinite set of ***process identifiers***, $\mathcal{K}$, and a set of π-***processes***, $\mathcal{P}r$. We will denote elements of $\mathcal{N}$ by a, b, c, d, e, v, x, and y, elements of $\mathcal{A}ct$ by α and β, elements of $\mathcal{K}$ by K, and elements of $\mathcal{P}r$ by P and Q. The set of processes can be constructed with the constructors for ***inaction***, ***prefix***, ***matching***, ***nondeterministic choice***, ***parallel composition***, ***restriction***, and ***recursion***, and an action can be a ***silent action***, ***input***, ***free output***, or ***bound output***. The grammar for $\mathcal{A}ct$ and $\mathcal{P}r$ is presented below.

$$\alpha ::= \tau \mid a(b) \mid \bar{a}b \mid \bar{a}(b)$$
$$P ::= \mathbf{0} \mid \alpha.P \mid [a = b]P \mid P + P \mid P|P \mid (\nu\ a)P \mid K\langle\tilde{a}\rangle$$

Each process identifier K has an associated arity and a definition of the form $K \stackrel{def}{=} (\tilde{b})P$, where $\tilde{b}$ are the free names of P (see definition 4). For the process $K\langle\tilde{a}\rangle$ the tuple $\tilde{a}$ must have the same length as $\tilde{b}$. The ***free***, ***bound***, ***extruded***, ***object***, and ***subject*** names of an action α, written $\mathrm{fn}(\alpha)$, $\mathrm{bn}(\alpha)$, $\mathrm{en}(\alpha)$, $\mathrm{ob}(\alpha)$, and $\mathrm{sub}(\alpha)$, respectively, are defined as follows.

α	$\mathrm{fn}(\alpha)$	$\mathrm{bn}(\alpha)$	$\mathrm{en}(\alpha)$	$\mathrm{ob}(\alpha)$	$\mathrm{sub}(\alpha)$
τ	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$a(b)$	$\{a\}$	$\{b\}$	$\emptyset$	$\{a\}$	$\{b\}$
$\bar{a}b$	$\{a, b\}$	$\emptyset$	$\emptyset$	$\{a\}$	$\{b\}$
$\bar{a}(b)$	$\{a\}$	$\{b\}$	$\{b\}$	$\{a\}$	$\{b\}$

The ***names of an action*** α are the free names and the bound names of α, i.e. $\mathrm{n}(\alpha) = \mathrm{fn}(\alpha) \cup \mathrm{bn}(\alpha)$. The free and bound names of a process are defined as follows.

Definition 4 (Free Names)
A name a is free in the process P, written $a \in \mathrm{fn}(P)$, if it belongs to the set $\mathrm{free}(P)$, where the function free: $\mathcal{P}r \to \mathcal{P}(\mathcal{N})$ is defined by the following.

$$\begin{aligned}
&\mathrm{free}(\mathbf{0}) = \emptyset \\
&\mathrm{free}(\alpha.P) = \mathrm{fn}(\alpha) \cup (\mathrm{free}(P) \setminus \mathrm{bn}(\alpha)) \\
&\mathrm{free}([a = b]P) = \{a, b\} \cup \mathrm{free}(P) \\
&\mathrm{free}(P_1|P_2) = \mathrm{free}(P_1) \cup \mathrm{free}(P_2) \\
&\mathrm{free}(P_1 + P_2) = \mathrm{free}(P_1) \cup \mathrm{free}(P_2) \\
&\mathrm{free}((\nu\ a)P) = \mathrm{free}(P) \setminus \{a\} \\
&\mathrm{free}(K\langle(a_1, a_2, \ldots, a_n)\rangle) = \{a_1, a_2, \ldots, a_n\}
\end{aligned}$$

■

Definition 5 (Bound Names)
A name a is bound in the process P, written $a \in \mathrm{bn}(P)$, if it belongs to the set $\mathrm{bound}(P)$, where the function bound: $\mathcal{P}r \to \mathcal{P}(\mathcal{N})$ is the least function that satisfies the following.

$$\begin{aligned}
&\mathrm{bound}(\mathbf{0}) = \emptyset \\
&\mathrm{bound}(\alpha.P) = \mathrm{bn}(\alpha) \cup \mathrm{bound}(P) \\
&\mathrm{bound}([a = b]P) = \mathrm{bound}(P) \\
&\mathrm{bound}(P_1|P_2) = \mathrm{bound}(P_1) \cup \mathrm{bound}(P_2) \\
&\mathrm{bound}(P_1 + P_2) = \mathrm{bound}(P_1) \cup \mathrm{bound}(P_2) \\
&\mathrm{bound}((\nu\ a)P) = \{a\} \cup \mathrm{bound}(P) \\
&\mathrm{bound}(K\langle(\tilde{a})\rangle) = \mathrm{bound}(P) \text{ where } K \stackrel{def}{=} (\tilde{b})P
\end{aligned}$$

■

The ***names of a process*** P, written $\mathrm{n}(P)$, consists of the free names and the bound names of P, i.e. $\mathrm{n}(P) = \mathrm{fn}(P) \cup \mathrm{bn}(P)$. We sometimes write $\mathrm{fn}(P_1, P_2, \ldots, P_n)$ for $\mathrm{fn}(P_1) \cup \mathrm{fn}(P_2) \cup \cdots \cup \mathrm{fn}(P_n)$, and similarly for bound names and names.

We identify processes up to renaming of bound names. Renaming of a bound name in a process is called α-***conversion***. If the processes P and Q can be identified up to renaming of bound names then P and Q are α-***convertible***, written $P \equiv_\alpha Q$.

Definition 6 (α-conversion)
α-conversion of a name a to a name b in a π-process P, where $b \notin \mathrm{n}(P)$, is the result of renaming all bound occurrences of a in P to b. ■

Substitutions are denoted by σ and are functions that map names to names, e.g. $\sigma \stackrel{def}{=} \{y_1/x_1\ y_2/x_2\ \cdots\ y_n/x_n\}$ denotes the simultaneous substitution that maps every free occurrence of the name x_i to the name y_i for all $i \in \{1, 2, \ldots, n\}$. A name a is ***neutral*** for a substitution σ if for all $b \in \mathcal{N}$, $\sigma(b) = a$ if and only if $b = a$. A set of names, V, is neutral for a substitution σ if all the names of V are neutral for σ. Application of a substitution σ to a process P is written $P\sigma$. If there exists some bound name a in P such that the substitution σ maps some name to a then a is α-converted to some b that is neutral for σ.

3.2 Semantics

The operational semantics for the subset of the π-calculus is given by the labeled transition system $(\mathcal{P}r, \mathcal{A}ct, \rightarrow)$ where $\rightarrow$ is the smallest relation closed under the rules in table 3.1. The symmetric versions of the rules Sum, Par, Com, and Close have been omitted. Transitions have the form $P \xrightarrow{\alpha} P'$.

[Alpha]	$\dfrac{P' \xrightarrow{\alpha} P''}{P \xrightarrow{\alpha} P''}$	$P \equiv_\alpha P'$	[Pre]	$\dfrac{}{\alpha.P \xrightarrow{\alpha} P}$
[Con]	$\dfrac{P\{\tilde{a}/\tilde{b}\} \xrightarrow{\alpha} P'}{K\langle\tilde{a}\rangle \xrightarrow{\alpha} P'}$	$K \stackrel{def}{=} (\tilde{b})P$	[Sum]	$\dfrac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$
[Par]	$\dfrac{P \xrightarrow{\alpha} P'}{P\|Q \xrightarrow{\alpha} P'\|Q}$	$\mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \emptyset$	[Com]	$\dfrac{P \xrightarrow{\bar{a}y} P' \quad Q \xrightarrow{a(x)} Q'}{P\|Q \xrightarrow{\tau} P'\|Q'\{y/x\}}$
[Close]	$\dfrac{P \xrightarrow{\bar{a}(x)} P' \quad Q \xrightarrow{a(x)} Q'}{P\|Q \xrightarrow{\tau} (\nu\ x)(P'\|Q')}$		[Match]	$\dfrac{P \xrightarrow{\alpha} P'}{[a = a]P \xrightarrow{\alpha} P'}$
[Res]	$\dfrac{P \xrightarrow{\alpha} P'}{(\nu\ b)P \xrightarrow{\alpha} (\nu\ b)P'}$	$b \notin \mathrm{n}(\alpha)$	[Open]	$\dfrac{P \xrightarrow{\bar{a}b} P'}{(\nu\ b)P \xrightarrow{\bar{a}(b)} P'}$ $b \neq a$

Table 3.1: Operational semantics for the π-calculus.

3.3 Bisimulation

In this section we define the notion of ***late bisimulation*** and ***early bisimulation*** first introduced by [5].

Definition 7 (Late Bisimulation)
A symmetric binary relation $\mathcal{R}$ on processes is a late bisimulation if $P\ \mathcal{R}\ Q$ and $P \xrightarrow{\alpha} P'$ where $\mathrm{bn}(\alpha) \cap (\mathrm{fn}(P) \cup \mathrm{fn}(Q)) = \emptyset$ implies

(i) if $\alpha = a(x)$ then $\exists Q'.(Q \xrightarrow{a(x)} Q' \wedge \forall y.(P'\{y/x\}\ \mathcal{R}\ Q'\{y/x\}))$, and

(ii) if $\alpha \neq a(x)$ then $\exists Q'.(Q \xrightarrow{\alpha} Q' \wedge P'\ \mathcal{R}\ Q')$.

■

From the definition of late bisimulation the notion of ***late bisimilarity*** is defined.

Definition 8 (Late Bisimilarity)
The processes P and Q are late bisimilar, written $P \dot{\sim} Q$, if there exists a late bisimulation $\mathcal{R}$ such that $P\ \mathcal{R}\ Q$. ■

As an example of two late bisimilar processes consider $a(x).\mathbf{0} \mid \bar{b}y.\mathbf{0}$ and $a(x).\bar{b}y.\mathbf{0} + \bar{b}y.a(x).\mathbf{0}$. It is easily seen that $a(x).\mathbf{0} \mid \bar{b}y.\mathbf{0} \dot{\sim} a(x).\bar{b}y.\mathbf{0} + \bar{b}y.a(x).\mathbf{0}$.

It turns out that with the operational semantics defined it is hard to give an efficient algorithm for checking late bisimilarity between processes. Suppose we need to check whether the processes P and Q are late bisimilar. If P has the transition $P \xrightarrow{a(x)} P'$, that is P can receive something on the channel a and become P', we can only explore transitions from P' by examining all the possible substitutions of x in P' and there are infinitely many of these.

Similar problems arise for ***early bisimilarity*** based on the late semantics given by the labeled transition system of table 3.1. ***Early bisimulation*** is defined as follows.

Definition 9 (Early Bisimulation)
A symmetric binary relation $\mathcal{R}$ on processes is an early bisimulation if $P\ \mathcal{R}\ Q$ and $P \xrightarrow{\alpha} P'$ where $\mathrm{bn}(\alpha) \cap (\mathrm{fn}(P) \cup \mathrm{fn}(Q)) = \emptyset$ implies

(i) if $\alpha = a(x)$ then $\forall y.(\exists Q'.(Q \xrightarrow{a(x)} Q' \wedge P'\{y/x\}\ \mathcal{R}\ Q'\{y/x\}))$, and

(ii) if $\alpha \neq a(x)$ then $\exists Q'.(Q \xrightarrow{\alpha} Q' \wedge P'\ \mathcal{R}\ Q')$.

■

From the definition of early bisimulation the notion of early bisimilarity is defined.

Definition 10 (Early Bisimilarity)
The processes P and Q are early bisimilar, written $P \dot{\sim}_E Q$, if there exists an early bisimulation $\mathcal{R}$ such that $P\ \mathcal{R}\ Q$. ■

Neither late nor early bisimulation are congruences since they are not closed under the constructor input prefix.

In the next section an alternative operational semantics and the definition of open bisimilarity is given. These make it possible to develop a more efficient bisimulation checking algorithm. The main reasons for using the definition of open bisimilarity in our implementation of a bisimulation checker is that open bisimilarity is a congruence and that a bisimulation checking algorithm for late or early bisimulation can be extracted from the open bisimulation checking algorithm [10]. Furthermore, the definition of open bisimilarity was also used in the Mobility Workbench and we hope [7] may benefit from our implementation.

Chapter 4

Algorithm for Checking Open Bisimilarity in the π-Calculus

In this chapter we present the notion of open bisimulation for the π-calculus and describe some of the problems associated with algorithmic checking for open bisimilarity. Finally, we present some results from [10] that make algorithmic checking for open bisimilarity possible. First, we present some basic notions needed for the semantics specialized for open bisimulation and the definition of open bisimulation. All lemmas, theorems, and corollaries presented in this chapter are proven in [10].

4.1 Syntax

We assume the same syntax as in the previous chapter except for the fact that we impose a strict ordering, $<$, on the set of names, $\mathcal{N}$, and that there exists a smallest name with respect to $<$. Furthermore, we introduce the notions of conditions and distinctions.

We let $\mathcal{M}$ denote the set of all ***conditions***. Conditions are ranged over by L, M, and N and are finite conjunctions of matching, e.g. $[a = b][c = d]$ is the condition that equates a with b and c with d. The ***names of a condition*** M, written $\mathrm{n}(M)$, are the names that appear in M. Composition of two conditions M and N is written MN. If every match of a condition, N, is implied by another condition, M, we write $M \rhd N$. If it is also the case that not every match of the condition M is implied by N we write $M \not\lhd\rhd N$. The trivially true condition is denoted by $\emptyset$. The substitution σ_M induced by a condition M maps each element of each equivalence class of M to the smallest element of that equivalence class, i.e. $\sigma_M(a) = \min\{b \mid M \rhd [a = b]\}$. To facilitate the evaluation of the expression $M \rhd N$, for some conditions M and N, conditions are transformed to a ***canonical form***.

Definition 11 (Canonical Form)
The canonical form of a condition M is the condition $M_c = [a_1 = b_1][a_2 = b_2] \cdots [a_n = b_n]$ where

(i) for all $a \in \mathrm{n}(M)$ with $a \neq \sigma_M(a)$ it holds that $[a = \sigma_M(a)] \in M_c$,

(ii) $a_i \neq b_i$ and either $b_i < b_{i+1}$ or $b_i = b_{i+1}$ and $a_i < a_{i+1}$, and

(iii) $M \triangleright M_c$.

■

The canonical form is defined in such a way that if M and N are conditions such that $M \lhd\rhd N$ then M_c is syntactically equal to N_c, where M_c and N_c are the canonical forms of M and N.

We let $\mathcal{D}is$ denote the set of all ***distinctions***. Distinctions are finite binary symmetric irreflexive relations on names $\mathcal{N}$ and are ranged over by D and E. If $(a, b) \in D$ for some distinction D then the two names a and b must be distinct, i.e. $a \neq b$. The names of D, written $\mathrm{n}(D)$, are the names that appear in D. A substitution σ ***respects*** D if $\sigma(a) \neq \sigma(b)$ for all $(a, b) \in D$. Likewise, a condition M respects the distinction D if the substitution, σ_M, induced by M respects D. Let $F \subseteq \mathcal{N}$ then $D - F$ denotes the distinction $\{(a, b) \in D \mid a, b \notin F\}$ and $D \cap F$ denotes the distinction $\{(a, b) \in D \mid a, b \in F\}$. In the definition of a distinction we will not always give all symmetric pairs, e.g. in $D \stackrel{def}{=} \{(a, b)\}$ the pair (b, a) has been left out.

4.2 Semantics

In this section we give a symbolic semantics specialized for open bisimulation for the subset of the π-calculus. A symbolic semantics is used to avoid the infinite branching that could occur with a traditional semantics when exploring the transitions of processes containing input prefixes. To see why an infinite branching can occur consider the process $a(x).P$ with the transition $a(x).P \xrightarrow{a(x)} P$. To explore the further transitions of P we have to explore the behavior of $P\{y/x\}$ for infinitely many ys. The notion of symbolic semantics for the π-calculus was first introduced by [1]. The operational semantics specialized for open bisimulation is given by the symbolic labeled transition system $(\mathcal{P}r, \mathcal{M} \times \mathcal{A}ct, \rightsquigarrow)$, where $\rightsquigarrow$ is the smallest relation closed under the rules in table 4.1. The symmetric versions of the rules Sum, Par, Com, and Close have been omitted. Transitions have the form $P \overset{(M,\alpha)}{\rightsquigarrow} P'$, where M represents the minimal condition required for P to perform the action α. We let μ range over $\mathcal{M} \times \mathcal{A}ct$.

