A Verification Tool for the Polyadic π-Calculus

Björn Victor

May 9, 1994
Revised May 26, 1994

Abstract

This thesis describes the polyadic version of the Mobility Workbench (MWB), an automated tool for manipulating and analyzing mobile concurrent systems (those with evolving connectivity structures) described in the polyadic π-calculus. The main feature of this version of the MWB is checking open bisimulation equivalences, and doing so with efficiency in mind.

The open bisimulation equivalences are described in a polyadic setting, and efficient characterisations of both the strong and the weak equivalences are presented and proven to coincide with their standard formulations.

The equivalence checking algorithm, which by necessity generates the state space "on-the-fly", is presented in detail and proven correct. Aspects of the implementation of the tool are described in some detail, for example the representation of π-calculus agents using de Bruijn indices.

We illustrate the MWB with an example automated analysis of a handover protocol for a mobile telephone system.

The printed version of the thesis differs from the original text in some minor typographical and linguistic details.

Contents

Chapter 1

Introduction

1.1 ... for my relatives

When large systems of software and hardware are created, it is important that everything works as intended. To verify this is a difficult task, in particular when dealing with *concurrent* systems, where many parallel parts collaborate in complex ways. We need to describe the system and its intended behaviour in a precise and formal way, which could make it possible to reason about the behaviour of the system and *prove* mathematically that the system does indeed do what it should.

One way of describing such concurrent systems formally is by using a *process algebra*. There are many process algebras; one representative example is CCS (Calculus of Communicating Systems) by Robin Milner. In a process algebra, we can formally define what it means to "do what is intended", for example by defining when a description of the system and a description of what is intended are *equal*. One definition of equality is that of *bisimilarity*, where the two system descriptions (henceforth *agents*) must be able to imitate each others actions, step by step, and continue to be equal after each step.

To check if two agents are equal is still difficult and tedious work with lots of details which can be lost or forgotten, and much mechanical manipulation of these details. Computers can help us, either by just keeping track of all the details, or better still, by doing the boring mechanical manipulation too. The latter is not always possible; the behaviour of the agents may be too large to fit in the computer (they might even be infinite), or the definition of equality may be described in a way not suited for implementation in a computer program.

This thesis describes a process algebra called the π-*calculus*, invented by Robin Milner, Joachim Parrow and David Walker, in which it is possible to describe concurrent systems with dynamically evolving structures. A particular notion of bisimularity for agents in this calculus is defined. This definition is nice for formal treatment, but it is not suitable for implementation. An alternative definition is given, which *is* well suited for implementation, and it is shown mathematically that it equates the same agents as the first definition. We have written a program which we call the Mobility Workbench, or MWB for short, which checks if two agents are equal according to this new definition, and as an example of its use we describe how we have verified that a protocol for mobile telephones does what is intended.

1.2 ...for my colleagues

Process algebra is one of the standard paradigms for describing and analysing communicating concurrent systems [Mil89, BW90]. "Basic" process algebras[1], e.g. CCS without value-passing [Mil89], model the pure communication and synchronization aspects of concurrent systems. From these, various other process algebras can be derived which emphasise different aspects of the reality they are intended to model; for example there are process algebras which handle real-time notions, probabilistic concepts, or combinations of these. Many process algebras support communication of values of various types: integers, booleans, etc. These values are separate entities from the processes and communication channels, and require separate theories for formal treatment. However, in any of these process algebras it is difficult to model systems where the structure of communication is changing dynamically, for example communication protocols for mobile telephone systems.

The π-calculus is a process algebra derived from CCS, well suited for describing systems with dynamically changing communication structures. The only values being passed in this algebra are *names* (representing communication channels), but *within* the algebra it is possible to encode other types of data, such as booleans, integers, lists, and even higher-order processes [San92], giving a single unified theory for processes, communication and values. It is in fact powerful enough to encode the λ-calculus [Mil92], which can be regarded as the canonical calculus of sequential computation. The π-calculus is perhaps not *the* canonical calculus of concurrent computation (see e.g. [Mil93c] for recent developments), but it is a significant step towards one.

As for all process algebras, different equivalences have been defined for the π-calculus; the late and early bisimulation equivalences of [MPW92], and the testing equivalence of [BDN92] are examples. Recently, Davide Sangiorgi developed *open* bisimulation equivalence [San93b], which is strictly finer than the late and early equivalences. Strong open equivalence is a congruence and has a simple axiomatization, and weak open equivalence has a simpler and "cleaner" definition than weak late equivalence. Sangiorgi also gave an alternative characterization of the strong open bisimulation equivalence, inspired by the work of Hennessy and Lin [HL92] on symbolic transition systems for value-passing CCS.

The π-calculus has been used to model communication protocols [OP92, Ora94], data structures [Mil93a], logic programming [Ros93, Li94], and object-oriented programming [Wal90, Wal92, Jon93]. When these specifications have been formally analysed, the analyses were done manually, since there had been no tool available for the π-calculus. This is very tedious and error-prone work, especially for large specifications, and one may argue that for a formal method to be truly successful, it needs automated tools. We somewhat ameliorate this situtation for the π-calculus by the work described in this thesis, but there is still much work to do.

We have developed a verification tool for the π-calculus, called the Mobility Workbench (MWB). Its main initial functionality is the verification of the open bisimulation equivalences (strong and weak), using algorithms based on alternative characterizations of the equivalences. Because of problems with name instantiation (described in Chapter 3), we can unfortunately not use the efficient "standard" algorithms used for checking bisimulation equivalences for e.g. CCS, such as the ones described in [PT87, KS90], which first build the complete state spaces, and then analyse them. Instead, our algorithms construct state

[1]Not to be confused with the process algebra called Basic Process Algebra, BPA.

spaces "on-the-fly", as we build bisimulation relations.