[Alpha] $\dfrac{P' \overset{\mu}{\rightsquigarrow} P''}{P \overset{\mu}{\rightsquigarrow} P''} \quad P \equiv_\alpha P'$

[Pre] $\dfrac{}{\alpha.P \overset{(\emptyset,\alpha)}{\rightsquigarrow} P}$ [Con] $\dfrac{P\{\tilde{a}/\tilde{b}\} \overset{\mu}{\rightsquigarrow} P'}{K\langle\tilde{a}\rangle \overset{\mu}{\rightsquigarrow} P'} \quad K \overset{def}{=} (\tilde{b})P$

[Sum] $\dfrac{P \overset{\mu}{\rightsquigarrow} P'}{P+Q \overset{\mu}{\rightsquigarrow} P'}$ [Par] $\dfrac{P \overset{\mu}{\rightsquigarrow} P'}{P|Q \overset{\mu}{\rightsquigarrow} P'|Q} \quad \mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \emptyset$

[Com] $\dfrac{P \overset{(M,\bar{a}y)}{\rightsquigarrow} P' \quad Q \overset{(N,b(x))}{\rightsquigarrow} Q'}{P|Q \overset{(L,\tau)}{\rightsquigarrow} P'|Q'\{y/x\}}$ where $L \overset{def}{=} MN[a=b]$

[Close] $\dfrac{P \overset{(M,\bar{a}(x))}{\rightsquigarrow} P' \quad Q \overset{(N,b(x))}{\rightsquigarrow} Q'}{P|Q \overset{(L,\tau)}{\rightsquigarrow} (\nu\ x)(P'|Q')}$ where $L \overset{def}{=} MN[a=b]$

[Match] $\dfrac{P \overset{(M,\alpha)}{\rightsquigarrow} P'}{[a=b]P \overset{(N,\alpha)}{\rightsquigarrow} P'}$ where $N \overset{def}{=} M[a=b]$

[Res] $\dfrac{P \overset{\mu}{\rightsquigarrow} P'}{(\nu\ b)P \overset{\mu}{\rightsquigarrow} (\nu\ b)P'} \quad b \notin \mathrm{n}(\mu)$

[Open] $\dfrac{P \overset{(M,\bar{a}b)}{\rightsquigarrow} P'}{(\nu\ b)P \overset{(M,\bar{a}(b))}{\rightsquigarrow} P'} \quad b \notin \mathrm{n}(M) \cup \{a\}$

Table 4.1: The specialized operational semantics for the π-calculus.

From the transition system given by the rules in table 4.1 we define another transition system, $(\mathcal{P}r, \mathcal{M} \times \mathcal{A}ct, \longrightarrow)$, such that the condition M of a transition $P \overset{(M,\alpha)}{\longrightarrow} P'$ is in canonical form and α and P' are closed under the substitution σ_M, i.e. $\alpha = \alpha\sigma_M$ and

$P' = P'\sigma_M$. The relation $\longrightarrow$ is the smallest relation closed under the following rule.

$$[\text{Canon}] \quad \frac{P \overset{(N,\alpha)}{\rightsquigarrow} P'}{P \overset{(M,\alpha\sigma_M)}{\longrightarrow} P'\sigma_M} \qquad \begin{array}{l} \text{bn}(\alpha) \cap \text{fn}(P) = \emptyset \text{ and} \\ M \text{ is the canonical form of } N \end{array}$$

4.3 Open Bisimulation

In this section we give the definition of ***open bisimulation***. First we need to define the notion of ***distinction-indexed relation***.

Definition 12 (Distinction-Indexed Relation)
A distinction-indexed relation $\mathcal{R}$ is a set $\{\mathcal{R}_D\}_D$ of relations $\mathcal{R}_D$ over π-processes, where D ranges over all distinctions in $\mathcal{D}is$. ■

Definition 13 (Open Bisimulation)
A distinction-indexed relation $\mathcal{R}$ is an open bisimulation if $P\ \mathcal{R}_D\ Q$ implies

(i) for all M, α, and P' such that $P \overset{(M,\alpha)}{\longrightarrow} P'$, with $\text{bn}(\alpha) \cap \text{fn}(Q, D) = \emptyset$ and M respects D, there exist some N, β, and Q' such that $Q \overset{(N,\beta)}{\longrightarrow} Q'$ and

- $M \rhd N$,
- $\alpha = \beta\sigma_M$, and
- $P'\ \mathcal{R}_{D'}\ Q'\sigma_M$ for $D' \overset{def}{=} (D \cup (\text{en}(\alpha) \times \text{fn}(P, Q)))\sigma_M$, and

(ii) the converse, with the role of P and Q exchanged.

■

From the definition of open bisimulation we define the notion of ***open bisimilarity***.

Definition 14 (Open bisimilarity)
The processes P and Q are open bisimilar with respect to distinction D, written $P \sim_D Q$, if $P\ \mathcal{R}_D\ Q$ for some open bisimulation $\mathcal{R}$. ■

As it is seen in definition 13 the distinction D' is D updated with the fact that the extruded names of the action α must be different from all free names in the processes P and Q and afterwards updated by applying σ_M to it. Since this kind of updating of distinctions will often be used in the following, we will use a shorter notation defined as follows.

Definition 15 (Distinction Update)
Let D be a distinction, $P_1, \ldots, P_n$ processes, M a condition, and α an action. Then we define $D^M_{(P_1,\ldots,P_n),\alpha}$ as

$$D^M_{(P_1,\ldots,P_n),\alpha} \stackrel{def}{=} (D \cup (\mathrm{en}(\alpha) \times \mathrm{fn}(P_1, \ldots, P_n)))\sigma_M$$

■

With this definition D' in definition 13 could be written as $D^M_{(P,Q),\alpha}$.

To use the method described in chapter 2 to check whether $P \sim_D Q$ it is necessary to generate the state spaces of P and Q separately. This cannot be done since the following dependencies between P and Q affect the transitions and the derivatives.

Dependency 1 The name emitted by P may not occur free in Q.

Dependency 2 The substitution σ_M determined by the condition M in a transition of P must be applied to the derivative of Q.

Dependency 3 There is a global distinction, which is updated using the free names from both processes.

To see a discussion of the problems the different dependencies introduce we refer to the introduction of [10].

In the following three sections alternative characterizations of $\sim$, proposed by [10], are given, which avoid the mentioned dependencies and make it possible to use a partition refinement strategy to check for open bisimilarity.

4.4 Constrained Processes

To make the indexing distinctions of open bisimulation local to processes, and thereby avoiding dependency 3 described in the previous section, [10] introduced the notion of ***constrained processes***, defined a bisimilarity on these, and showed that there is a useful connection between constrained process bisimilarity and π-process open bisimilarity.

Definition 16 (Constrained Process)
A constrained process is a pair $\langle P, D\rangle$, where P is a π-process and D is a distinction such that $\mathrm{n}(D) \subseteq \mathrm{fn}(P)$. ■

We let $\mathcal{CP}$ denote the set of all constrained processes. $\mathcal{CP}$ is ranged over by A and B.

Definition 17 (Free and Bound Names of Constrained Processes)
Let $A \stackrel{def}{=} \langle P, D\rangle \in \mathcal{CP}$. Then the free names of A, written $\mathrm{fn}(A)$, are defined as $\mathrm{fn}(A) = \mathrm{fn}(P)$ and the bound names of A, written $\mathrm{bn}(A)$, are defined as $\mathrm{bn}(A) = \mathrm{bn}(P)$.

■

Application of a substitution σ to a constrained process $A \stackrel{def}{=} \langle P, D\rangle$ is written $A\sigma$ and abbreviates $\langle P\sigma, D\sigma\rangle$.

The transitions of a constrained process $\langle P, D\rangle$ are defined from those of the π-process P as

$$\frac{P \stackrel{(M,\alpha)}{\longrightarrow} P'}{\langle P, D\rangle \stackrel{(M,\alpha)}{\longrightarrow} \langle P', D^M_{P,\alpha} \cap \mathrm{fn}(P')\rangle} \qquad M \text{ respects } D$$

As it is seen $\langle P, D\rangle$ can make the "same" transitions as P except those where the condition M of the transition conflicts with the distinction D.

For two constrained processes to be bisimilar they must fulfill two requirements. The first one is that they are ***compatible***, e.i. their distinctions must agree on the common names.

Definition 18 (Compatibility)
The constrained processes $\langle P, D\rangle$ and $\langle Q, E\rangle$ are compatible, written $\langle P, D\rangle \Downarrow \langle Q, E\rangle$, if $D \cap \mathrm{fn}(Q) = E \cap \mathrm{fn}(P)$. A relation $\mathcal{E}$ on constrained processes is compatible if $A \Downarrow B$ for each pair $(A, B) \in \mathcal{E}$. ■

The second requirement for two constrained processes A and B to be bisimilar is that (A, B) is contained in a $\frown$***-bisimulation***.

Definition 19 ($\frown$-bisimulation)
A relation $\mathcal{E}$ on constrained processes is a $\frown$-bisimulation if $\langle P, D\rangle\ \mathcal{E}\ \langle Q, E\rangle$ implies

(i) for all M, α, and $\langle P', D'\rangle$ such that $\langle P, D\rangle \stackrel{(M,\alpha)}{\longrightarrow} \langle P', D'\rangle$, with $\mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \emptyset$, there exist some N, β, and $\langle Q', E'\rangle$ such that $\langle Q, E\rangle \stackrel{(N,\beta)}{\longrightarrow} \langle Q', E'\rangle$ and
   - $M \triangleright N$,
   - $\alpha = \beta\sigma_M$, and
   - $\langle P', D'\rangle \mathcal{E} \langle Q'\sigma_M, (D \cup E)^M_{(P,Q),\alpha} \cap \mathrm{fn}(Q'\sigma_M)\rangle$, and

(ii) the converse, with the role of $\langle P, D\rangle$ and $\langle Q, E\rangle$ exchanged.

■

From the definition of constrained bisimulation we define the notion of ***constrained bisimilarity***.

Definition 20 (Constrained Bisimilarity)
The constrained processes A and B are constrained bisimilar, written $A \frown B$, if $A \Downarrow B$ and there exists a $\frown$-bisimulation $\mathcal{E}$ such that $A\ \mathcal{E}\ B$. ■

The following theorem shows the connection between π-process open bisimilarity, $\sim$, and constrained bisimilarity, $\frown$.

Theorem 1 (Characterization of $\sim$ in terms of $\frown$)
$P \sim_D Q$ if and only if $\langle P, D \cap \mathrm{fn}(P)\rangle \frown \langle Q, D \cap \mathrm{fn}(Q)\rangle$ ■

4.5 Non-Redundant Transitions

As it is seen from the definition of $\frown$-bisimulation, it suffers from some of the same problems as the definition of open bisimulation with respect to constructing an algorithm based on partition refinement. A name emitted by $\langle P, D\rangle$ may not occur free in $\langle Q, E\rangle$ and the substitution σ_M determined by the condition M in a transition of $\langle P, D\rangle$ must be applied to the derivative of $\langle Q, E\rangle$. To remove the latter of these problems, dependency 2 described in section 4.3, [10] introduced the notion of ***non-redundant transitions*** of a constrained process.

Intuitively a transition $A \stackrel{(M,\alpha)}{\longrightarrow} A'$ is non-redundant if there does not exist another transition $A \stackrel{(N,\beta)}{\longrightarrow} A''$ where M implies N, $\alpha = \beta\sigma_M$, and A' and A'' are bisimilar.

Definition 21 (Non-Redundant Transitions)
Let $\mathcal{D}$ be a relation on constrained processes. A transition $\langle P, D\rangle \stackrel{(M,\alpha)}{\longrightarrow} \langle P', D'\rangle$ is redundant for $\mathcal{D}$ if there exists a transition $\langle P, D\rangle \stackrel{(N,\beta)}{\longrightarrow} \langle P'', D''\rangle$ such that

(i) $M \not\triangleleft\triangleright N$,

(ii) $\alpha = \beta\sigma_M$, and

(iii) $\langle P', D'\rangle \; \mathcal{D} \; \langle P''\sigma_M, D^M_{P,\alpha} \cap \mathrm{fn}(P''\sigma_M)\rangle$.

A transition $A \stackrel{(M,\alpha)}{\longrightarrow} A'$ is non-redundant for $\mathcal{D}$, written $A \stackrel{(M,\alpha)}{\longrightarrow} A' \in \mathrm{nr}(\mathcal{D})$, if it is not redundant for $\mathcal{D}$. ■

Computing the non-redundant transitions of a process is as difficult as checking for bisimilarity[10].

The following lemma shows that when we have two bisimilar constrained processes A and B then, if A has a non-redundant transition, B can match it with a non-redundant transition with the same condition and action.

Lemma 1
If $A \frown B$ and $A \stackrel{(M,\alpha)}{\longrightarrow} A' \in \mathrm{nr}(\frown)$ with $\mathrm{bn}(\alpha) \cap \mathrm{fn}(B) = \emptyset$ then there exists a B' such that $B \stackrel{(M,\alpha)}{\longrightarrow} B' \in \mathrm{nr}(\frown)$ and $A' \frown B'$. ■

A non-redundant transition for a relation is also non-redundant for a subset of that relation. This is stated in the following lemma.

Lemma 2
If $\mathcal{D} \subseteq \mathcal{E}$ then $\mathrm{nr}(\mathcal{E}) \subseteq \mathrm{nr}(\mathcal{D})$. ■

4.6 Active Names

To avoid dependency 1 described in section 4.3, the choice of bound names of the actions of matching transitions for bisimilar constrained processes should be made local to the processes. Since the free names of two bisimilar constrained processes are not necessarily the same, the choice cannot be based on these. [10] has shown that the choice can be based on ***active names*** since these are the same for bisimilar constrained processes. Intuitively the active names of a constrained process are the subset of the free names which influence the behavior of the process.

Definition 22 (Active Names)
Let $\mathcal{D}$ be a relation on constrained processes and let $\mathrm{an}_{\mathcal{D}}$ be the least fixed point of the function $\psi : (\mathcal{CP} \to \mathcal{P}(\mathcal{N})) \to (\mathcal{CP} \to \mathcal{P}(\mathcal{N}))$ defined as

$$\psi(f)(A) = \bigcup_{\{M,\alpha,A' | A \xrightarrow{(M,\alpha)} A' \in \mathrm{nr}(\mathcal{D})\}} \mathrm{fn}(M,\alpha) \cup (f(A') \setminus \mathrm{bn}(\alpha))$$

The name n is active in A with respect to $\mathcal{D}$ if $n \in \mathrm{an}_{\mathcal{D}}(A)$ and otherwise n is inactive in A with respect to $\mathcal{D}$. ■

Computing the active names of a process is as difficult as checking for bisimilarity[10].

An active name in a constrained process with respect to a relation is also active with respect to a subset of this relation. This is stated in the following lemma.

Lemma 3
If $\mathcal{D} \subseteq \mathcal{E}$ then $\mathrm{an}_{\mathcal{E}} \subseteq \mathrm{an}_{\mathcal{D}}$. ■

The following lemma shows that two bisimilar constrained processes have the same set of active names with respect to $\frown$.

Lemma 4
If $A \frown B$ then $\mathrm{an}_{\frown}(A) = \mathrm{an}_{\frown}(B)$. ■

This lemma implies that if for each transition $A \xrightarrow{(M,\alpha)} A'$ the possibly bound name in α is converted to the least not active name of A then the actions on matching transitions are equal.

Non-redundant transitions with their actions converted as described are called ***normalized transitions***.