The particular instance of the π-calculus in which we have chosen to do our work is the *polyadic* π-calculus [Mil93a]. In this calculus, each communication can pass zero or more names, while in the (original) *monadic* π-calculus [MPW92], exactly one name is passed. The polyadicity does not add expressiveness, but makes modelling less tedious, and allows a notion of *sorts* and *sorting*, which in the π-calculus plays a rôle similar to that of types and typing in sequential computation. We have not yet explored sorts and sortings in the tool, but we intend to do so in the future. We further restrict the calculus we use to only positive matching, i.e, we can only say "if-then", and not "if-then-else". This is because the open equivalence is only defined for positive matching. We use the normal recursively defined agent identifiers instead of the replication operator introduced by Milner in [Mil93a]. By disallowing parallel composition within recursively defined agents and only considering guarded recursion, we get *finite control* and a decidable equivalence problem.

An earlier version of the Mobility Workbench, for the monadic π-calculus, was described in a paper written with Faron Moller [VM94], from which parts of chapters 3 and 5 are taken.

Apart from the fact that the π-calculus is sometimes referred to as "a calculus of mobile processes", the name of the tool, "the Mobility Workbench", is reminiscent of the fact that personal mobility was necessary for its creation; in July and August of 1993, I visited the Laboratory for Foundations of Computer Science at Edinburgh University, where I worked with Faron Moller and Davide Sangiorgi, and also met a lot of other nice people. My visit was partly funded by the CONFER project, which also funds my normal mobility, when I work one day a week at the Swedish Institute of Computer Science in Kista. "The Mobility Workbench" is also vaguely related to the name "the Concurrency Workbench", a tool for CCS, the initial development of which my advisor Joachim Parrow took part in while *he* was visiting Edinburgh.

Outline: The outline of this thesis is as follows. In Chapter 2 we define the formal framework, the polyadic π-calculus, as well as the equivalences which the tool handles. In Chapter 3 we describe the verification algorithm. Chapter 4 contains some discussion about implementation matters, and in Chapter 5 we give two examples of the use of the MWB, including a full version of the (by now standard) mobile telephones example. In Chapter 6 we relate our work to that of others, and finally in Chapter 7 conclusions are drawn and future work is described. Appendix A contains the proof that the algorithm presented in chapter 3 is correct, and Appendix B contains a preliminary User's Guide to the Mobility Workbench.

Chapter 2

Definitions

In this chapter, we introduce the polyadic π-calculus and its operational semantics. We also define the equivalences we are interested in, which are the polyadic versions of the open bisimulation equivalences defined by Sangiorgi in [San93b]. Following [San93b], we also define efficient characterisations of the equivalences along with an operational semantics, and show that the strong equivalence coincides with its efficient characterisation. Finally we show that the weak equivalence coincides with its efficient characterisation, which — along with the treatment of polyadic terms – consititutes the original contributions provided in this chapter.

2.1 The Polyadic π-calculus

The π-calculus comes in two basic styles; the *monadic* calculus [MPW92], where exactly *one* name is communicated at each synchronisation, and the *polyadic* calculus [Mil93a], where *zero or more* names are communicated. We chose to work in the polyadic calculus, because we think it is easier and more elegant to use for modelling purposes, and because it allows a notion of sorts [Mil93a, Mil90], roughly analogous to types in functional programming. Just as types are helpful in functional programming, sorts are helpful in π-calculus modelling (see e.g. [Ora94]).

Terms in the polyadic π-calculus are either *abstractions* (ranged over by F, G), *concretions* (ranged over by C, E), or *processes* (ranged over by P, Q). These are collectively called *agents* (ranged over by A, B). We let $X, Y \ldots$ range over agent identifiers; $a, b, x, y, \ldots$ over names; $\overline{a}, \overline{b}, \overline{x}, \overline{y}, \ldots$ over co-names; $\alpha, \beta, \ldots$ over names, co-names or τ; $\vec{x}, \vec{y}, \ldots$ over vectors of names of length $|\vec{x}|, |\vec{y}| \geq 0$.

The syntax of the π-calculus is given by the following BNF equations.

$$
\begin{array}{lcll}
A & ::= & F \mid C \mid X & \textit{agents} \\
F & ::= & P \mid (\lambda x)F \mid (\boldsymbol{\nu} x)F & \textit{abstractions} \\
C & ::= & P \mid [x]C \mid (\boldsymbol{\nu} x)C & \textit{concretions} \\
P & ::= & \mathbf{0} \mid \alpha.A \mid P + Q \mid P \mid Q \mid (\boldsymbol{\nu} x)P \mid [a = b]P & \textit{processes} \\
\alpha & ::= & x \;\; (\textit{input}) \;\mid\; \overline{x} \;\; (\textit{output}) \;\mid\; \tau \;\; (\textit{internal}) &
\end{array}
$$

Briefly, $\mathbf{0}$ represents an inactive process; each agent identifier X has a corresponding definitional body A (written $X \stackrel{def}{=} A$; we only allow guarded recursion); matching $[a = b]P$ is read as *"if $a = b$ then P"*; sum $M + N$ offers

1. $P \equiv Q$ if P is α-convertible to Q
2. Abelian monoid laws for $+$ and $\mathbf{0}$:
   $P+Q \equiv Q+P, P+(Q+R) \equiv (P+Q)+R, P+\mathbf{0} \equiv P$
3. Abelian monoid laws for $\mid$ and $\mathbf{0}$
4. $(\boldsymbol{\nu} x)\mathbf{0} \equiv \mathbf{0}, (\boldsymbol{\nu} x)(\boldsymbol{\nu} y)P \equiv (\boldsymbol{\nu} y)(\boldsymbol{\nu} x)P, (\boldsymbol{\nu} x)P \mid Q \equiv (\boldsymbol{\nu} x)(P \mid Q)$ if $x \notin \mathrm{fn}(Q)$
5. $(\boldsymbol{\nu} y)(\lambda x)P \equiv (\lambda x)(\boldsymbol{\nu} y)P$, if $x \neq y$
6. $(\boldsymbol{\nu} y)[x]P \equiv [x](\boldsymbol{\nu} y)P$, if $x \neq y$
7. $(\boldsymbol{\nu} x)(\boldsymbol{\nu} x)P \equiv (\boldsymbol{\nu} x)P$
8. $[x=x]P \equiv P$
9. $X \equiv P$ if $X \stackrel{def}{=} P$

Figure 2.1: Structural congruence

the choice of M or N; composition $P_1 \mid P_2$ places the two processes P_1 and P_2 side-by-side in parallel execution; restriction $(\boldsymbol{\nu} x)P$ hides the name x from the environment of P; and $\alpha.A$ can perform the relevant input, output or silent τ action, thereby evolving into A. An input prefix $x.$ is called *positive*, and always prefixes an abstraction, while an output prefix $\overline{x}.$ is called *negative* and always prefixes a concretion. A τ prefix always prefixes a process. We write $(\lambda\vec{x})$ for $(\lambda x_1)(\lambda x_2)\ldots(\lambda x_n)$ where $\vec{x} = (x_1, x_2, \ldots, x_n)$ and likewise for $(\boldsymbol{\nu}\vec{x})$ and $[\vec{x}]$.

In some standard works on the π-calculus (e.g. [Mil93a]), a *replication* operator "!" is defined, where $!P$ means $P \mid !P$, or "as many copies of P as you need". Another notable variant of the π-calculus syntax is to allow mismatching, either by an if-then-else construction $[a=b]P\ Q$, or by an explicit mismatch operator $[a \neq b]P$ (see e.g. [PS93]).

We now give an operational semantics for the π-calculus, which we will use to define the bisimulation equivalences. This is based on [Mil93a] and, in part, on [San93a]. The semantics consists of two parts: a structural congruence (Figure 2.1) and a commitment relation (Figure 2.2). Note that because of the STRUCT rule, we do not need symmetric rules for SUM, PAR and COMM. In $P \succ \alpha.A$, we call A the *derivative* of the commitment.

Definition 2.1 (Standard form) [Mil93a] *An abstraction F is in* standard form *if it is written $(\lambda\vec{x})P$, where P is a process.*

A concretion C is in standard form *if it is written $(\boldsymbol{\nu}\vec{y})[\vec{x}]P$, where $\vec{y} \subseteq \vec{x}$ and P is a process.*

A useful fact (due to Milner [Mil93a]) is the following:

1. Every abstraction F can be converted to a *standard form*

$$F \equiv (\lambda\vec{x})P$$

   by pushing restrictions inwards using rules 1 and 5 of the structural congruence.

2. Every concretion can be converted to a *standard form*

$$C \equiv (\boldsymbol{\nu}\vec{y})[\vec{x}]P \quad (\vec{y} \subseteq \vec{x})$$

   by pulling restrictions on names in $\vec{x}$ outwards and pushing all others inwards, using rules 1 and 6 of the structural congruence.

$$\textsc{pre}: \ \alpha.P \succ \alpha.P \qquad \textsc{sum}: \ \frac{P \succ \alpha.P'}{P + Q \succ \alpha.P'}$$

$$\textsc{comm}: \ \frac{P \succ x.F \qquad Q \succ \overline{x}.C}{P \mid Q \succ \tau.(F \bullet C)} \qquad \textsc{par}: \ \frac{P \succ \alpha.P'}{P \mid Q \succ \alpha.(P' \mid Q)}$$

$$\textsc{res}: \ \frac{P \succ \alpha.P'}{(\boldsymbol{\nu} b)P \succ \alpha.(\boldsymbol{\nu} b)P'} \ b \notin \mathrm{n}(\alpha)$$

$$\textsc{struct}: \ \frac{P \equiv Q \qquad P \succ \alpha.P' \qquad P' \equiv Q'}{Q \succ \alpha.Q'}$$

Figure 2.2: Commitment relation

Definition 2.2 (Arity) [Mil93a] *The* arity *of an abstraction $A \equiv (\lambda\vec{x})P$ or concretion $A \equiv (\boldsymbol{\nu}\vec{y})[\vec{x}]P$ is the length of the vector $\vec{x}$ in the standard form of A.*

Definition 2.3 (Pseudo-application) [Mil93a]
Pseudo-application *of an abstraction F to a concretion C where F and C have equal arities, (used in the* COMM *rule below), is defined as follows.*
We write $F \bullet C$ for $(\boldsymbol{\nu}\vec{z})(P\{\vec{y}/\vec{x}\} \mid Q)$ where F has standard form $(\lambda\vec{x})P$, C has standard form $(\boldsymbol{\nu}\vec{z})[\vec{y}]Q$, $\vec{x} \cap \vec{z} = \emptyset$ and $|\vec{x}| = |\vec{y}|$

Definition 2.4 (Application) [Mil93a]
We write $((\lambda x)P)y$ for $P\{y/x\}$.