Definition 23 (Normalized Transitions)
The normalized transitions for a relation $\mathcal{D}$ have the form $A \stackrel{(M,\alpha)}{\longmapsto}_{\mathcal{D}} A'$ and are defined from the rules Norm1 and Norm2 given by

$$[\text{Norm1}] \quad \frac{A \stackrel{(M,\alpha)}{\longrightarrow} A' \in \text{nr}(\mathcal{D})}{A \stackrel{(M,\alpha)}{\longmapsto}_{\mathcal{D}} A'} \qquad \text{bn}(\alpha) = \emptyset$$

$$[\text{Norm2}] \quad \frac{A \stackrel{(M,\alpha)}{\longrightarrow} A' \in \text{nr}(\mathcal{D})}{A \stackrel{(M,\beta)}{\longmapsto}_{\mathcal{D}} A'\{v/y\ y/v\}} \qquad \begin{array}{l} \alpha = a(y) \text{ or } \alpha = \bar{a}(y) \text{ and} \\ \beta \stackrel{def}{=} \begin{cases} a(v) & \text{if } \alpha = a(y) \\ \bar{a}(v) & \text{if } \alpha = \bar{a}(y) \end{cases} \end{array}$$

where $y \stackrel{def}{=} \min\{\mathcal{N} \setminus \text{fn}(A)\}$ and $v \stackrel{def}{=} \min\{\mathcal{N} \setminus \text{an}_{\mathcal{D}}(A)\}$ ■

Note that the substitution applied to A' in the rule Norm2 is necessary since the first inactive name may also occur free.

The normalized transitions are used to form a new kind of bisimulation relation on constrained processes called ***active names bisimulation***.

Definition 24 (Active Names Bisimulation)
A relation $\mathcal{D}$ is an $\frown_{\text{an}}$-bisimulation if $A \mathcal{D} B$ implies that

(i) if $A \stackrel{(M,\alpha)}{\longmapsto}_{\mathcal{D}} A'$ then there exists some B' such that $B \stackrel{(M,\alpha)}{\longmapsto}_{\mathcal{D}} B'$ and $A' \mathcal{D} B'$ and

(ii) the converse, with the rule of A and B exchanged.

■

Finally ***active names bisimilarity*** between constrained processes is defined from active names bisimulation.

Definition 25 (Active Names Bisimilarity)
The processes A and B are active names bisimilar, written $A \frown_{\text{an}} B$, if $A \Downarrow B$ and there exists an $\frown_{\text{an}}$-bisimulation $\mathcal{E}$ such that $A \mathcal{E} B$. ■

The following theorem states that $\frown$ and $\frown_{\text{an}}$ coincide.

Theorem 2 (Characterization of $\frown$ in terms of $\frown_{\text{an}}$)
$A \frown B$ if and only if $A \frown_{\text{an}} B$ ■

4.7 The Iterative Method

As it is seen from the previous sections, open bisimilarity can be characterized by $\frown_{\text{an}}$ which avoids the three dependencies described in section 4.3. Using this characterization [10] defined an iterative method that computes open bisimilarity through a sequence of approximations that progressively refine a partition of the set of all constrained processes. The method makes use of the functions $\Psi_{\mathcal{D}}$ and Φ defined as follows.

Definition 26 (The Functions $\Psi_{\mathcal{D}}$ and Φ)
Let $\mathcal{D}$ be a relation on constrained processes. The function $\Psi_{\mathcal{D}} : \mathcal{P}(\mathcal{CP}\times\mathcal{CP}) \rightarrow \mathcal{P}(\mathcal{CP}\times\mathcal{CP})$ is defined as

$(A, B) \in \Psi_{\mathcal{D}}(\mathcal{E})$ if and only if

(i) $A \; \mathcal{D} \; B$,

(ii) for all M, α, and A' where $A \overset{(M,\alpha)}{\longmapsto}_{\mathcal{D}} A'$ there exists B' such that $B \overset{(M,\alpha)}{\longmapsto}_{\mathcal{D}} B'$ and $A' \; \mathcal{E} \; B'$, and

(iii) the converse with the role of A and B exchanged.

The function $\Phi : \mathcal{P}(\mathcal{CP}\times\mathcal{CP}) \rightarrow \mathcal{P}(\mathcal{CP}\times\mathcal{CP})$ is defined as $\Phi(\mathcal{D}) =$ greatest fixed point of $\Psi_{\mathcal{D}}$.

■

Function Φ is well-defined since, for each relation $\mathcal{D}$, the function $\Psi_{\mathcal{D}}$ is monotone. From the definition of Φ and $\frown_{\text{an}}$-bisimulation it can be seen that

Theorem 3
$\mathcal{D}$ is a fixed point of Φ if and only if $\mathcal{D}$ is an $\frown_{\text{an}}$-bisimulation. ■

Even though Φ appears suited for a partition refinement algorithm it can not be used directly because it is not monotone. Since Φ is not monotone it is not known whether it has a greatest fixed point and even though a greatest fixed point does exist Tarski's theorem[12] cannot be used to compute it[10]. Because of this [10] defined a chain $\{\Phi^i\}_i$ and showed that the maximum compatible subset of the limit of this chain coincides with $\frown$.

Definition 27 (The Relation Φ^i)

$$\begin{aligned} \Phi^0 &\overset{def}{=} \mathcal{CP} \times \mathcal{CP} \\ \Phi^{i+1} &\overset{def}{=} \Phi(\Phi^i) \\ \Phi^i &\overset{def}{=} \bigcap_{j \,<\, i} \Phi^j \qquad \text{if } i \text{ is a limit ordinal.} \end{aligned}$$

■

Definition 28 (The Relation $\simeq$)
The relation $\simeq$ on constrained processes is defined by

$$\simeq \stackrel{def}{=} \lim_i \Phi^i$$

■

The relation $\simeq$ is well-defined since by definition of Φ the chain $\{\Phi^i\}_i$ is non-increasing.

Theorem 4 (Characterization of $\frown$ in terms of $\simeq$)
$A \frown B$ if and only if $(A, B) \in (\simeq \cap \Downarrow)$ ■

From theorem 1 it follows that open bisimilarity of two π-processes with respect to a distinction can be proven using the iterative method, that is by computing the limit of the chain $\{\Phi^i\}_i$.

Corollary 1
$P \sim_D Q$ if and only if $\langle P, D \cap \mathrm{fn}(P)\rangle \simeq \langle Q, D \cap \mathrm{fn}(Q)\rangle$ ■

The following lemma shows that two bisimilar constrained processes have the same set of active names with respect to Φ^i.

Lemma 5
If $A \frown B$ then $\mathrm{an}_{\Phi^i}(A) = \mathrm{an}_{\Phi^i}(B)$ for all i. ■

The next section describes the algorithm given by [10] for checking open bisimilarity.

4.8 The Algorithm

In this section we present the algorithm proposed by [10] for checking for open bisimilarity between two processes P and Q with respect to the distinction D. The algorithm can be seen in figure 4.1.

1. Generate the saturated state graphs (S_P, T_P) and (S_Q, T_Q) for the constrained processes $\langle P, D \cap \mathrm{fn}(P)\rangle$ and $\langle Q, D \cap \mathrm{fn}(Q)\rangle$.
2. Initialize $\mathcal{W}$ to be the partition consisting of one block containing all the constrained processes $S_P \cup S_Q$.
3. Repeat the following steps until the partition $\mathcal{W}$ becomes stable.
   - 3-1. Set *Nonred* to be the subset of transitions in $T_P \cup T_Q$ that are non-redundant for $\mathcal{W}_\sim$.
   - 3-2. Compute the active names with respect to $\mathcal{W}_\sim$ for each process in $S_P \cup S_Q$.
   - 3-3. If necessary refine the partition $\mathcal{W}$ so that all processes in a block of $\mathcal{W}$ have the same set of active names.
   - 3-4. Set *Normtrans* to be the normalized transitions for $\mathcal{W}_\sim$ generated by the transitions in *Nonred*.
   - 3-5. Apply a partition refinement algorithm to the partition $\mathcal{W}$ using the transitions in *Normtrans*. Set $\mathcal{W}$ to be the resulting partition.
4. Check if $\langle P, D \cap \mathrm{fn}(P)\rangle$ and $\langle Q, D \cap \mathrm{fn}(Q)\rangle$ are in the same block of the partition $\mathcal{W}$.

Figure 4.1: Algorithm for checking $P \sim_D Q$.

The algorithm terminates if the saturated state graphs generated in step 1 are finite[10]. This is the case for the set of ***finite control processes***, $\mathcal{P}r_{fc}$, i.e. processes generated by the following grammar[2].

$$\begin{aligned} P_{fc} ::= &\ R \mid P_{fc}|P_{fc} \mid (\nu a)P_{fc} \\ R ::= &\ \mathbf{0} \mid \alpha.R \mid [a=b]R \mid R+R \mid K\langle\tilde{a}\rangle \mid (\nu a)R \end{aligned}$$

where a process identifier K has the form $K \stackrel{def}{=} (\tilde{b})R$.

To see that algorithm does not always terminate consider the process $P \stackrel{def}{=} K\langle a\rangle$, where $K \stackrel{def}{=} (a)(\bar{a}a \mid K\langle a\rangle)$. Since $P \stackrel{(\emptyset,\bar{a}a)}{\longrightarrow} 0 \mid \underbrace{\bar{a}a \mid \cdots \mid \bar{a}a}_{k} \mid K\langle a\rangle$ for all $k \geq 0$ the saturated state graph for P is infinite.

Some comments, noted by [10], on some of the steps of the algorithm in figure 4.1 are given below.

Step 1. The saturated state graph of a process A_0 is the pair (S, T), where S is the minimal set of constrained processes and T is the minimal set of transitions between processes

in S where $A_0 \in S$, and S and T are closed under the operations **Sat-trans**, **Sat-nonred**, and **Sat-bunch** defined as follows.

Sat-trans. If $A \in S$ and $A \stackrel{(M,\alpha)}{\longrightarrow} A'$ with $\mathrm{bn}(\alpha) \subseteq \{\min\{\mathcal{N} \setminus \mathrm{fn}(A)\}\}$ then $A' \in S$ and $A \stackrel{(M,\alpha)}{\longrightarrow} A' \in T$.

Sat-nonred. If $\langle P, D\rangle \stackrel{(M,\alpha)}{\longrightarrow} \langle P', D'\rangle \in T$ and $\langle P, D\rangle \stackrel{(N,\beta)}{\longrightarrow} \langle P'', D''\rangle \in T$ with $M \not\lhd\rhd N$ and $\alpha = \beta\sigma_M$ then $\langle P''\sigma_M, D^M_{P,\alpha} \cap \mathrm{fn}(P''\sigma_M)\rangle \in S$.

Sat-bunch. If $A \stackrel{(M,\alpha)}{\longrightarrow} A' \in T$ with $\mathrm{bn}(\alpha) = \{y\}$ where $y \stackrel{def}{=} \min\{\mathcal{N} \setminus \mathrm{fn}(A)\}$ then $A'\{v/y\ y/v\} \in S$ for all $v < y$.

Step 3. Each iteration corresponds to an application of the function Φ defined in the previous section.

Step 3-1. Following definition 21 compute the non-redundant transitions. This can be done efficiently if for each transition $\langle P, D\rangle \stackrel{(M,\alpha)}{\longrightarrow} \langle P', D'\rangle$ a list of the processes $\langle P''\sigma_M, D^M_{P,\alpha} \cap \mathrm{fn}(P''\sigma_M)\rangle$ such that $\langle P, D\rangle \stackrel{(N,\beta)}{\longrightarrow} \langle P'', D''\rangle \in T$ with $M \not\lhd\rhd N$ and $\alpha = \beta\sigma_M$ has been constructed before step 3. A transition $A \stackrel{(M,\alpha)}{\longrightarrow} A'$ is then non-redundant for $\mathcal{W}_\sim$ if and only if none of the processes in the list associated with it is in the same block as A'.

Step 3-2. The active names can be efficiently computed from the non-redundant transitions computed in Step 3-1 using a transitive-closure algorithm.

Step 3-4. The normalized transitions are generated by applying the inference rules Norm1 or Norm2 to each transition in *Nonred* (when generating the saturated state graphs the derivatives of normalized transitions are added to S_P or S_Q, so no new processes need to be added).

4.9 Examples

In this section we present two examples showing how the open bisimulation checking algorithm works.

Example 1:

The first example shows what happens when the algorithm is used for checking whether the processes $P \stackrel{def}{=} a(x).\mathbf{0} \mid \bar{b}y.\mathbf{0}$ and $Q \stackrel{def}{=} a(x).\bar{b}y.\mathbf{0} + \bar{b}y.a(x).\mathbf{0} + [a = b]\tau.\mathbf{0}$ are open bisimilar with respect to the empty distinction. Assume that the names in $\mathcal{N}$ are ordered such that $a < b < c < \ldots < x < y < \ldots$. Then the algorithm runs as follows.

Step 1. The generated saturated state graphs for the constrained processes $A \stackrel{def}{=} \langle P, \emptyset\rangle$ and $B \stackrel{def}{=} \langle Q, \emptyset\rangle$ are shown in figure 4.2. All states and transitions are added using

the **Sat-trans** operation. We will denote the processes in the graphs by $S_1, \ldots, S_8$ such that S_i refers to the constrained process with i associated with it in the figure.

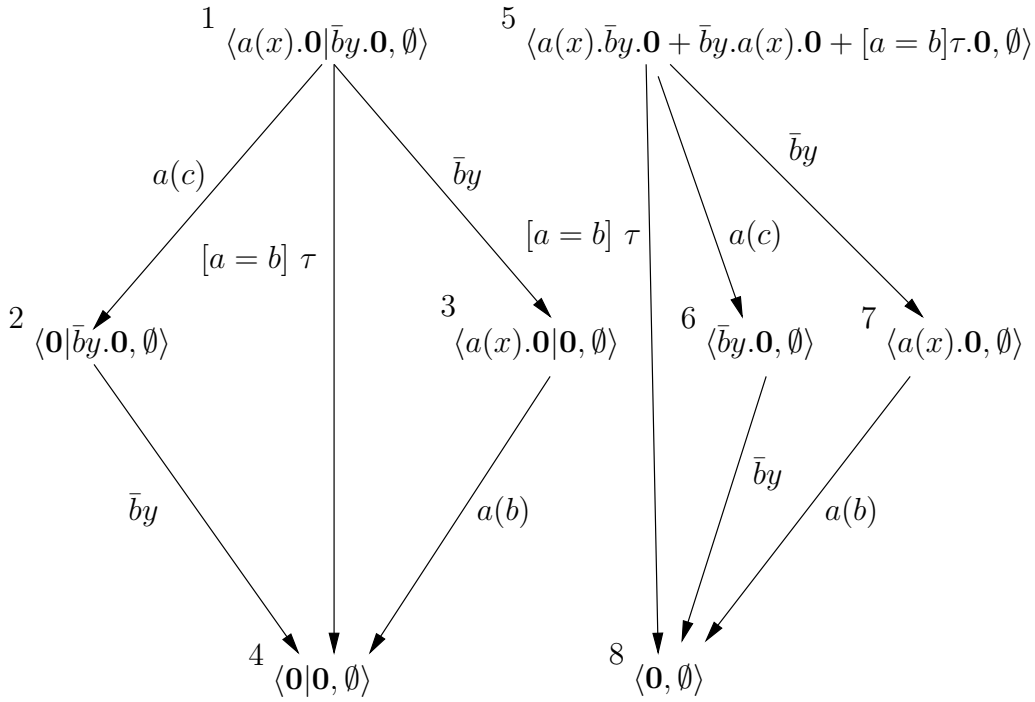

Figure 4.2: The saturated state graphs of the constrained processes A and B.

Step 2. The partition $\mathcal{W}$ is initialized. It consists of one block containing all states/processes in the saturated state graphs, i.e. $\mathcal{W} := \{\{S_1, \ldots, S_8\}\}$.

Step 3. Two iterations of step 3 are needed to stabilize $\mathcal{W}$.

Iteration 1.

Step 3-1. All transition in the graph are non-redundant with respect to $\mathcal{W}_\sim$, so *Nonred* contains all transitions in the graphs.

Step 3-2. The active names of the processes in the graphs are computed. $\text{an}_{\mathcal{W}_\sim}(S_1) = \text{an}_{\mathcal{W}_\sim}(S_5) = \{a, b, y\}$, $\text{an}_{\mathcal{W}_\sim}(S_2) = \text{an}_{\mathcal{W}_\sim}(S_6) = \{b, y\}$, $\text{an}_{\mathcal{W}_\sim}(S_3) = \text{an}_{\mathcal{W}_\sim}(S_7) = \{a\}$, and $\text{an}_{\mathcal{W}_\sim}(S_4) = \text{an}_{\mathcal{W}_\sim}(S_8) = \emptyset$.

Step 3-3. $\mathcal{W}$ is refined using the sets of active names. So $\mathcal{W}$ is split into four blocks, $\mathcal{W} := \{\{S_1, S_5\}, \{S_2, S_6\}, \{S_3, S_7\}, \{S_4, S_8\}\}$.