Definition 2.5 (Composition of concretions and abstractions) [Mil93a]
We extend parallel composition to handle composition of concretions with concretions and abstractions with abstractions. This is used in the PAR *rule.*

1. *We write $F \mid G$ for $(\lambda\vec{x}\vec{y})(P \mid Q)$ where $F \equiv (\lambda\vec{x})P$ and $G \equiv (\lambda\vec{y})Q$ and $\vec{x} \not\subseteq \mathrm{n}(G)$ and $\vec{y} \not\subseteq \mathrm{n}(F)$.*
2. *We write $C \mid D$ for $(\boldsymbol{\nu}\vec{x}\vec{y})[\vec{u}\vec{v}](P \mid Q)$ where $C \equiv (\boldsymbol{\nu}\vec{x})[\vec{u}]P$ and $D \equiv (\boldsymbol{\nu}\vec{y})[\vec{v}]Q$ and $\vec{x} \not\subseteq \mathrm{n}(D)$ and $\vec{y} \not\subseteq \mathrm{n}(C)$.*

(Note that $\mid$ is still commutative on processes, and that $A \mid P \equiv P \mid A$ for any agent A and process P).

To handle the restriction operator in our definition of bisimulations, we need to introduce *distinctions* [MPW92], ranged over by D.

Definition 2.6 (Distinction) *A* distinction *is a finite symmetric and irreflexive binary relation over names.*

We use distinctions to express which names are not allowed to be equated by instantiation. This is used when handling restictions — a restricted name is different from all other known names.

Definition 2.7

1. *A substitution σ* respects *a distinction D if $(a,b) \in D$ implies $\sigma(a) \neq \sigma(b)$.*
2. *We write $D\sigma$ for the distinction $\{(\sigma(a), \sigma(b)) : (a,b) \in D\}$.*

We will sometimes write distinctions without giving all symmetric pairs, e.g., we will write $\{(a,b)\}$ when we really mean $\{(a,b),(b,a)\}$.

2.2 Open Bisimulation Equivalences

Of the various different equivalence relations for π-calculus agents, most notable of which are perhaps the *late* and *early* bisimulation equivalences of [MPW92], we have chosen to work with the *open* bisimulation equivalences of Sangiorgi [San93b]. While the early equivalence instantiates a bound name at the synchronisation and late equivalence instantiates it just after the synchronisation, the open equivalence does not instantiate names until it can no longer be deferred. This leads to a uniform treatment of names as variables, while names in the late and early cases sometimes play the rôle of constants and sometimes of variables.

Notable features of the strong open equivalence are that it is a congruence, is strictly finer that the late and early equivalences and congruences, has a simple axiomatization, and, most importantly for implementation, has an efficient characterization which we use in our verification tool. The weak equivalence also has an efficient characterization, but the exact relation between the open weak equivalence and the late and early weak equivalences, respectively, has not yet been investigated.

Sangiorgi defined the strong and weak open bisimulation equivalences for the monadic calculus, and gave an efficient characterization for the strong equivalence. In this section we define the open bisimulation equivalences for the polyadic calculus, and give efficient characterizations for both the strong and weak equivalences. In order to handle distinctions, the bisimulation definitions use *sets* of relations, indexed by distinctions.

Definition 2.8 (Open D-bisimilarity, $\sim_D$, standard characterization)
The set $\mathcal{R} = \{\mathcal{S}_D\}_{\mathcal{D}}$ of agent relations, where $\mathcal{D}$ is a set of distinctions, each $D \in \mathcal{D}$, is an indexed open simulation *if for each $\mathcal{S}_D \in \mathcal{R}$ and for each σ which respects D, $P\ \mathcal{S}_D\ Q$ implies*

1. *whenever $P\sigma \succ \alpha.P'$ then there exists a Q' such that $Q\sigma \succ \alpha.Q'$ and $P'\ \mathcal{S}_{D\sigma}\ Q'$ where $\mathcal{S}_{D\sigma} \in \mathcal{R}$.*
2. *when $P\sigma$ has standard form $(\boldsymbol{\nu}\vec{x})[\vec{y}]P'$ then $Q\sigma$ has standard form $(\boldsymbol{\nu}\vec{x})[\vec{y}]Q'$ and $P'\ \mathcal{S}_{D'}\ Q'$, where $D' = D\sigma \cup (\vec{x} \times \mathrm{fn}(P\sigma, Q\sigma))$ and $\mathcal{S}_{D'} \in \mathcal{R}$.*
3. *when $P\sigma$ has standard form $(\lambda\vec{x})P'$ then $Q\sigma$ has standard form $(\lambda\vec{x})Q'$ and $P'\ \mathcal{S}_{D\sigma}\ Q'$, where $\mathcal{S}_{D\sigma} \in \mathcal{R}$*

$\mathcal{R} = \{\mathcal{S}_D\}_{\mathcal{D}}$ is an indexed open bisimulation *if both $\{\mathcal{S}_D\}_{\mathcal{D}}$ and $\{\mathcal{S}_D^{-1}\}_{\mathcal{D}}$ are indexed open simulations.*

The agents P and Q are open D-bisimilar*, written $P \sim_D Q$, if there is an indexed open bisimulation $\mathcal{R} = \{\mathcal{S}_D\}_{\mathcal{D}}$ such that $P\ \mathcal{S}_D\ Q$ for some $\mathcal{S}_D \in \mathcal{R}$.*

Since we confine ourselves to finite branching agents, this may be equivalently expressed as

$$\sim_D = \bigcup\{\mathcal{R} : \mathcal{R} \textit{ is an indexed open bisimulation}\}$$

The second clause of the definition extends the distinction, making the restricted names different from all other free names in the agents.