Step 3-4. The normalized transitions are computed. These are the same as the transitions in *Nonred*.

Step 3-5. The refinement of $\mathcal{W}$ using the normalized transition do not result in any new blocks in $\mathcal{W}$.

Iteration 2. In the second iteration of step 3 nothing happens to $\mathcal{W}$, so $\mathcal{W}$ has become stable.

Step 4. $A = S_1$ and $B = S_5$ are in the same block of $\mathcal{W}$, so P and Q are open bisimilar with respect to the empty distinction as expected.

Example 2:

In the second example we check whether the processes $P \stackrel{def}{=} K_1\langle a, e\rangle$ and $Q \stackrel{def}{=} K_2\langle e\rangle$ are open bisimilar with respect to the empty distinction. The process identifiers K_1 and K_2 are defined as follows.

$$K_1 \stackrel{def}{=} (a,e)(K_2\langle e\rangle + [a=e]a(b).K_1\langle a,b\rangle)$$
$$K_2 \stackrel{def}{=} (e)(e(b).K_2\langle b\rangle)$$

Assume that the names in $\mathcal{N}$ are ordered such that $a < b < c < d < e < \ldots$. Then the algorithm runs as follows.

Step 1. The generated saturated state graphs for the constrained processes $A \stackrel{def}{=} \langle P, \emptyset\rangle$ and $B \stackrel{def}{=} \langle Q, \emptyset\rangle$ are shown in figure 4.3.

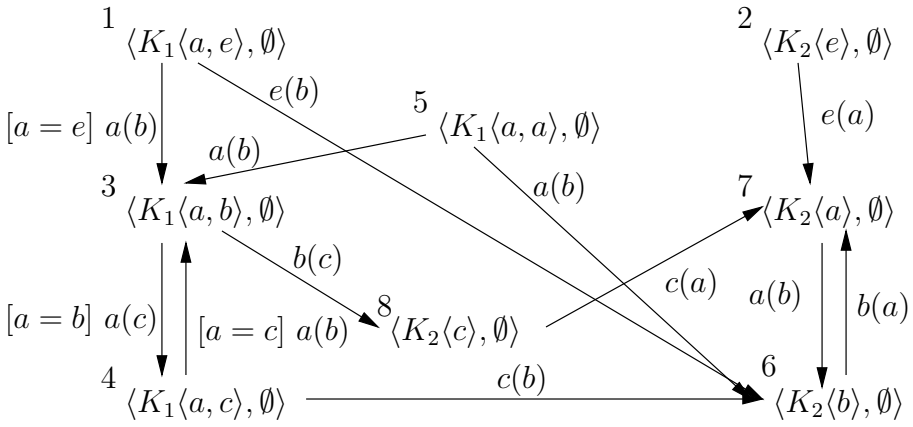

Figure 4.3: The saturated state graphs of the constrained processes A and B.

All states and transitions, except S_5, could have been added using the **Sat-trans** operation. S_5 is added using the **Sat-bunch** operation on the transition from S_1 to

S_3. Although there are transitions that fulfill the requirements in the **Sat-nonred** operation no states are are added using this operation since the states that need to be added to the graphs are already included using the **Sat-trans** operation.

Step 2. The partition $\mathcal{W}$ is initialized. It consists of one block containing all states/processes in the saturated state graphs, i.e. $\mathcal{W} := \{\{S_1, \ldots, S_8\}\}$.

Step 3. Two iterations of step 3 are needed to stabilize $\mathcal{W}$.

Iteration 1.

Step 3-1. All transitions in the graphs except the three transitions $S_1 \overset{([a=e],a(b))}{\longrightarrow} S_3$, $S_3 \overset{([a=b],a(c))}{\longrightarrow} S_4$, and $S_4 \overset{([a=c],a(b))}{\longrightarrow} S_3$ are non-redundant with respect to $\mathcal{W}_\sim$.

Step 3-2. The active names of the processes in the graphs are computed using the non-redundant transitions. $\text{an}_{\mathcal{W}_\sim}(S_1) = \text{an}_{\mathcal{W}_\sim}(S_2) = \{e\}$, $\text{an}_{\mathcal{W}_\sim}(S_3) = \text{an}_{\mathcal{W}_\sim}(S_6) = \{b\}$, $\text{an}_{\mathcal{W}_\sim}(S_4) = \text{an}_{\mathcal{W}_\sim}(S_8) = \{c\}$, and $\text{an}_{\mathcal{W}_\sim}(S_5) = \text{an}_{\mathcal{W}_\sim}(S_7) = \{a\}$.

Step 3-3. $\mathcal{W}$ is refined using the sets of active names. So $\mathcal{W}$ is split in to four blocks, $\mathcal{W} := \{\{S_1, S_2\}, \{S_3, S_6\}, \{S_4, S_8\}, \{S_5, S_7\}\}$.

Step 3-4. The normalized transitions are computed and are shown in figure 4.4.

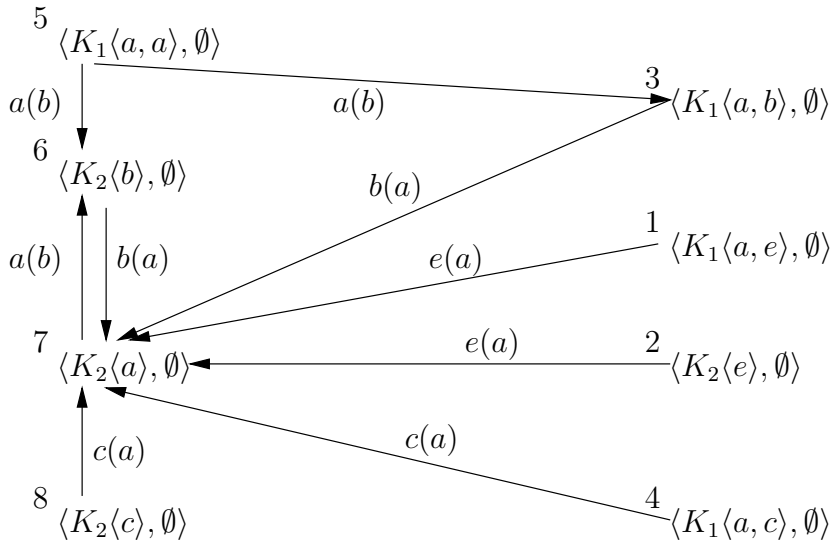

Figure 4.4: The state graphs with normalized transitions of the constrained processes A and B.

Step 3-5. The refinement of $\mathcal{W}$ using the normalized transition does not result in any new blocks in $\mathcal{W}$.

Iteration 2. In the second iteration of step 3 nothing happens to $\mathcal{W}$, so $\mathcal{W}$ has become stable.

Step 4. $A = S_1$ and $B = S_2$ are in the same block of $\mathcal{W}$, so P and Q are open bisimilar with respect to the empty distinction as expected.

Chapter 5

Implementation of Open Bisimulation Checker

In this chapter we describe our first implementation of the open bisimulation checker algorithm presented in section 4.8. The algorithm is implemented using Standard ML. In our first implementation no attempts have been made to optimize the algorithm. It just follows the original ideas presented in [10].

In the following we describe the implementation of each step in the algorithm. For each of the steps we give a pseudo code algorithm instead of the actual SML-code. This is done since these algorithms take up less space and are more readable than the actual code which just follows the algorithms. In the pieces of pseudo code presented key words are converted to uppercase and comments are enclosed in (* and *).

Furthermore, we give the time-complexity of the implementation of each step. Before describing the steps of the algorithm we describe the data types for processes and the saturated state graphs, simple functions, how to find the transitions needed in step 1 of the algorithm, and the partition refinement algorithm used in step 3-5.

5.1 Data Types

In this section data types for processes and the state graphs that are built when testing for open bisimilarity are described.

5.1.1 Data Types for π-Processes

The SML data type definitions for π-processes are presented in figure 5.1.

```
DATATYPE names = name OF STRING * INT

DATATYPE prefixes = tau
                  | inputPrefix OF names * names
                  | outputPrefix OF names * names
                  | boundOutputPrefix OF names * names

DATATYPE process  = nil
                  | prefixProcess OF prefixes * process
                  | matchProcess OF names * names * process
                  | parallelProcess OF process * process
                  | newProcess OF names * process
                  | sumProcess OF process * process
                  | recursionProcess OF STRING * names LIST

DATATYPE processIdentifiers =
           processIdentifier OF STRING * names LIST * process

DATATYPE transitions = transition OF condition * prefixes *
                       process
```

Figure 5.1: SML data type definitions for π-processes.

An element in the ordered set $\mathcal{N}$ of names is represented by a string and an integer. The string represents the name itself and the integer ensures that the names can be ordered in any possible way. There are four types of prefixes; the silent prefix is represented by tau and the three prefixes input, free output, and bound output are represented by a pair of names, the first name being the object name and the second the subject name. π-processes can be built using the constructors of section 3.1 and each type of process has a special representation. The inactive process is represented by nil, a matching process as a three-tuple containing the two names that have to match and the rest of the process, a prefix process by a prefix and a process, both parallel processes and sum processes as a pair of processes, and a restricted process by a name and a process. A recursive process is represented by a string and a list of names where the string is the name of a process identifier and the list of names is the argument to the process identifier. A process identifier is represented by a string, a list of names, and a π-process where the string is the name of the process identifier and the list of names consists of the free names of the π-process. A transition of a process is represented by a three-tuple containing the condition, the prefix, and the derivative of the process with respect to that transition.

5.1.2 Substitutions, Distinctions, and Conditions

Substitutions are represented as lists of pairs of names where the first name of a pair of names in the list is the name to be substituted by the second name (see figure 5.2).

```
TYPE substitution = (names * names) LIST
```

Figure 5.2: SML data type definitions for substitutions.

Distinctions and conditions are represented as lists of pairs of names (see figure 5.3). For a distinction the list contains the pairs of names that must be be distinct, and for a condition the list contains the pair of names (a, b) if $[a = b]$ is a match of the condition.

```
TYPE distinction = (names * names) LIST
TYPE condition = (names * names) LIST
```

Figure 5.3: SML data type definitions for distinctions and conditions.

With these representations of substitutions and conditions it can be seen that a substitution σ_M does not need to be constructed explicitly as it is the same as M_c.

Given a condition M we canonize it as follows.

The names of the pairs in M are grouped in equality classes. For each of these equality classes a condition is formed by pairing the smallest name of the equality class with each of the other names of the equality class, letting the smallest name be the second name of these pairs. The pairs of these conditions are rearranged such that the pairs of names of each condition are ordered in ascending order with respect to the first name of the pairs. The conditions of the equality classes are connected and the pairs are rearranged in such a way that all the pairs of names of this condition are ordered in ascending order with respect to the second name of the pairs of names.

Functions to perform simple operations such as substitutions, canonization of conditions, and testing whether two processes are α-convertible etc. have been implemented.

5.1.3 Data Types for Constrained Processes and the State Graphs

The SML data type definitions for constrained processes and the state graphs are presented in figure 5.4.

```
DATATYPE constrainedProcesses = cp OF process * distinction

DATATYPE cpTransitions = cpTransition OF condition * prefixes *
                         constrainedProcesses

DATATYPE states = state OF constrainedProcesses *
           (transitions * int * int LIST) LIST

DATATYPE graphs = graph OF (int * states) LIST
```

Figure 5.4: SML data type definitions for constrained processes and the state graph.

The state graph is represented by a list of pairs of an integer and a state where the integer indicates the state number. Each state is represented by a constrained process and a list of three-tuples of a transition, an integer, and a list of integers. The integer indicates which state is reached by taking the transition from the constrained process, and the list of integers is used for computing the non-redundant transitions efficiently. A constrained process is represented by a π-process and a distinction.

5.2 Finding Transitions

In this section we describe how to find the transitions needed to generate the saturated state graphs in step 1. Since each of the graphs must be closed under the **Sat-trans** operation we need all transitions $A \stackrel{(M,\alpha)}{\longrightarrow} A'$ with $\mathrm{bn}(\alpha) \subseteq \{\min\{\mathcal{N} \setminus \mathrm{fn}(A)\}\}$ for each constrained process A in the graphs. Before we can find the transition of a constrained process $A = \langle P, D\rangle$ we need the transitions of the π-process P.

5.2.1 Transitions for π-Processes

To find the transitions of a π-process P we use the transition system in table 4.1. Since the transitions added to the saturated state graphs using the **Sat-trans** operation have the form $A \stackrel{(M,\alpha)}{\longrightarrow} A'$ with $\mathrm{bn}(\alpha) \subseteq \{\min\{\mathcal{N} \setminus \mathrm{fn}(A)\}\}$ it is only necessary to find the transitions $P \stackrel{(M,\alpha)}{\longrightarrow} P'$ with $\mathrm{bn}(\alpha) \subseteq \{\min\{\mathcal{N} \setminus \mathrm{fn}(P)\}\}$. The transitions needed are given by the function $[\![\cdot]\!]_{TR}$ defined as

$$
\begin{aligned}
[\![P]\!]_{TR} = &\{P \stackrel{(M,\alpha)}{\longrightarrow} P' \mid P \stackrel{(M,\alpha)}{\longrightarrow} P' \in [\![P]\!]_T \wedge \mathrm{bn}(\alpha) = \emptyset\} \cup \\
&\{P \stackrel{(M,a(y))}{\longrightarrow} P' \mid \exists P \stackrel{(M,a(w))}{\longrightarrow} P'' \in [\![P]\!]_T.(y = \min\{\mathcal{N} \setminus \mathrm{fn}(P)\} \wedge P' = P''\{y/w\})\} \cup \\
&\{P \stackrel{(M,\overline{a}(y))}{\longrightarrow} P' \mid \exists P \stackrel{(M,\overline{a}(w))}{\longrightarrow} P'' \in [\![P]\!]_T.(y = \min\{\mathcal{N} \setminus \mathrm{fn}(P)\} \wedge P' = P''\{y/w\})\}
\end{aligned}
$$

where $[\![\cdot]\!]_T$ is defined as the least function satisfying the following.

$$
\begin{aligned}
[\![\mathbf{0}]\!]_T &= \emptyset \\
[\![\alpha.P]\!]_T &= \{\alpha.P \stackrel{(\emptyset,\alpha)}{\longrightarrow} P\} \\
[\![P+Q]\!]_T &= [\![P]\!]_T \cup [\![Q]\!]_T \\
[\![P|Q]\!]_T &= \{P|Q \stackrel{(M,\alpha)}{\longrightarrow} P'|Q \mid P \stackrel{(M,\alpha)}{\longrightarrow} P' \in [\![P]\!]_T \wedge \mathrm{bn}(\alpha) = \emptyset\} \cup \\
&\quad \{P|Q \stackrel{(M,\alpha)}{\longrightarrow} P'|Q \mid \exists P \stackrel{(M,a(x))}{\longrightarrow} P'' \in [\![P]\!]_T.(\alpha = a(y) \wedge P' = P''\{y/x\} \wedge \\
&\qquad\qquad y = \min\{\mathcal{N} \setminus \mathrm{fn}(P,Q)\})\} \cup \\
&\quad \{P|Q \stackrel{(M,\alpha)}{\longrightarrow} P'|Q \mid \exists P \stackrel{(M,\overline{a}(x))}{\longrightarrow} P'' \in [\![P]\!]_T.(\alpha = \overline{a}(y) \wedge P' = P''\{y/x\} \wedge \\
&\qquad\qquad y = \min\{\mathcal{N} \setminus \mathrm{fn}(P,Q)\})\} \cup \\
&\quad \{P|Q \stackrel{(MN[a=b],\tau)}{\longrightarrow} P'|Q'\{y/x\} \mid P \stackrel{(M,\overline{a}y)}{\longrightarrow} P' \in [\![P]\!]_T \wedge Q \stackrel{(N,b(x))}{\longrightarrow} Q' \in [\![Q]\!]_T\} \cup \\
&\quad \{P|Q \stackrel{(MN[a=b],\tau)}{\longrightarrow} (\nu\ z)(P'|Q') \mid \exists P \stackrel{(M,\overline{a}(x))}{\longrightarrow} P'' \in [\![P]\!]_T, Q \stackrel{(N,b(y))}{\longrightarrow} Q'' \in [\![Q]\!]_T. \\
&\qquad\qquad (Q' = Q''\{z/y\} \wedge P' = P''\{z/x\} \wedge z = \min\{\mathcal{N} \setminus \mathrm{n}(P,Q)\})\} \cup \\
&\quad \{P|Q \stackrel{(M,\alpha)}{\longrightarrow} P|Q' \mid Q|P \stackrel{(M,\alpha)}{\longrightarrow} Q'|P \in [\![Q|P]\!]_T\} \\
[\![(\nu\ b)P]\!]_T &= \{(\nu\ b)P \stackrel{(M,\alpha)}{\longrightarrow} (\nu\ b)P' \mid P \stackrel{(M,\alpha)}{\longrightarrow} P' \in [\![P]\!]_T \wedge b \notin \mathrm{n}(M,\alpha)\} \cup \\
&\quad \{(\nu\ b)P \stackrel{(M,\alpha)}{\longrightarrow} (\nu\ b)P' \mid \exists P \stackrel{(M,a(b))}{\longrightarrow} P'' \in [\![P]\!]_T.(\alpha = a(y) \wedge P' = P''\{y/b\} \wedge \\
&\qquad\qquad y = \min\{\mathcal{N} \setminus (\mathrm{fn}(P) \cup \{b\})\} \wedge b \notin \mathrm{n}(M) \cup \{a\})\} \cup \\
&\quad \{(\nu\ b)P \stackrel{(M,\alpha)}{\longrightarrow} (\nu\ b)P' \mid \exists P \stackrel{(M,\overline{a}(b))}{\longrightarrow} P'' \in [\![P]\!]_T.(\alpha = \overline{a}(y) \wedge P' = P''\{y/b\} \wedge \\
&\qquad\qquad y = \min\{\mathcal{N} \setminus (\mathrm{fn}(P) \cup \{b\})\} \wedge b \notin \mathrm{n}(M) \cup \{a\})\} \cup \\
&\quad \{(\nu\ b)P \stackrel{(M,\overline{a}(b))}{\longrightarrow} P' \mid P \stackrel{(M,\overline{a}b)}{\longrightarrow} P' \in [\![P]\!]_T \wedge b \notin \mathrm{n}(M) \cup \{a\}\} \\
[\![[a=b]P]\!]_T &= \{[a=b]P \stackrel{(M[a=b],\alpha)}{\longrightarrow} P' \mid P \stackrel{(M,\alpha)}{\longrightarrow} P' \in [\![P]\!]_T\} \\
[\![K\langle\tilde{a}\rangle]\!]_T &= \{K\langle\tilde{a}\rangle \stackrel{(M,\alpha)}{\longrightarrow} P' \mid K \stackrel{def}{=} (\tilde{b})P \wedge P\{\tilde{a}/\tilde{b}\} \stackrel{(M,\alpha)}{\longrightarrow} P' \in [\![P\{\tilde{a}/\tilde{b}\}]\!]_T\}
\end{aligned}
$$

When the transitions of a recursive π-process are computed special care must be taken. Consider the process $P \stackrel{def}{=} K\langle a,b\rangle$ where $(a,b)K \stackrel{def}{=} K\langle a,b\rangle + \overline{a}b.\mathbf{0}$, in this case $[\![P]\!]_T = [\![P]\!]_T \cup \{P \stackrel{(\emptyset,\overline{a}b)}{\longrightarrow} \mathbf{0}\}$. Since $[\![\cdot]\!]_T$ is defined as a least fixed point we can conclude $[\![P]\!]_T = \{P \stackrel{(\emptyset,\overline{a}b)}{\longrightarrow} \mathbf{0}\}$. To make the computation of transitions of a π-process P automatic we need to keep track of which process identifiers and instantiations of these we have already seen in the computation and if an identifier and an instantiation of it has been seen it should not contribute to the set of transitions of P.

5.2.2 Canonized Transitions for π-Processes

To make comparisons between transitions easier we convert each transition of a process to a canonical form as described in section 4.1. The canonical transitions for a π-process are given by

$$[\![P]\!]_{CTR} = \{P \xrightarrow{(M,\alpha)} P' \mid \exists P \xrightarrow{(N,\beta)} P'' \in [\![P]\!]_{TR}.(M = \text{canonize}(N) \wedge \alpha = \beta\sigma_M \wedge P' = P''\sigma_M)\}$$

5.2.3 Transitions for Constrained Processes

Finally, we find the transitions for a constrained process $A = \langle P, D\rangle$ by

$$[\![\langle P, D\rangle]\!]_{CPTR} = \{\langle P, D\rangle \xrightarrow{(M,\alpha)} \langle P', D^M_{P,\alpha}\rangle \mid P \xrightarrow{(M,\alpha)} P' \in [\![P]\!]_{CTR} \wedge M \text{ respects } D\}$$

5.3 Partition Refinement

In this section our implementation of the simple partition refinement algorithm of [8] is described. First some definitions are necessary.

Definition 29 ($E(S)$ and $E^{-1}(S)$)
Let E be a binary relation over the set U. For any subset $S \subseteq U$, $E(S) = \{y \mid \exists x \in S.(x, y) \in E\}$ and $E^{-1}(S) = \{x \mid \exists y \in S.(x, y) \in E\}$. ■

Definition 30 (Stable Partition)
If $B \subseteq U$ and $S \subseteq U$ then B is stable with respect to S if $B \subseteq E^{-1}(S)$ or $B \cap E^{-1}(S) = \emptyset$. A partition $\mathcal{W}$ of U is stable with respect to S if all the blocks of $\mathcal{W}$ are stable with respect to S. $\mathcal{W}$ is stable if it is stable with respect to each of its blocks. ■

The partition refinement algorithm uses a function, split$(S, \mathcal{W}, E)$, that given $S \subseteq U$ refines each block B of partition $\mathcal{W}$ to the two blocks $B' = B \cap E^{-1}(S)$ and $B'' = B \setminus E^{-1}(S)$ if $B \cap E^{-1}(S) \neq \emptyset$ and $B \setminus E^{-1}(S) \neq \emptyset$. The set $S \subseteq U$, is a splitter for a partition $\mathcal{W}$ and a relation E if split$(S, \mathcal{W}, E) \neq \mathcal{W}$. The partition refinement algorithm also uses a function, findSplitter$(\mathcal{W}, E)$. If there exists a block $B \in \mathcal{W}$ such that split$(B, \mathcal{W}, E) \neq \mathcal{W}$ the function findSplitter$(\mathcal{W}, E)$ returns one of these Bs, otherwise it returns the empty set. The algorithm is presented in figure 5.5, and it takes as input an initial partition $\mathcal{W}$ of some set U and a relation E over $U \times U$ and returns the coarsest stable refinement of $\mathcal{W}$.

FUNCTION refine($\mathcal{W}, E$)
  REPEAT
    S:= findSplitter($\mathcal{W}, E$)
    $\mathcal{W}$:= split($S, \mathcal{W}, E$)
  UNTIL $\mathcal{W}$ is stable
  RETURN $\mathcal{W}$

Figure 5.5: Partition refinement algorithm.

The time-complexity of this partition refinement algorithm is $O(n \cdot m)$ where n is the size of U and m is the size of E[8].

In our implementation a partition is represented by a list of lists where each list in the list of lists contains elements of U and represents a block of the partition. Relations are represented by a list of pairs of elements of U.

The refinement algorithm in figure 5.5 only refines a partition with respect to one type of relation/function. To solve an instance $U, \mathcal{W}, f_1, \ldots, f_k$ of the generalized partition refinement problem an iterative method is used where the function refine in figure 5.5 is applied once for each function f_i in each iteration until the partition $\mathcal{W}$ stabilizes.

5.4 Step 1 - Construction of the State Graphs

In this section our implementation of step 1 of the algorithm in figure 4.1 is described. In step 1 the saturated state graph (S, T) for a process to be tested for open bisimilarity is generated such that S and T are the minimal sets of constrained processes and transitions, respectively, that are closed under the three operations **Sat-trans**, **Sat-nonred**, and **Sat-bunch** described in section 4.8.

The saturated state graph for a constrained process is generated by maintaining two lists, call them the process list and the state list, respectively. The process list contains pairs of an integer and a constrained process and the state list contains pairs of an integer and a state. The integers of both lists represent state numbers of processes in the final state graph. Initially the process list contains the pair $(i, \langle P_{In}, D_{In}\rangle)$, where P_{In} is one of the processes to be checked for open bisimilarity with respect to the distinction D_{In} and i is the next unused state number, and the state list is empty. Then for each process $\langle P, D\rangle$ of the process list the following steps are used to create a state. The process $\langle P, D\rangle$ is then removed from the process list and the created state is added to the state list.

Find transitions: The transitions of $\langle P, D\rangle$ are found using the function $[\![\cdot]\!]_{CPTR}$ described in section 5.2.

Sat-trans: If one of the derivatives of $\langle P, D\rangle$ with respect to the transitions found in the previous step is not already present in either the process list or the state list the next unused state number is associated with it and it is added to the process list.

Sat-nonred: If there exist transitions $\langle P, D\rangle \stackrel{M,\alpha}{\longrightarrow} \langle P', D'\rangle$ and $\langle P, D\rangle \stackrel{N,\beta}{\longrightarrow} \langle P'', D''\rangle$ such that $M \not\lhd\rhd N$ and $\alpha = \beta\sigma_M$ then the process $CP'' \stackrel{def}{=} \langle P''\sigma_M, D^M_{P,\alpha} \cap \mathrm{fn}(P''\sigma_M)\rangle$ is given the next unused state number and is then added to the process list if it is not already present in either the process list or the state list. The state number associated with the process CP'' is added to the integer list associated with the transition $\langle P, D\rangle \stackrel{M,\alpha}{\longrightarrow} \langle P', D'\rangle$.

Sat-bunch: Processes are generated according to the operation **Sat-bunch** described in section 4.8. If such a process is not already present in either the process list or the state list the next unused state number is associated with it and it is added to the process list.

These steps are performed until the process list is empty and the saturated state graph (S, T) of $\langle P_{In}, D_{In}\rangle$ is represented by the final state list.

The algorithm for constructing the saturated state graph for a process P_{In} and a distinction D_{In} is given in figure 5.6. The argument $COUNTER$ is a number (≥ 0) that will be associated with $\langle P_{In}, D_{In}\rangle$. The function anElementOf(PL) returns (and removes) an element of PL, i.e. a pair of a state number and a constrained process. The function findProcessNumberInGraph(A, PL, G) searches for the constrained process A in PL and G. If A is present in either PL or G the number associated with the process is returned, otherwise -1 is returned indicating A is not present in PL or G.

FUNCTION constructSaturatedStateGraph($P_{In}, D_{In}, COUNTER$)
  $G := \emptyset$
  $PL := \{(COUNTER, \langle P_{In}, D_{In}\rangle)\}$
  WHILE $PL \neq \emptyset$ DO{
    $(i, CP) :=$ anElementOf(PL)
    $T := [\![CP]\!]_{CPTR}$
    $(TR, TR') := (\emptyset, \emptyset)$
    FOR EACH transition $(A \stackrel{(M,\alpha)}{\longrightarrow} A') \in T$ DO {
      $PNUM :=$ findProcessNumberInGraph(A', PL, G)
      IF ($PNUM = -1$) THEN
        $COUNTER := COUNTER + 1$
        $PL := PL \cup \{(COUNTER, A')\}$
        $TR := TR \cup \{(A \stackrel{(M,\alpha)}{\longrightarrow} A', COUNTER\}$
      ELSE $TR := TR \cup \{(A \stackrel{(M,\alpha)}{\longrightarrow} A', PNUM)\}$}(* END FOR *)

FOR EACH tuple $(\langle P, D\rangle \stackrel{(M,\alpha)}{\longrightarrow} \langle P', D'\rangle, l) \in TR$ DO {
  $AIL := \emptyset$
  $TRM := TR \setminus \{(\langle P, D\rangle \stackrel{(M,\alpha)}{\longrightarrow} \langle P', D'\rangle, l)\}$
  FOR EACH tuple $(\langle P, D\rangle \stackrel{(N,\beta)}{\longrightarrow} \langle P'', D''\rangle, m) \in TRM$ DO {
    IF ($M \not\lhd\rhd N$ AND $\alpha = \beta\sigma_M$) THEN
      $PNUM :=$ findProcessNumberInGraph$(\langle P''\sigma_M, D''^{M}_{P,\alpha} \cap \mathrm{fn}(P''\sigma_M)\rangle, PL, G)$
      IF ($PNUM = -1$) THEN
        $COUNTER := COUNTER + 1$
        $PL := PL \cup \{(COUNTER, \langle P''\sigma_M, D''^{M}_{P,\alpha} \cap \mathrm{fn}(P''\sigma_M)\rangle)\}$
        $AIL := AIL \cup \{COUNTER\}$
      ELSE $AIL := AIL \cup \{PNUM\}$}(* END FOR *)
  $TR' := TR' \cup \{(\langle P, D\rangle \stackrel{(M,\alpha)}{\longrightarrow} \langle P', D'\rangle, l, AIL)\}$}(* END FOR *)
$y := \min\{\mathcal{N} \setminus \mathrm{fn}(CP)\}$
FOR EACH tuple $(A \stackrel{(M,\alpha)}{\longrightarrow} A', j, AIL) \in TR'$ DO
  IF ($\mathrm{bn}(\alpha) = \{y\}$) THEN
    FOR EACH name $v < y$ DO
      $PNUM :=$ findProcessNumberInGraph$(A'\{v/y\ y/v\}, PL, G)$
      IF ($PNUM = -1$) THEN
        $COUNTER := COUNTER + 1$
        $PL := PL \cup \{(COUNTER, A'\{v/y\ y/v\})\}$
  $G := \{(i, (CP, TR'))\}$}(* END WHILE *)
RETURN G

Figure 5.6: Algorithm for constructing the saturated state graph of $\langle P_{In}, D_{In}\rangle$.

For finite control processes the time-complexity of constructing a saturated state graph is exponential with respect to the syntactic length of the process. This is due to the state space explosion problem[11].

5.5 Step 2 - Initializing Partition $\mathcal{W}$

In step 2 the initial partition $\mathcal{W}$ is constructed as a single block that consists of all the states of the saturated state graphs of the two processes to be checked for open bisimilarity. A partition is represented as a list of lists of integers. Therefore, $\mathcal{W}$ is a list containing one list

consisting of all the state numbers. The time-complexity of step 2 is linear with respect to the sum of the sizes of the state graphs.

5.6 Step 3 - Stabilizing Partition $\mathcal{W}$

In this section our implementation of the five sub steps of step 3 is described. Afterwards the time-complexity of step 3 is analyzed.

5.6.1 Step 3-1 - Compute the Non-Redundant Transitions

In this subsection our implementation of step 3-1 of the algorithm in figure 4.1 is described. In this step the non-redundant transitions of the constrained processes in the state graphs with respect to the partition $\mathcal{W}$ are computed. A transition $\langle P, D\rangle \stackrel{(M,\alpha)}{\longrightarrow} \langle P', D'\rangle$ is non-redundant if none of the numbers in the list of integers associated with this transition is in the same block of the partition $\mathcal{W}$ as the number associated with the process $\langle P', D'\rangle$. The algorithm for finding the non-redundant transitions of the state graphs is presented in figure 5.7. G is the union of the state graphs of the two processes being checked for open bisimilarity, AIL denotes a list of integers associated with a transition, $NRTL$ denotes a list of pairs of a non-redundant transition and an integer, and $GNRT$ is a new state graph containing only non-redundant transitions. The function inSameBlock$(i, J, \mathcal{W})$ returns true if the integer i and any integer $j \in J$ are in the same block of the partition $\mathcal{W}$ and false otherwise.

FUNCTION step3-1$(G, \mathcal{W})$
  $GNRT := \emptyset$
  FOR EACH state $(i, (A, TL)) \in G$ DO{
    $NRTL := \emptyset$
    FOR EACH tuple $(\langle P, D\rangle \stackrel{(M,\alpha)}{\longrightarrow} \langle P', D'\rangle, j, AIL) \in TL$ DO
      IF (not(inSameBlock$(j, AIL, \mathcal{W})$))
        THEN $NRTL := NRTL \cup \{(\langle P, D\rangle \stackrel{(M,\alpha)}{\longrightarrow} \langle P', D'\rangle, j)\}$
    $GNRT := GNRT \cup \{(i, (A, NRTL))\}$}(* END FOR *)
  RETURN $GNRT$

Figure 5.7: Algorithm for computing non-redundant transitions.