As examples,

$$[x = y]z.\mathbf{0} \sim_{\{(x,y)\}} \mathbf{0}$$

and

$$\overline{x}\,.\mathbf{0} \mid y.\mathbf{0} \sim_{\{(x,y)\}} \overline{x}\,.y.\mathbf{0} + y.\overline{x}\,.\mathbf{0}$$

Lemma 2.9 *For the monadic subset of the polyadic π-calculus, i.e. for agents using abstractions and concretions of arity 1 only, Definition 2.8 coincides with the corresponding definition (6.2) in [San93b].*

PROOF: By straightforward transition induction in the monadic setting, defining $a.(\lambda x)P$ as $a(x).P$ and $\overline{a}\,.[x]P$ as $\overline{a}\; x.P$. □

2.2.1 Efficient Characterisation

To avoid the universal quantification over substitutions in Definition 2.8, Sangiorgi [San93b] defines an alternative characterisation of the equivalence. When we carry the characterisation over to the polyadic setting we base it on a variant of the commitment relation. The relation is indexed by a finite (possibly empty) *match sequence* (ranged over by M, N, L), e.g. $[a = b][c = d]$. The intuition of the match sequence is that it is a minimal condition for the commitment to be possible. M also defines a substitution σ_M, which is the "minimal" substitution which would allow the commitment in the standard characterisation.

For example,

$$[a = b]\overline{a}\,.[x]P \succ_{[a=b]} \overline{a}\,.[x]P$$

where $a = b$ is the condition for the $\overline{a}$ commitment. If $\sigma_{[a=b]} = \{a/b\}$, we have that

$$([a = b]\overline{a}\,.[x]P)\sigma_{[a=b]} \succ \overline{a}\,.[x]P$$

using the standard characterisation.

The new conditional structural congruence has the same rules as for the standard structural congruence (Figure 2.2), except that rule 8 for the match operator is removed and a corresponding rule is instead added to the conditional commitment relation.

The conditional version of the commitment relation (Figure 2.3) is similar to the standard version (see Figure 2.2), but adds a rule MATCH, and for simplicity, replaces STRUCT by DEF, ABEL, and ALPHA. In the figure, we treat the match sequence $[a = a]$ as the empty sequence; thus $M[a = a]$ is identical to M.

Thus we only need the structural congruence relation for rewriting agents to standard form (Definition 2.1).

The following definition and lemma give what we need about match sequences and substitutions.

Definition 2.10

1. *$M \vartriangleright N$ if whenever the matches in M are true, then the matches in N are also true.*

2. *$M \vartriangleleft\!\vartriangleright N$ if $M \vartriangleright N$ and $N \vartriangleright M$*

$$\text{PRE}: \ \alpha.P \succ_{\emptyset} \alpha.P \qquad \text{SUM}: \ \frac{P \succ_M \alpha.P'}{P + Q \succ_M \alpha.P'}$$

$$\text{COMM}: \ \frac{P \succ_M x.F \qquad Q \succ_N \overline{y}.C}{P \mid Q \succ_{MN[x=y]} \tau.(F \bullet C)} \qquad \text{PAR}: \ \frac{P \succ_M \alpha.P'}{P \mid Q \succ_M \alpha.(P' \mid Q)}$$

$$\text{RES}: \ \frac{P \succ_M \alpha.P'}{(\nu b)P \succ_M \alpha.(\nu b)P'} \ b \notin \mathrm{n}(\alpha, M)$$

$$\text{MATCH}: \ \frac{P \succ_M \alpha.P'}{[a=b]P \succ_{M[a=b]} \alpha.P'}$$

$$\text{DEF}: \ \frac{X \stackrel{def}{=} (\lambda\vec{x})P \qquad ((\lambda\vec{x})P)\vec{y} \succ_M \alpha.P'}{X\vec{y} \succ_M \alpha.P'} \qquad \text{ALPHA}: \ \frac{P =_\alpha Q \qquad Q \succ_M \beta.Q'}{P \succ_M \beta.Q'}$$

$$\text{ABEL}: \ \frac{P = Q \text{ due to abelian monoid laws for } \mid \text{ and } + \qquad P \succ_M P'}{Q \succ_M P'}$$

Figure 2.3: Conditional commitment relation

3. *We write* $\mathrm{n}(M)$ *for the names of* M*.*
4. M respects *a distinction* D *if* $(a, b) \in D$ *implies* $M \not\triangleright [a = b]$
5. *We denote by* σ_M *a minimal substitution on names induced by the equivalence classes associated with the equivalence relation corresponding to* M*; we select one representative of each class and map the other members of the class to it. (We don't care how the representative is selected as long as it is deterministic).*
6. $P\sigma\rho = (P\sigma)\rho$
7. $\sigma \triangleright \rho$ *if* $\rho(a) = \rho(b)$ *implies* $\sigma(a) = \sigma(b)$

Lemma 2.11 (Sangiorgi [San93b, Lemma 5.3])

1. *If* $M \triangleright N$ *then* $M\sigma \triangleright N\sigma$
2. *If* $M \triangleright N$ *then* $\sigma_M \triangleright \sigma_N$
3. *If* $\sigma \triangleright \sigma'$ *then there exists a substitution* ρ *such that* $\sigma = \sigma'\rho$

As in [San93b], in all lemmas below, by alpha conversion it is assumed that a substitution σ acts as an identity on the bound names of the agents and keeps the separation between bound and free names. Therefore, in $\big((\boldsymbol{\nu}x)P\big)\sigma$, the free occurrences of x in P are not modified by σ.

We now have the machinery necessary to give the conditional characterisation of open bisimilarity.