The time-complexity of the algorithm is $O(n^2 \cdot m^2)$, where n is the number of states in G and m is the maximum number of transitions of a constrained process in G.

5.6.2 Step 3-2 - Computation of Active Names

In this subsection we describe how we compute the active names for the constrained processes in the saturated state graphs with respect to a partition $\mathcal{W}$.

The active names with respect to $\mathcal{W}$ of the constrained processes in the saturated state graphs are computed using the state graph, $GNRT$, constructed in step 3-1, in the transitive closure algorithm in figure 5.8. The array $AN_{\mathcal{W}}$ will be used for holding the active names of the processes.

FUNCTION step3-2($GNRT$)
  $n :=$ size($GNRT$)
  $AN_{\mathcal{W}}[i] := \emptyset$ FOR ALL $i \in \{0, 1, \ldots, n-1\}$
  FOR $dummyCounter := 1$ TO n DO
    FOR EACH state $(i, (A, NRTL)) \in GNRT$ DO
      FOR EACH transition $(A \stackrel{(M,\alpha)}{\longrightarrow} A', l) \in NRTL$ DO
        $AN_{\mathcal{W}}[i] := AN_{\mathcal{W}}[i] \cup \mathrm{fn}(M, \alpha) \cup (AN_{\mathcal{W}}[l] \setminus \mathrm{bn}(\alpha))$
  RETURN $AN_{\mathcal{W}}$

Figure 5.8: Algorithm for computing active names.

After the algorithm has terminated $AN_{\mathcal{W}}[i]$ is the active names with respect to $\mathcal{W}$ of the constrained process with the number i associated with it. The time-complexity of the algorithm is $O(n^2 \cdot m)$, where m is the maximum number of non-redundant transitions with respect to $\mathcal{W}$ of a constrained process in one of the two state graphs. Alternatively we could exchange the fourth line of the algorithm with "repeat the following until $AN_{\mathcal{W}}[i]$ becomes stable for all i". This would reduce the number of iterations in the cases where the maximum length of a cycle in the $GNRT$ graph is less than the number of constrained processes in the two state graphs. However, if the time saved on the fewer iterations is less than the time needed to test for stability of the $AN_{\mathcal{W}}[i]$s nothing is won compared with the algorithm above.

5.6.3 Step 3-3 - Active Names Partition Refinement

The goal of step 3-3 is to refine the partition $\mathcal{W}$ using the active names of the constrained processes. From lemma 5 we know that two constrained processes A and B have the same active names with respect to a partition if $A \frown B$. So, if this is not the case the processes could be placed in different blocks of the partition.

Let $AN_{\mathcal{W}}$ be an array such that $AN_{\mathcal{W}}[i]$ is the set of active names of the process with the number i associated with it. The algorithm in figure 5.9 refines $\mathcal{W}$ to a partition $\mathcal{W}'$

in which two constrained processes are in the same block if and only if they have the same active names with respect to $\mathcal{W}$.

FUNCTION step3-3($AN_{\mathcal{W}}, \mathcal{W}$)
  $\mathcal{W}' := \emptyset$
  FOR EACH block $B = \{j_1, j_2, \ldots, j_k\} \in \mathcal{W}$ DO{
    $\mathcal{W}_B := \emptyset$
    FOR $i := 1$ TO k DO
      IF $(\exists X \in \mathcal{W}_B.(\exists y \in X.AN_{\mathcal{W}}[y] = AN_{\mathcal{W}}[j_i]))$
        THEN $\mathcal{W}_B := (\mathcal{W}_B \setminus \{X\}) \cup \{X \cup \{j_i\}\}$
        ELSE $\mathcal{W}_B := \mathcal{W}_B \cup \{\{j_i\}\}$
    $\mathcal{W}' := \mathcal{W}' \cup \mathcal{W}_B$}(* END FOR *)
  RETURN $\mathcal{W}'$

Figure 5.9: Algorithm for partition refinement using active names.

The time-complexity of the algorithm is $O(n^2)$.

Step 3-3 of the algorithm is not really necessary as it is seen from the definition of Φ in section 4.7. The reason it is in the algorithm is to reduce the running time.

5.6.4 3-4 - Computation of Normalized Transitions

In this step we use the non-redundant transitions from step 3-1 and the active names computed in step 3-2 to generate the normalized transitions of each constrained process in the two state graphs. The algorithm for computing normalized transitions is given in figure 5.10. It returns a state graph containing normalized transitions. The function findProcNum(A, $GNRT$) searches for the constrained process A in G and returns the number associated with this process.

FUNCTION step3-4($GNRT$, $AN_{\mathcal{W}}$)
  $GNT := \emptyset$
  FOR EACH $(i,(A,NRTL)) \in GNRT$ DO{
    $NTL := \emptyset$
    FOR EACH transition $(A \stackrel{(M,\alpha)}{\longrightarrow} A', j) \in NRTL$ DO
      IF ($\alpha = \tau$ OR $\alpha = \bar{a}b$ FOR SOME $a, b \in \mathcal{N}$)
      THEN $NTL := NTL \cup \{(A \stackrel{(M,\alpha)}{\longmapsto} A', j)\}$
      ELSE
        $y := \min\{\mathcal{N} \setminus \mathrm{fn}(A)\}$
        $v := \min\{\mathcal{N} \setminus AN_{\mathcal{W}}[i]\}$
        $A'' := A'\{v/y\ y/v\}$
        $NTL := NTL \cup \{(A \stackrel{(M,\alpha)}{\longmapsto} A'', \mathrm{findProcNum}(A'', GNRT))\}$
    $GNT := GNT \cup \{(i,(A,NTL))\}\}(*$ END FOR $*)$
  RETURN GNT

Figure 5.10: Algorithm for computing normalized transitions.

The time complexity of step 3-4 is $O(n^2 \cdot m)$, where n is the number of states in the saturated state graphs and m is the maximum number of non-redundant transitions for a process in the graphs.

5.6.5 Step 3-5 - Normalized Transition Partition Refinement

In this step we make a refinement of the partition $\mathcal{W}$ such that two constrained processes are in the same block of $\mathcal{W}$ if and only if they can make the same normalized transitions and reach constrained processes which again are in the same block of $\mathcal{W}$. To make this refinement we build a relation for each type of transition $\stackrel{(M,\alpha)}{\longmapsto}$. For each of these relations we use the refinement algorithm described in section 5.3 to refine partition $\mathcal{W}$. This can be done with the algorithm in figure 5.11.

FUNCTION step3-5($GNT, \mathcal{W}$)
  $RS := \emptyset$
  FOR EACH $(i, (A, NTL)) \in GNT$ DO
    FOR EACH transition $(A \stackrel{(M,\alpha)}{\longmapsto} A', j) \in NTL$ DO
      IF $(\exists((M, \alpha), R) \in RS)$
      THEN $RS := (RS \setminus \{((M, \alpha), R)\}) \cup \{((M, \alpha), R \cup \{(i, j)\})\}$
      ELSE $RS := RS \cup \{\{((M, \alpha), \{(i, j)\})\}\}$
  REPEAT
    $\mathcal{W}' := \mathcal{W}$
    FOR EACH $((M, \alpha), E) \in RS$ DO
      $\mathcal{W} :=$ refine$(\mathcal{W}, E)$
  UNTIL $\mathcal{W} = \mathcal{W}'$
  RETURN $\mathcal{W}$

Figure 5.11: Algorithm for partition refinement using normalized transitions.

The time-complexity of the construction of the set of relations RS is $O(n^2 \cdot m^2)$, where m is the maximum number of transitions of a constrained process in the state graphs and the time-complexity of the body of the repeat-until-loop is $O(\sum_{t=1}^{|RS|} |E_i|n)$, where the E_is are the second elements of the tuples in RS. Since at least one block is split in each iteration and since a block containing one element cannot be split the repeat-until-loop is run at most n times. Since $O(\sum_{t=1}^{|RS|} |E_i|n) \leq O(n^2 \cdot m)$ the time-complexity of the refinement of $\mathcal{W}$ is $O(n^3 \cdot m)$. Note that this is also the time-complexity for the total time spent in the repeat-until-loop for the whole bisimulation checking algorithm. The time-complexity of step 3-5 is $O(n^2 \cdot m^2 + n^3 \cdot m)$.

5.6.6 Time-Complexity of Step 3

Sub steps 3-1 to 3-5 are executed until the partition $\mathcal{W}$ stabilizes. Since at least one block is split in each iteration and since a block containing one element cannot be split the time-complexity of step 3 is $n(\sum_{i=1}^{4} C_i) + O(n \cdot n^2 \cdot m^2 + n^3 \cdot m) = O(n^3 \cdot m^2)$, where C_i is the time-complexity of step 3-i, m is the maximum number of transitions of a constrained process in the state graphs, and n is the number of states.

5.7 Step 4 - Result

In step 4 it is tested whether the two processes to be checked for open bisimilarity are in the same block of $\mathcal{W}$, which can be done in linear time with respect to the number of states in the graphs.

5.8 Main Function

In this section we present the main function openBisimCheck(P, Q, D) and give the time-complexity of it. The function openBisimCheck(P, Q, D), shown in figure 5.12, takes as input the two processes P and Q to be checked for open bisimilarity with respect to the distinction D and returns true if they are bisimilar and false otherwise.

FUNCTION openBisimCheck(P, Q, D)
  $G' :=$ constructSaturatedStateGraph$(P, D \cap \text{fn}(P), 0)$
  $G'' :=$ constructSaturatedStateGraph$(Q, D \cap \text{fn}(Q), \text{size}(G'))$
  $G := G' \cup G''$
  $\mathcal{W} := \{\{0, \ldots, \text{size}(G) - 1\}\}$
  REPEAT
    $\mathcal{W}' := \mathcal{W}$
    $GNRT :=$ step3-1$(G, \mathcal{W})$
    $AN :=$ step3-2$(GNRT)$
    $\mathcal{W} :=$ step3-3$(AN, \mathcal{W})$
    $GNT :=$ step3-4$(GNRT, AN)$
    $\mathcal{W} :=$ step3-5$(GNT, \mathcal{W})$
  UNTIL $\mathcal{W} = \mathcal{W}'$
  RETURN (inSameBlock$(0, \{\text{size}(G) - 1\}, \mathcal{W})$)

Figure 5.12: Algorithm for checking for open bisimilarity.

The time-complexity of openBisimCheck(P, Q, D) for finite control processes is exponential with respect to the syntactic length of the processes P and Q.

Chapter 6

Optimization

In this section we describe how the algorithm and the implementation of the open bisimulation checker can be optimized with respect to run time. As it is seen in the previous sections the time-complexity of steps 2 and 3 are functions of the sizes of the saturated state graphs generated in step 1. So, an obvious optimization would be to reduce the size of the graphs. This can be done by analyzing the states added using the **Sat-bunch** and **Sat-nonred** operations in step 1. Finally, we will describe how some of the sub steps of step 3 can be optimized. Some of these steps can be optimized by using arrays instead of lists to represent the state graphs. Therefore, in step 3-1 of the algorithm, we convert our graphs from lists of states to one array containing the states of both state graphs. The states are placed such that the state with number i associated with it is placed in the ith position of the array (the pseudo code for the optimized version of step 3-1 can be seen in appendix A). This takes time linear to the sum of the sizes of the graphs.

6.1 Reducing the Sizes of the State Graphs

The sizes of the state graphs can be reduced by changing the **Sat-bunch** and **Sat-nonred** operations as described in the following subsections.

6.1.1 Optimizing the Sat-bunch Operation

In **Sat-bunch** states are added to the state graphs to make sure that all the processes needed for the normalized transitions computed in step 3-4 are present in the state graphs. For each transition $A \xrightarrow{(M,\alpha)} A'$ with $\mathrm{bn}(\alpha) = \{y\}$, where $y = \min\{\mathcal{N} \setminus \mathrm{fn}(A)\}$, the state $A'\{x/y\ y/x\}$ is added to S for all $x < y$. Since $\min\{\mathcal{N} \setminus \mathrm{an}_{\mathcal{W}}(A)\} \leq y$ for every $\mathcal{W}$, this ensures that for each partition $\mathcal{W}$ the state $A'\{v/y\ y/v\}$, where $v = \min\{\mathcal{N} \setminus \mathrm{an}_{\mathcal{W}}(A)\}$, is in S as needed for the normalization of $A \xrightarrow{(M,\alpha)} A' \in \mathrm{nr}(\mathcal{W})$. To see why some of the constrained processes added in the **Sat-bunch** operation are extraneous consider as an example the

π-process $P \stackrel{def}{=} a(g).\bar{c}d.\bar{e}f.\bar{b}g.\mathbf{0}$, where $a < b < c < d < e < f < g$. When P is checked for open bisimilarity with another process with respect to the empty distinction, initially S contains the constrained process $A = \langle P, \emptyset \rangle$ and by the **Sat-trans** operation the transition $t = A \stackrel{(\emptyset, a(g))}{\longrightarrow} A'$, where $A' = \langle \bar{c}d.\bar{e}f.\bar{b}g.\mathbf{0}, \emptyset \rangle$, is added to T. Since $\mathrm{fn}(A) = \{a, b, c, d, e, f\}$ and $\min\{\mathcal{N} \backslash \mathrm{fn}(A)\} = g$ the states $A'\{a/g\ g/a\}, \ldots, A'\{f/g\ g/f\}$ are added to S due to the **Sat-bunch** operation. It is easily seen that these states will never be used in the normalization of t since all the free names of A are active for every partition $\mathcal{W}$. We could therefore omit these states in the generation of the state graph for A.

As seen from the example the number of states added by the **Sat-bunch** operation can be reduced if some of the names of a process A with a transition $A \stackrel{(M,\alpha)}{\longrightarrow} A'$ with $\mathrm{bn}(\alpha) = \{y\}$, where
$y = \min\{\mathcal{N} \setminus \mathrm{fn}(A)\}$, are known to be active with respect to all partitions and are smaller than y. It is worth noting that by adding one state less to a saturated state graph the size of the graph is, in most cases, reduced by more than one, since the state that would have been added would have implied that more states would have been added to the graph which would have implied that even more states would have been added to the graph and so on.

As mentioned in section 4.6 the computation of active names of a process is as difficult as checking for open bisimilarity, so in some cases it is not possible to leave out all unnecessary states in the generation of a state graph. Therefore we find some of the states that can be omitted by using approximations of the names of a process that are active with respect to all partitions. We let the approximation of the active names of the process $A = \langle P, D \rangle$, $\mathrm{ApproxAN}(A)$, be the set $\mathrm{AN}(P, \emptyset, \emptyset)$ where $\mathrm{AN}(P, N, B)$ is the least function that satisfies the following. N is a set of new names and B is a set of bound but not new names.

$$\mathrm{AN}(\mathbf{0}, N, B) = \emptyset$$

$$\mathrm{AN}(\tau.P, N, B) = \mathrm{AN}(P, N, B)$$

$$\mathrm{AN}(a(b).P, N, B) = \begin{cases} \emptyset & \text{if } a \in N \\ (\{a\} \setminus B) \cup \mathrm{AN}(P, N \setminus \{b\}, B \cup \{b\}) & \text{otherwise} \end{cases}$$

$$\mathrm{AN}(\bar{a}b.P, N, B) = \begin{cases} \emptyset & \text{if } a \in N \\ (\{a, b\} \setminus (N \cup B)) \cup \mathrm{AN}(P, N \setminus \{b\}, B \cup (N \cap \{b\})) & \text{otherwise} \end{cases}$$

$$\mathrm{AN}(\bar{a}(b).P, N, B) = \begin{cases} \emptyset & \text{if } a \in N \\ (\{a\} \setminus B) \cup \mathrm{AN}(P, N \setminus \{b\}, B \cup \{b\}) & \text{otherwise} \end{cases}$$

$$\mathrm{AN}([a = b]P, N, B) = \emptyset$$

$$\mathrm{AN}(P|Q, N, B) = \mathrm{AN}(P, N, B) \cup \mathrm{AN}(Q, N, B)$$

$$\mathrm{AN}(P + Q, N, B) = \mathrm{AN}(P, N, B) \cup \mathrm{AN}(Q, N, B)$$

$$\mathrm{AN}((\nu\ a)P, N, B) = \mathrm{AN}(P, N \cup \{a\}, B \setminus \{a\})$$

$$\mathrm{AN}(K\langle \tilde{a} \rangle, N, B) = \mathrm{AN}(P\{\tilde{a}/\tilde{b}\}, N, B) \text{ where } K \stackrel{def}{=} (\tilde{b})P$$

The approximation $\mathrm{ApproxAN}(A)$ of active names in A can be computed efficiently by syntactic analysis of A. Since transitions with trivially true conditions are always non-redundant it is easily seen that $\mathrm{ApproxAN}(A) \subseteq \mathrm{an}_{\mathcal{W}_\sim}(A)$ for all $\mathcal{W}$. Therefore, an optimized version of the **Sat-bunch** operation can be defined as follows.

Sat-bunch-approx. If $A \stackrel{(M,\alpha)}{\longrightarrow} A' \in T$ with $\mathrm{bn}(\alpha) = \{y\}$, where $y \stackrel{def}{=} \min\{\mathcal{N} \setminus \mathrm{fn}(A)\}$, then $A'\{v/y\ y/v\} \in S$ for all $v < y$ such that $v \notin \mathrm{ApproxAN}(A)$.

Using the **Sat-bunch-approx** operation can lead to cases with significant reductions of the sizes of the state graphs. Consider the following examples of applications of the algorithms. We assume that $a < b < c < d < e < f < g < h < i$.

Applying the algorithm with the **Sat-bunch** operation to the following two, clearly not bisimilar, processes

$$P_1 \stackrel{def}{=} \overline{a}(c).e(f).\mathbf{0} \mid \overline{a}(b).c(d).\mathbf{0} \mid a(b).\overline{c}b.\mathbf{0}$$
$$P_2 \stackrel{def}{=} \overline{a}(g).c(f).\mathbf{0} \mid \overline{a}(i).c(h).\mathbf{0} \mid \overline{a}(c).e(f).\mathbf{0} \mid \overline{a}(b).c(d).\mathbf{0}$$

with the empty distinction, $D = \emptyset$, gives state graphs with 746 states total. Applying the algorithm with the **Sat-bunch-approx** operation gives state graphs with 396 states total. Thus, in this case, running the algorithm with the **Sat-bunch-approx** operation reduces the sizes of the graphs by a factor 1.88.

In some cases only a small reduction of the sizes of the graphs can be achieved by running the algorithm with the **Sat-bunch-approx** operation. Applying the algorithm with the **Sat-bunch** operation to the following two processes

$$P_3 = K_2\langle(c,e)\rangle + [a=e]\overline{c}(b).a(d).K_1\langle(a,b,d)\rangle$$
$$P_4 = \overline{c}(b).e(d).K_2\langle(b,d)\rangle$$

where

$$K_1 \stackrel{def}{=} (a,c,e)P_1 \text{ and } K_2 \stackrel{def}{=} (c,e)P_2$$

with the empty distinction, $D = \emptyset$, gives state graphs with 44 states total. Applying the algorithm with the **Sat-bunch-approx** operation gives state graphs with 42 states total. Thus, in this case, running the algorithm with the **Sat-bunch-approx** operation only reduces the sizes of the graphs by a factor 1.05. Through numerous test runs it has become apparent that the time spent on approximating the active names is time well spent.

6.1.2 Optimizing the Sat-nonred Operation

As seen in the previous subsection the size of the state graph of a process can be reduced by analyzing the states added using the **Sat-bunch** operation. In this subsection we will describe how the sizes of the state graphs can be reduced further by analyzing the states added using the **Sat-nonred** operation. The **Sat-nonred** operation says that for each pair of transitions $\langle P, D\rangle \stackrel{(M,\alpha)}{\longrightarrow} \langle P', D'\rangle$ and $\langle P, D\rangle \stackrel{(N,\beta)}{\longrightarrow} \langle P'', D''\rangle$ with $M \not\lhd\rhd N$ and $\alpha = \beta\sigma_M$ the process $B \stackrel{def}{=} \langle P''\sigma_M, D^M_{P,\alpha} \cap \mathrm{fn}(P''\sigma_M)\rangle$ should be added to the state graphs. If it can be proven that $\langle P', D'\rangle$ and B will be placed in different blocks of the partition $\mathcal{W}$ after the first iteration in step 3 it is not necessary to add B, since it would not have any influence on

the non-redundancy of the transition $\langle P, D\rangle \stackrel{(M,\alpha)}{\longrightarrow} \langle P', D'\rangle$ with respect to the partition $\mathcal{W}$ after the first iteration in step 3. Proving whether two processes will be placed in different blocks of the partition $\mathcal{W}$ after the first iteration in step 3 is not trivial without actually performing the first iteration in step 3. So, as with the **Sat-bunch** operation we will only find some of the states that can be omitted. One way to prove that two processes $\langle P', D'\rangle$ and B will be placed in different blocks is to examine the initial transitions of $\langle P', D'\rangle$ and B. B can be omitted if it does not have similar transitions to $\langle P', D'\rangle$. It should be noted that the computation of the non-redundant transitions in step 3-1 should be changed for the first iteration in step 3, since the transition $\langle P, D\rangle \stackrel{(M,\alpha)}{\longrightarrow} \langle P', D'\rangle$ is non-redundant with respect to the initial partition if and only if there does not exist a transition $\langle P, D\rangle \stackrel{(N,\beta)}{\longrightarrow} \langle P'', D''\rangle$ such that $M \not\triangleleft\triangleright N$, $\alpha = \beta\sigma_M$, and $\langle P', D'\rangle$ and $\langle P''\sigma_M, D^M_{P,\alpha} \cap \mathrm{fn}(P''\sigma_M)\rangle$ are in the same block of the initial partition. The last condition, $\langle P', D'\rangle$ and $\langle P''\sigma_M, D^M_{P,\alpha} \cap \mathrm{fn}(P''\sigma_M)\rangle$ are in the same block of the initial partition, is trivially true with the current version of the **Sat-nonred** operation, but not always if some of the states that should have been added due to this operation are omitted.

6.2 Optimizing the Computation of Non-Redundant Transitions

The running time of the computation of the non-redundant transitions can be reduced by using lemma 2. This lemma states that if a transition is non-redundant for a relation then it is also non-redundant for a subset of the relation. So, if a transition is computed as being non-redundant in iteration i in step 3, it will also be non-redundant in iteration $i+1$. Therefore, a simple optimization would be to mark a transition when it is first computed as being non-redundant in step 3-1 and not checking whether the marked transitions are non-redundant.

6.3 Optimizing the Computation of Active Names

As mentioned in subsection 5.6.2 the time-complexity of the algorithm for computing active names is $O(n^2 \cdot m)$, where m is the maximum number of non-redundant transitions with respect to any partition of a constrained process in the state graphs. Experiments with the implementation have shown that the total time spent in step 3-2 can be reduced tremendously by using the result of lemma 3 and changing the outer for-loop with a repeat until construction which stops when all the $\mathrm{AN}_{\mathcal{W}}[i]$s become stable.

Since the partition $\mathcal{W}$ becomes finer for each iteration of step 3 it follows from lemma 3 that the set of active names for a constrained process in iteration j is contained in the set of active names for the same process in iteration $j+1$. So, the active names computed in one iteration could be reused as a starting point in the next iteration and thereby saving some time, because the $\mathrm{AN}_{\mathcal{W}}[i]$s stabilize more quickly. With these considerations in mind a new algorithm for computing active names was developed and is presented in figure 6.1.

```
FUNCTION optimizedStep3-2(GNRT, AN_W)
  REPEAT
    stable := true
    FOR i = 0 TO size(GNRT) − 1 DO
      (A, NRTL) := GNRT[i]
      AN := ∅
      FOR EACH transition (A −(M,α)→ A', l) ∈ NRTL DO
        AN := AN ∪ fn(M, α) ∪ (AN_W[l] \ bn(α))
      stable := stable ∧ (AN = AN_W[i])
      AN_W[i] := AN
  UNTIL stable = true
  RETURN AN_W
```

Figure 6.1: Optimized algorithm for computing active names.

The time-complexity for computing active names is not reduced with the new algorithm and in the worst case it runs even slower than the old one because of the comparisons between sets of active names. The computation of active names could also have been optimized by finding the sizes of the saturated state graphs for the processes being checked for open bisimilarity and changing the outer for-loop of the first algorithm to loop as many times as the maximum size.

6.4 Optimizing the Computation of the Normalized Transitions

The running time of the computation of the normalized transitions can be reduced if the data structures are changed such that a list of numbers is associated with each transition. For a transition t this list should contain the state numbers for those states added to the saturated state graph by the **Sat-trans** and the **Sat-bunch** operations due to t. When the non-redundant transition $t = A \xrightarrow{(M,\alpha)} A'$ is normalized to $A \xrightarrow{(M,\alpha')} A''$ the state number of A'' can be found quickly by examining the states in the array at the positions corresponding to the numbers in the list of numbers associated with t. The pseudo code for the optimized version of the algorithm for computing the normalized transitions can be found in appendix A.

6.5 A Faster Partition Refinement Algorithm

Our first implementation of the algorithm for open bisimulation checking included an implementation of the partition refinement algorithm described in section 5.3. This algorithm is very easy to implement but unfortunately it has the time-complexity $O(n \cdot m)$, where n is the size of the set to be partitioned and m is the size of the relation on which the partitioning is based. Paige and Tarjan[8] have developed a more efficient algorithm for solving the generalized partition refinement problem. The higher efficiency of this algorithm is obtained through a better strategy for finding the sets used as splitters. The algorithm is presented below. The function split works the same way as described in section 5.3.

FUNCTION optimizedRefine($\mathcal{W}, E$)
$U := \bigcup \mathcal{W}$
$\mathcal{W}' := \emptyset$
FOR EACH block $B \in \mathcal{W}$ DO
    $B' := B \cup E^{-1}(U)$
    $B'' := B \setminus E^{-1}(U)$
    $\mathcal{W}' := \mathcal{W}' \cup \{B', B''\}$
$X := \{U\}$
REPEAT
    Find a block $S \in X$ where $S \notin \mathcal{W}'$
    Find a block $B \subseteq \mathcal{W}'$ where $|B| \leq \frac{|S|}{2}$
    $X := (X \setminus \{S\}) \cup \{B\} \cup \{S \setminus B\}$
    $\mathcal{W}' := \text{split}(S \setminus \{B\}, \text{split}(B, \mathcal{W}'))$
UNTIL $X = \mathcal{W}'$
RETURN $\mathcal{W}'$

Figure 6.2: Optimized partition refinement algorithm.

The algorithm can be implemented such that the time complexity is $O(m \cdot \log(n) + n)$. To obtain this, several efficient data structures are required. A description of these and an outline of an implementation can be found in [8].

We have implemented the algorithm presented above and replaced the implementation of the first partition refinement algorithm with it. The time complexity of step 3-5 has not changed but performance tests have shown that in many cases the running time is reduced significantly.

6.6 Heuristics

In this section we present some heuristics that may reduce the running time of the open bisimilarity checking algorithm.

Since the partition $\mathcal{W}$ becomes finer for each iteration of step 3 and the two processes being checked are open bisimilar if and only if they are in the same block of the final partition the algorithm can be changed such that it terminates and outputs false as soon as the processes being checked are put in two different blocks. Checking whether the processes are in the same block could be done after sub step 3-3 or sub step 3-5 in step 3. If the data structures are changed such that each element in a block points to the block containing it, the checking can be done in constant time. If the algorithm is changed such that the checking is performed it will terminate much faster in the cases where obviously non-open bisimilar processes like $\bar{a}(b).EvilP$ and $\mathbf{0}$ are checked, since the processes are put into different blocks after the first iteration in step 3. In the case where two open bisimilar processes are checked nothing is won, but the time wasted on checking whether they are put in different blocks is negligible compared to the overall runtime.

As already seen the number of states in the graph depends heavily on the number of states added by the **Sat-bunch** operation. In some cases the sizes of the saturated state graphs can be reduced by reordering the names. Consider the following example. Let P be the π-process defined by $P \stackrel{def}{=} a(g).\bar{c}d.\bar{e}f.\bar{b}g.\mathbf{0}$. If a, b, c, d, e, f, g, and h are the first eight names in $\mathcal{N}$, and $a < b < c < d < e < f < g < h$, then the size of the saturated state graph for $\langle P, \emptyset \rangle$ is 23. By reordering the names such that $h < a < b < c < d < e < f < g$ the size of the saturated state graph is reduced to 5 states. So, by a clever reordering of the names the size of the graph can be reduced significantly. In the example given the perfect reordering was easy to find, but this is not always the case.

6.7 Remarks on the Optimized Algorithm

The optimized algorithm is the same as the first algorithm where the steps are updated according to some of the optimizations suggested in this chapter. The optimized algorithm can be found in appendix A. It is worth noting that the time-complexity is not reduced in the optimized version of the algorithm but performance tests have shown that the running time is reduced significantly in many cases. Results of performance tests of the implementation of the first and the optimized algorithm and comparisons of these can be found in appendix B.

Chapter 7

Conclusion

We have implemented the algorithm for open bisimulation checking in the π-calculus proposed by [10]. Due to the state space explosion problem the time-complexity of the algorithm is exponential with respect to the depth of the processes being checked for open bisimilarity. This is only the case when the algorithm is applied to finite control processes. For other processes the algorithm may not even terminate. However, the algorithm terminates in a reasonable amount of time for many interesting processes.

We have optimized the algorithm for open bisimulation checking. By approximating the active names of processes we can identify processes that do not have to be present in the state graphs and thereby, in many cases, reduce the size of the state graphs and the overall running time of the algorithm. We have implemented this optimized algorithm and other optimizations have been applied to this implementation to bring the running time further down. The time-complexity of this implementation is not better than the implementation of the first version of the algorithm. However, performance tests have shown that in many cases a considerable reduction of the running time can be achieved.

Bibliography

[1] Boreale, Michele & De Nicola, Rocco. *A Symbolic Semantics for the π-Calculus*. Journal of Information and Computation, 126(1):34-52, 1996.

[2] Dam, Mads. *Model Checking Mobile Processes*. Journal of Information and Computation, 129(1):35-51, 1996.

[3] Kanellakis, Paris C. & Smolka, Scott A. *CCS Expressions, Finite State Processes, and Three Problems of Equivalence*. Journal of Information and Computation, 86:43-68, 1990.

[4] Milner, Robin. *Communication and Concurrency*. Prentice Hall International, Englewood Cliffs, 1989. ISBN: 0-13-115007-3.

[5] Milner, Robin & Parrow, Joachim & Walker, David. *A Calculus of Mobile Processes, Part I/II*. Journal of Information and Computation, 100:1-77, September 1992.

[6] Moller, Faron. *Edinburgh Concurrency Workbench user manual (Version 7.1)*. Updated since CWB v7.0 by Perdita Stevens. Laboratory for Foundations of Computer Science, University of Edinburgh, July 1999.

[7] Moller, Faron & Victor, Björn. *The Mobility Workbench — A Tool for the π-Calculus*. In proceedings of CAV'94 volume 818 of Lecture Notes in Computer Science, pp. 428-440. Springer Verlag, 1994.

[8] Paige, Robert & Tarjan, Robert E. *Three Partition Refinement Algorithms*. SIAM Journal on Computing, 16(6):973-989, December 1987.

[9] Parrow, Joachim. *An Introduction to the π-Calculus*. Chapter of *Handbook of Process Algebra*, ed. Jan A. Bergstra, Alban Ponse, and Scott A. Smolka.

[10] Pistore, Marco & Sangiorgi, Davide. *A Partition Refinement Algorithm for the π-Calculus*. In proceedings of CAV'96 volume 1102 of Lecture Notes in Computer Science. Springer Verlag, 1996. Complete version available from file://ftp.di.unipi.it/pub/Papers/pistore/cav96long.ps.gz .

[11] Rabinovich, Alexander. *Checking Equivalences Between Concurrent Systems of Finite Agents.* In proceedings of ICALP '92 volume 623 of Lecture Notes in Computer Science, pp. 696 - 707, Springer Verlag, 1992.

[12] Tarski, Alfred. *A Lattice-theoretical Fixpoint Theorem and Its Applications.* Pacific Journal of Mathematics, 5:285-309, 1955.