Definition 2.12 (Conditional open D-bisimilarity, $\asymp_D$)
The set $\mathcal{R} = \{\mathcal{S}_D\}_\mathcal{D}$ is an indexed conditional open simulation *if for each $\mathcal{S}_D \in \mathcal{R}$, $P\ \mathcal{S}_D\ Q$ implies*

1. *whenever $P \succ_M \alpha.P'$ such that M respects D, then $\exists N, \beta$ and Q' such that $Q \succ_N \beta.Q'$ and*
   (a) $M \rhd N$
   (b) $\alpha\sigma_M = \beta\sigma_M$
   (c) $P'\sigma_M\ \mathcal{S}_{D\sigma_M}\ Q'\sigma_M$, *where* $\mathcal{S}_{D\sigma_M} \in \mathcal{R}$
2. *when P has standard form $(\boldsymbol{\nu}\vec{x})[\vec{y}]P'$, then Q has standard form $(\boldsymbol{\nu}\vec{x})[\vec{y}]Q'$, and $P'\ \mathcal{S}_{D'}\ Q'$, where $D' = D \cup \{\vec{x} \times \mathrm{fn}(P,Q)\}$ and $\mathcal{S}_{D'} \in \mathcal{R}$*
3. *when P has standard form $(\lambda\vec{x})P'$ then Q has standard form $(\lambda\vec{x})Q'$ and $P'\ \mathcal{S}_D\ Q'$*

$\mathcal{R} = \{\mathcal{S}_D\}_\mathcal{D}$ *is an* indexed conditional open bisimulation *if both $\{\mathcal{S}_D\}_\mathcal{D}$ and $\{\mathcal{S}_D^{-1}\}_\mathcal{D}$ are indexed conditional open simulations.*

The agents P and Q are conditional open D-bisimilar, *written $P \asymp_D Q$, if there is an indexed conditional open bisimulation $\mathcal{R} = \{\mathcal{S}_D\}_\mathcal{D}$ such that $P\ \mathcal{S}_D\ Q$. This may be equivalently expressed as*

$$\asymp_D = \bigcup\{\mathcal{R} : \mathcal{R} \text{ is a conditional open } D\text{-bisimulation}\}$$

To show that this alternative characterisation of open bisimulation equivalence makes sense, we now show that the two definitions equate the same agents, i.e. that $\asymp_D$ coincides with $\sim_D$ (Theorem 2.22), and for this we need the following propositions and lemmas (2.15 through 2.20), which are translated to the polyadic calculus from [San93b], where Sangiorgi proved the corresponding theorem for the monadic calculus.

Definition 2.13 *A set $\mathcal{R} = \{\mathcal{S}_D\}_\mathcal{D}$ of agent relations is* closed under respectful substitutions *if for each $\mathcal{S}_D \in \mathcal{R}$, for all P and Q, $P\ \mathcal{S}_D\ Q$ and σ respects D implies $P\sigma\ \mathcal{S}_{D\sigma}\ Q\sigma$.*

Definition 2.14 (Indexed ground bisimulation) *A set $\mathcal{R} = \{\mathcal{S}_D\}_\mathcal{D}$ of agent relations is an* indexed ground simulation *if for each $\mathcal{S}_D \in \mathcal{R}$, $P\ \mathcal{S}_D\ Q$ implies*

- *whenever $P \succ \alpha.P'$ then $\exists Q'$ such that $Q \succ \alpha.Q'$ and $P'\ \mathcal{S}_D\ Q'$*
- *when P has standard form $(\boldsymbol{\nu}\vec{x})[\vec{y}]P'$, then Q has standard form $(\boldsymbol{\nu}\vec{x})[\vec{y}]Q'$ and $P'\ \mathcal{S}_{D'}\ Q'$ for $D' = D \cup \{\vec{x} \times \mathrm{fn}(P,Q)\}$*
- *when P has standard form $(\lambda\vec{x})P'$, then Q has standard form $(\lambda\vec{x})Q'$ and $P'\ \mathcal{S}_D\ Q'$*

$\mathcal{R} = \{\mathcal{S}_D\}_\mathcal{D}$ *is an* indexed ground bisimulation *if both $\{\mathcal{S}_D\}_\mathcal{D}$ and $\{\mathcal{S}_D^{-1}\}_\mathcal{D}$ are indexed ground simulations.*

The proofs of Lemmas 2.17, 2.19 and 2.20 are very close to the ones in [San93b]. The Propositions and Lemmas using distinctions (2.15, 2.16 and 2.18) have proofs very close to the corresponding ones in Section 2.2.2, using strong lemmas and definitions instead of weak. We choose to only present the weak versions of the proofs in this thesis.

Proposition 2.15 *Suppose that $\mathcal{R} = \{\mathcal{S}_D\}_\mathcal{D}$ is a set of relations which is an indexed ground bisimulation, and closed under respectful substitutions. Then $\mathcal{R}$ is an indexed open bisimulation.*

Proposition 2.16 *$\sim_D$ is the largest indexed ground bisimulation which is closed under all respectful substitutions.*

Lemma 2.17

1. *If $P \succ_M \alpha.Q$, then for all σ there are M', α' and Q' such that $P\sigma \succ_{M'} \alpha'.Q'$, where $M' \lhd\!\rhd M\sigma$, $\alpha' = \alpha\sigma$, and $Q' =_\alpha Q\sigma$*
2. *For all σ, if $P\sigma \succ_{M'} \alpha'.Q'$, then there are M, α and Q such that $P \succ_M \alpha.Q$, where $M' \lhd\!\rhd M\sigma$, $\alpha' = \alpha\sigma$, and $Q' =_\alpha Q\sigma$.*

Lemma 2.18 *$\asymp_D$ is closed under respectful substitutions, i.e., $P \asymp_D Q$ implies $P\sigma \asymp_{D\sigma} Q\sigma$ for any σ respecting D.*

Lemma 2.19 *If $P\sigma_M \succ \alpha.Q'$ then $P \succ_N \beta.Q$ where*

- $M \rhd N$
- $\alpha = \beta\sigma_M$
- $Q' =_\alpha Q\sigma_M$

Lemma 2.20 *If $P \succ_M \alpha.P'$ then $P\sigma_M \succ \alpha\sigma_M.P'\sigma_M$.*

Lemma 2.21 *If $P \succ \alpha.P'$ then $P\sigma \succ \alpha\sigma.P'\sigma$*
Proof: By induction on the depth of inference, using the assumption that σ acts as an identity on the bound names of the agents. □

Theorem 2.22 *$\sim_D$ coincides with $\asymp_D$.*
Proof: Following Sangiorgi [San93b], using the above lemmas in place of the corresponding monadic ones. This proof has the same structure as the proof of Theorem 2.27 (which relates the characterizations of the weak equivalence); we choose to present that proof in detail instead. □

2.2.2 Weak Relations

In this section we define the weak equivalence based on a weak version of the commitment relation. We start by giving the weak version of the standard commitment relation. In this definition, we use $(\succ \tau.)^\star$ to mean the reflexive and transitive closure of the relation $(\succ \tau.)$, which is defined by

$$P(\succ \tau.)P' \quad \textit{iff} \quad P \succ \tau.P'$$

Definition 2.23 (Weak standard commitment relation, $\ggcurly$)
$P \ggcurly \alpha.Q$ iff

1. *$\alpha = \tau$ and $P(\succ \tau.)^\star Q$;*

2. *if* $\alpha \neq \tau$ *then*
   $P(\succ \tau.)^\star P' \succ \alpha.P''$ *and*
   - *if* P'' *is an abstraction in standard form* $(\lambda\vec{x})F$*, then* $F(\succ \tau)^\star.P'''$*, and* $Q \equiv (\lambda\vec{x})P'''$*;*
   - *if* P'' *is a concretion in standard form* $(\boldsymbol{\nu}\vec{x})[\vec{y}]C$*, then* $C(\succ \tau)^\star.P'''$*, and* $Q \equiv (\boldsymbol{\nu}\vec{x})[\vec{y}]P'''$*;*
   - *if* P'' *is a process, then* $P''(\succ \tau.)^\star Q$*.*

The following definition is the variant of the weak standard commitment relation used in the conditional characterisation of the weak open equivalence. The main difference from the preceding definition is that it collects the conditions for the involved strong commitments.

Definition 2.24 (Weak conditional commitment relation, $\succ\!\!\succ_M$)
$P \succ\!\!\succ_M \alpha.Q$ iff

1. $\alpha = \tau$ *and* $P(\succ_{N_1} \tau \ldots \succ_{N_n} \tau.)Q$ *for* $n \geq 0$*, where* $M = N_1 \cdots N_n$*;*
2. *if* $\alpha \neq \tau$ *then*
   $P(\succ_{N_1} \tau \ldots \succ_{N_n} \tau.)P_n$*,* $P_n \succ_L \alpha.P_{n+1}$ *and*
   - *if* P_{n+1} *is an abstraction in standard form* $(\lambda\vec{x})F$*, then* $F(\succ_{K_1} \tau \ldots \succ_{K_m} \tau.)P_{n+m+1}$*, and* $Q \equiv (\lambda\vec{x})P_{n+m+1}$*;*
   - *if* P_{n+1} *is a concretion in standard form* $(\boldsymbol{\nu}\vec{x})[\vec{y}]C$*, then* $C(\succ_{K_1} \tau \ldots \succ_{K_m} \tau.)P_{n+m+1}$*, and* $Q \equiv (\boldsymbol{\nu}\vec{x})[\vec{y}]P_{n+m+1}$*;*
   - *if* P_{n+1} *is a process, then* $P_{n+1}(\succ_{K_1} \tau \ldots \succ_{K_m} \tau.)Q$*.*

   for $n, m \geq 0$*, where* $M = N_1 \cdots N_n \cdot L \cdot K_1 \cdots K_m$*, and the conditions on the* τ *commitments after the* α *commitment must not use the names bound, i.e.,* $\mathrm{n}(K_1 \cdots K_m) \cap \vec{x} = \emptyset$*.*

Next follow the definitions of the standard and conditional characterizations of weak bisimilarity. The difference from their strong counterparts is the use of the weak commitment relation.

Definition 2.25 (Standard weak open D-bisimilarity, $\approx_D$) *The set* $\mathcal{R} = \{\mathcal{S}_D\}_\mathcal{D}$ *of agent relations is an* indexed weak open simulation *if for each* $\mathcal{S}_D \in \mathcal{R}$ *and for each* σ *which respects* D*,* $P\ \mathcal{S}_D\ Q$ *implies*

1. *whenever* $P\sigma \succ \alpha.P'$ *then there exists a* Q' *such that* $Q\sigma \succ\!\!\succ \alpha.Q'$ *and* $P'\ \mathcal{S}_{D\sigma}\ Q'$*, where* $\mathcal{S}_{D\sigma} \in \mathcal{R}$*;*
2. *when* $P\sigma$ *has standard form* $(\boldsymbol{\nu}\vec{x})[\vec{y}]P'$ *then* $Q\sigma$ *has standard form* $(\boldsymbol{\nu}\vec{x})[\vec{y}]Q'$ *and* $P'\ \mathcal{S}_{D'}\ Q'$*, where* $D' = D\sigma \cup \{\vec{x} \times \mathrm{fn}(P\sigma, Q\sigma)\}$ *and* $\mathcal{S}_{D'} \in \mathcal{R}$*;*
3. *when* $P\sigma$ *has standard form* $(\lambda\vec{x})P'$ *then* $Q\sigma$ *has standard form* $(\lambda\vec{x})Q'$ *and* $P'\ \mathcal{S}_{D\sigma}\ Q'$*, where* $\mathcal{S}_{D\sigma} \in \mathcal{R}$*.*