Appendiks A

The Optimized Algorithm

In this appendix we present the pseudo code for the optimized algorithm for checking for open bisimilarity between two processes P and Q with the distinction D as the function openBisimCheck(P, Q, D). The algorithm makes use of some functions for searching in the state graph. The function findProcNumFast(CP,$GNRT$,IL) used in step 3-4 searches for the constrained process CP in the array $GNRT$ at the positions corresponding to the numbers contained in the list of integers IL and returns the position of CP in the array $GNRT$. The functions anElementOf(PL), findProcessNumberInGraph(A, PL, G), ApproxAN(CP), inSameBlock$(j, AIL, \mathcal{W})$, and optimizedRefine$(\mathcal{W}, E)$ are described in chapters 5 and 6.

FUNCTION constructStateGraph$(P_{In}, D_{In}, COUNTER)$
  $G := \emptyset$
  $PL := \{(COUNTER, \langle P_{In}, D_{In}\rangle)\}$
  WHILE $PL \neq \emptyset$ DO{
    $(i, CP) :=$ anElementOf(PL)
    $T := [\![CP]\!]_{CPTR}$
    $(TR, TR', TRAN) := (\emptyset, \emptyset, \emptyset)$
    FOR EACH transition $(A \stackrel{(M,\alpha)}{\longrightarrow} A') \in T$ DO {
      $PNUM :=$ findProcessNumberInGraph(A', PL, G)
      IF $(PNUM = -1)$ THEN
        $COUNTER := COUNTER + 1$
        $PL := PL \cup \{(COUNTER, A')\}$
        $TR := TR \cup \{(A \stackrel{(M,\alpha)}{\longrightarrow} A', COUNTER)\}$
      ELSE $TR := TR \cup \{(A \stackrel{(M,\alpha)}{\longrightarrow} A', PNUM)\}\}$(* END FOR *)

FOR EACH tuple $(\langle P, D\rangle \stackrel{(M,\alpha)}{\longrightarrow} \langle P', D'\rangle, l) \in TR$ DO {
$AIL := \emptyset$
$TRM := TR \setminus \{(\langle P, D\rangle \stackrel{(M,\alpha)}{\longrightarrow} \langle P', D'\rangle, l)\}$
FOR EACH tuple $(\langle P, D\rangle \stackrel{(N,\beta)}{\longrightarrow} \langle P'', D''\rangle, m) \in TRM$ DO {
IF ($M \not\lhd\rhd N$ AND $\alpha = \beta\sigma_M$) THEN
$PNUM :=$ findProcessNumberInGraph$(\langle P''\sigma_M, D^M_{P,\alpha} \cap \mathrm{fn}(P''\sigma_M)\rangle, PL, G)$
IF ($PNUM = -1$) THEN
$COUNTER := COUNTER + 1$
$PL := PL \cup \{(COUNTER, \langle P''\sigma_M, D^M_{P,\alpha} \cap \mathrm{fn}(P''\sigma_M)\rangle)\}$
$AIL := AIL \cup \{COUNTER\}$
ELSE $AIL := AIL \cup \{PNUM\}$}(* END FOR *)
$TR' := TR' \cup \{(\langle P, D\rangle \stackrel{(M,\alpha)}{\longrightarrow} \langle P', D'\rangle, l, AIL)\}$}(* END FOR *)
$AAN :=$ ApproxAN(CP)
$y := \min\{\mathcal{N} \setminus \mathrm{fn}(CP)\}$
FOR EACH tuple $(A \stackrel{(M,\alpha)}{\longrightarrow} A', j, AIL) \in TR'$ DO
$IL := \{j\}$
IF ($\mathrm{bn}(\alpha) = \{y\}$) THEN
FOR EACH name $v < y$ DO
IF ($v \notin AAN$) THEN
$PNUM :=$ findProcessNumberInGraph$(A'\{v/y\ y/v\}, PL, G)$
IF ($PNUM = -1$) THEN
$COUNTER := COUNTER + 1$
$PL := PL \cup \{(COUNTER, A'\{v/y\ y/v\})\}$
$IL := IL \cup \{COUNTER\}$
ELSE $IL := IL \cup \{PNUM\}$
$TRAN := TRAN \cup \{(A \stackrel{(M,\alpha)}{\longrightarrow} A', j, APL, IL)\}$
$G := \{(i, (CP, TRAN))\}$}(* END WHILE *)
RETURN G

FUNCTION step2(G)
$\mathcal{W} := \{\{0, \ldots, \mathrm{size}(G) - 1\}\}$
RETURN $\mathcal{W}$

FUNCTION optimizedStep3-1($G, \mathcal{W}$)
  $GNRT[i]$:= undefined state FOR ALL $i \in \{0, \ldots, \text{size}(G) - 1\}$
  FOR EACH state $(i, (A, TL)) \in G$ DO{
    $NRTL := \emptyset$
    FOR EACH tuple $(\langle P, D\rangle \stackrel{(M,\alpha)}{\longrightarrow} \langle P', D'\rangle, j, AIL) \in TL$ DO{
      IF (not(inSameBlock($j, AIL, \mathcal{W}$)))
        THEN $NRTL := NRTL \cup \{(\langle P, D\rangle \stackrel{(M,\alpha)}{\longrightarrow} \langle P', D'\rangle, j)\}$}(* END FOR *)
    $GNRT[i] := \{(A, NRTL)\}$}(* END FOR *)
  RETURN $GNRT$

FUNCTION optimizedStep3-2($GNRT, AN_{\mathcal{W}}$)
  REPEAT
    $stable := true$
    FOR $i = 0$ TO size($GNRT$) $- 1$ DO
      $(A, NRTL) := GNRT[i]$
      $AN := \emptyset$
      FOR EACH transition $(A \stackrel{(M,\alpha)}{\longrightarrow} A', l) \in NRTL$ DO
        $AN := AN \cup \text{fn}(M, \alpha) \cup (AN_{\mathcal{W}}[l] \setminus \text{bn}(\alpha))$
      $stable := stable \wedge (AN = AN_{\mathcal{W}}[i])$
      $AN_{\mathcal{W}}[i] := AN$
  UNTIL $stable = true$
  RETURN $AN_{\mathcal{W}}$

FUNCTION optimizedStep3-3($AN_{\mathcal{W}}, \mathcal{W}$)
  $\mathcal{W}' := \emptyset$
  FOR EACH block $B = \{j_1, j_2, \ldots, j_k\} \in \mathcal{W}$ DO{
    $\mathcal{W}_B := \emptyset$
    FOR $i := 1$ TO k DO
      IF $(\exists X \in \mathcal{W}_B.(\exists y \in X.AN_{\mathcal{W}}[y] = AN_{\mathcal{W}}[j_i]))$
        THEN $\mathcal{W}_B := (\mathcal{W}_B \setminus \{X\}) \cup \{X \cup \{j_i\}\}$
        ELSE $\mathcal{W}_B := \mathcal{W}_B \cup \{\{j_i\}\}$
    $\mathcal{W}' := \mathcal{W}' \cup \mathcal{W}_B$}(* END FOR *)
  RETURN $\mathcal{W}'$

FUNCTION optimizedStep3-4($GNRT$, $AN_{\mathcal{W}}$)
  $GNT[i]$:= undefined state FOR ALL $i \in \{0, \ldots, \text{size}(GNRT) - 1\}$
  FOR $i = 0$ TO size($GNRT$) $- 1$ DO{
    $(A, NRTL) := GNRT[i]$
    $NTL := \emptyset$
    FOR EACH transition $(A \stackrel{(M,\alpha)}{\longrightarrow} A', j, IL) \in NRTL$ DO
      IF ($\alpha = \tau$ OR $\alpha = \bar{a}b$ FOR SOME $a, b \in \mathcal{N}$)
      THEN $NTL := NTL \cup \{(A \stackrel{(M,\alpha)}{\longmapsto} A', j)\}$
      ELSE
        $y := \min\{\mathcal{N} \setminus \text{fn}(A)\}$
        $v := \min\{\mathcal{N} \setminus AN_{\mathcal{W}}[i]\}$
        $A'' := A'\{v/y\ y/v\}$
        $NTL := NTL \cup \{(A \stackrel{(M,\alpha)}{\longmapsto} A'', \text{findProcNumFast}(A'', GNRT, IL))\}$
    $GNT[i] := \{(A, NTL)\}$}(* END FOR *)
  RETURN GNT

FUNCTION optimizedStep3-5($GNT, \mathcal{W}$)
  $RS := \emptyset$
  FOR $i = 0$ TO size($GNT - 1$) DO{
    $(A, NTL) := GNT[i]$
    FOR EACH transition $(A \stackrel{(M,\alpha)}{\longmapsto} A', j) \in NTL$ DO
      IF $(\exists((M, \alpha), R) \in RS)$
      THEN $RS := (RS \setminus \{((M, \alpha), R)\}) \cup \{((M, \alpha), R \cup \{(i, j)\})\}$
      ELSE $RS := RS \cup \{\{((M, \alpha), \{(i, j)\})\}\}$}(* END FOR *)
  REPEAT
    $\mathcal{W}' := \mathcal{W}$
    FOR EACH $((M, \alpha), E) \in RS$ DO
      $\mathcal{W} := \text{optimizedRefine}(\mathcal{W}, E)$
  UNTIL$\mathcal{W}' = \mathcal{W}$
  RETURN $\mathcal{W}$

```
FUNCTION optimizedStep3And4(i_P, i_Q, G, W)
  AN[i] := ∅ FOR ALL i ∈ {0, ..., size(G) − 1}
  REPEAT
    W' := W
    GNRT := optimizedStep3-1(G, W)
    AN := optimizedStep3-2(GNRT, AN)
    W := optimizedStep3-3(AN, W)
    GNT := optimizedStep3-4(GNRT, AN)
    W := optimizedStep3-5(GNT, W)
    IF (not(inSameBlock(i_P, {i_Q}, W))) THEN
      RETURN false
  UNTIL W = W'
  RETURN (inSameBlock(i_P, {i_Q}, W))


FUNCTION openBisimCheck(P, Q, D)
  G' := constructStateGraph(P, D ∩ fn(P), 0)
  G'' := constructStateGraph(Q, D ∩ fn(Q), size(G'))
  G := G' ∪ G''
  W := step2(G)
  RETURN optimizedStep3And4(0, size(G'), G, W)
```

Appendiks B

Performance Tests

In this appendix we present some results of performance tests of the implementation of the first version of the algorithm and the optimized version, and compare these results. The performance tests are run on an AMD Athlon 700 MHz processor. Application of the first algorithm and the optimized algorithm to two processes, P_1 and P_2, and a distinction, D, is indicated by openBisimCheck$_{First}(P_1, P_2, D)$ and openBisimCheck$_{Opt}(P_1, P_2, D)$, respectively. We assume $a < b < c < d < e < f < g < h < i$.

B.1 π-Processes Used in Performance Tests

The following π-processes and distinctions are used in the performance tests of the algorithms.

Processes:

$$P_1 \stackrel{def}{=} K_2\langle(c,e)\rangle + [a=e]\overline{c}b.a(d).K_1\langle(a,b,d)\rangle$$

$$P_2 \stackrel{def}{=} \overline{c}(b).e(d).K_2\langle(b,d)\rangle$$

$$P_3 \stackrel{def}{=} (K_2\langle(c,e)\rangle + [a=e]\overline{c}b.a(d).K_1\langle(a,b,d)\rangle) \mid \overline{a}b.\overline{c}d.\mathbf{0}$$

$$P_4 \stackrel{def}{=} \overline{c}(b).e(d).K_2\langle(b,d)\rangle \mid \overline{a}b.\overline{c}d.\mathbf{0}$$

$$P_5 \stackrel{def}{=} \overline{a}(c).e(f).\mathbf{0} \mid \overline{a}(b).c(d).\mathbf{0} \mid a(b).\overline{c}b.\mathbf{0}$$

$$P_6 \stackrel{def}{=} \overline{a}(g).c(f).\mathbf{0} \mid \overline{a}(i).c(h).\mathbf{0} \mid \overline{a}(c).e(f).\mathbf{0} \mid \overline{a}(b).c(d).\mathbf{0}$$

$$P_7 \stackrel{def}{=} \overline{a}(g).c(f).\mathbf{0} \mid \overline{a}(i).c(h).\mathbf{0} \mid \overline{a}(c).e(f).\mathbf{0} \mid \overline{a}(b).c(d).\mathbf{0} \mid a(b).\overline{c}b.\mathbf{0}$$

where

$$K_1 \stackrel{def}{=} (a, c, e)P_1$$
$$K_2 \stackrel{def}{=} (c, e)P_2$$

Distinctions:

$$D \stackrel{def}{=} \emptyset$$

B.2 Results of Performance Tests

The results of the performance tests can be seen in table B.1.

	Total Size of Graphs	Run Time	Result
openBisimCheck$_{First}(P_1, P_2, D)$	44 states	0.05 secs	True
openBisimCheck$_{Opt}(P_1, P_2, D)$	42 states	0.03 secs	True
openBisimCheck$_{First}(P_3, P_4, D)$	35131 states	$>$ 5 hours	?
openBisimCheck$_{Opt}(P_3, P_4, D)$	211 states	0.38 secs	True
openBisimCheck$_{First}(P_2, P_4, D)$	4105 states	2319.91 secs	False
openBisimCheck$_{Opt}(P_2, P_4, D)$	88 states	0.08 secs	False
openBisimCheck$_{First}(P_5, P_5, D)$	176 states	1.07 secs	True
openBisimCheck$_{Opt}(P_5, P_5, D)$	76 states	0.07 secs	True
openBisimCheck$_{First}(P_5, P_6, D)$	746 states	23.53 secs	False
openBisimCheck$_{Opt}(P_5, P_6, D)$	396 states	0.45 secs	False
openBisimCheck$_{First}(P_7, P_7, D)$	7834 states	10942.43 secs	True
openBisimCheck$_{Opt}(P_7, P_7, D)$	2738 states	38.96 secs	True

Table B.1: Results of performance tests of the first and optimized algorithm.

It can be seen from table B.1 that the implementation of the optimized version of the algorithm runs faster than the first version of the algorithm in the test cases. In the case where the first algorithm was run on the processes P_3 and P_4 execution was deliberately terminated prematurely, after five hours, and thus no result was returned from the algorithm, explaining the question mark in the table.

Recent BRICS Report Series Publications

RS-01-8 **Ulrik Frendrup and Jesper Nyholm Jensen.** ***Checking for Open Bisimilarity in the π-Calculus*****. February 2001. 61 pp.**

RS-01-7 **Gregory Gutin, Khee Meng Koh, Eng Guan Tay, and Anders Yeo.** ***On the Number of Quasi-Kernels in Digraphs*****. January 2001. 11 pp.**

RS-01-6 **Gregory Gutin, Anders Yeo, and Alexey Zverovich.** ***Traveling Salesman Should not be Greedy: Domination Analysis of Greedy-Type Heuristics for the TSP*****. January 2001. 7 pp.**

RS-01-5 **Thomas S. Hune, Judi Romijn, Mariëlle Stoelinga, and Frits W. Vaandrager.** ***Linear Parametric Model Checking of Timed Automata*****. January 2001. 44 pp. To appear in** ***Tools and Algorithms for The Construction and Analysis of Systems: 7th International Conference*****, TACAS '01 Proceedings, LNCS, 2001.**

RS-01-4 **Gerd Behrmann, Ansgar Fehnker, Thomas S. Hune, Kim G. Larsen, Paul Pettersson, and Judi Romijn.** ***Efficient Guiding Towards Cost-Optimality in*** **UPPAAL. January 2001. 21 pp. To appear in** ***Tools and Algorithms for The Construction and Analysis of Systems: 7th International Conference*****, TACAS '01 Proceedings, LNCS, 2001.**

RS-01-3 **Gerd Behrmann, Ansgar Fehnker, Thomas S. Hune, Kim G. Larsen, Paul Pettersson, Judi Romijn, and Frits W. Vaandrager.** ***Minimum-Cost Reachability for Priced Timed Automata*****. January 2001. 22 pp. To appear in** ***Hybrid Systems: Computation and Control*****, 2001.**

RS-01-2 **Rasmus Pagh and Jakob Pagter.** ***Optimal Time-Space Trade-Offs for Non-Comparison-Based Sorting*****. January 2001. ii+20 pp.**

RS-01-1 **Gerth Stølting Brodal, Anna Östlin, and Christian N. S. Pedersen.** ***The Complexity of Constructing Evolutionary Trees Using Experiments*****. 2001.**