$\mathcal{R} = \{\mathcal{S}_D\}_\mathcal{D}$ *is a* weak open D-bisimulation *if both* $\{\mathcal{S}_D\}_\mathcal{D}$ *and* $\{\mathcal{S}_D^{-1}\}_\mathcal{D}$ *are open simulations.*

The agents P *and* Q *are* weak open D-bisimilar*, written* $P \approx_D Q$*, if there is a weak open D-bisimulation* $\mathcal{R} = \{\mathcal{S}_D\}_\mathcal{D}$ *such that* $P\ \mathcal{S}_D\ Q$*. This may be equivalently expressed as*

$$\approx_D = \bigcup\{\mathcal{R} : \mathcal{R} \text{ is a weak open } D\text{-bisimulation}\}$$

A simple example is

$$(\boldsymbol{\nu} c)(a.\overline{c}.\mathbf{0} \mid c.\overline{b}.\mathbf{0}) \approx_{\emptyset} a.\overline{b}.\mathbf{0}$$

Definition 2.26 (Weak conditional open D-bisimilarity, $\lessapprox_D$)
The set $\mathcal{R} = \{\mathcal{S}_D\}_{\mathcal{D}}$ is an indexed conditional weak open simulation *if for each $\mathcal{S}_D \in \mathcal{R}$, $P\ \mathcal{S}_D\ Q$ implies*

1. *whenever $P \succ_M \alpha.P'$ such that M respects D, then $\exists N, \beta$ and Q' such that $Q \succ\!\!\succ_N \beta.Q'$ and*
   (a) $M \rhd N$
   (b) $\alpha\sigma_M = \beta\sigma_M$
   (c) $P'\sigma_M\ \mathcal{S}_{D\sigma_M}\ Q'\sigma_M$, *where* $\mathcal{S}_{D\sigma_M} \in \mathcal{R}$
2. *when P has standard form $(\boldsymbol{\nu}\vec{x})[\vec{y}]P'$, then Q has standard form $(\boldsymbol{\nu}\vec{x})[\vec{y}]Q'$, and $P'\ \mathcal{S}_{D'}\ Q'$, where $D' = D \cup \{\vec{x} \times \mathrm{fn}(P,Q)\}$ and $\mathcal{S}_{D'} \in \mathcal{R}$*
3. *when P has standard form $(\lambda\vec{x})P'$ then Q has standard form $(\lambda\vec{x})Q'$ and $P'\ \mathcal{S}_D\ Q'$*

$\mathcal{R} = \{\mathcal{S}_D\}_{\mathcal{D}}$ *is an* indexed conditional weak open D-bisimulation *if both $\{\mathcal{S}_D\}_{\mathcal{D}}$ and $\{\mathcal{S}_D^{-1}\}_{\mathcal{D}}$ are $\lessapprox_D$-simulations. The agents P and Q are* weak conditional open D-bisimilar, *written $P \lessapprox_D Q$, if there is an indexed conditional weak open D-bisimulation $\mathcal{R} = \{\mathcal{S}_D\}_{\mathcal{D}}$ such that $P\ \mathcal{S}_D\ Q$ This may be equivalently expressed as*

$$\lessapprox_D = \bigcup\{\mathcal{R} : \mathcal{R} \textit{ is a weak conditional open } D\textit{-bisimulation}\}$$

The conditional characterisation of the weak open equivalence is what we have implemented in the tool, so we want to make sure that this equivalence relates the same agents as the standard version. This is expressed by the following theorem.

Theorem 2.27 *$\approx_D$ coincides with $\lessapprox_D$.*

We prove this theorem by stating the weak versions of the propositions and lemmas needed for the strong version of this theorem (2.22). These are proven from the strong versions. The less theoretically minded person may choose to skip forward to the next chapter, on page 20, since the remainder of this chapter is devoted to the proof of Theorem 2.27.

Definition 2.28 (Indexed weak ground bisimulation) *A set $\mathcal{R} = \{\mathcal{S}_D\}_{\mathcal{D}}$ of agent relations is an* indexed weak ground simulation *if for each $\mathcal{S}_D \in \mathcal{R}$, $P\ \mathcal{S}_D\ Q$ implies*

- *whenever $P \succ \alpha.P'$ then $\exists Q'$ such that $Q \succ\!\!\succ \alpha.Q'$ and $P'\ \mathcal{S}_D\ Q'$*
- *when P has standard form $(\boldsymbol{\nu}\vec{x})[\vec{y}]P'$, then Q has standard form $(\boldsymbol{\nu}\vec{x})[\vec{y}]Q'$ and $P'\ \mathcal{S}_{D'}\ Q'$ for $D' = D \cup \{\vec{x} \times \mathrm{fn}(P,Q)\}$*
- *when P has standard form $(\lambda\vec{x})P'$, then Q has standard form $(\lambda\vec{x})Q'$ and $P'\ \mathcal{S}_D\ Q'$*

$\mathcal{R} = \{\mathcal{S}_D\}_{\mathcal{D}}$ *is an* indexed weak ground bisimulation *if both $\{\mathcal{S}_D\}_{\mathcal{D}}$ and $\{\mathcal{S}_D^{-1}\}_{\mathcal{D}}$ are indexed weak ground simulations.*

Proposition 2.29 *Let $R = \{\mathcal{S}_D\}_{\mathcal{D}}$ be an indexed weak ground bisimulation, which is closed under respectful substitutions. Then $\mathcal{R}$ is a weak open D-bisimulation.*

PROOF: The two definitions (2.13 and 2.28) together give the definition of a weak open D-bisimulation (2.25). An indexed weak ground bisimulation has the same definition as a weak open D-bisimulation, except that it is not closed under respectful substitutions, but if we add that, we get exactly a weak open D-bisimulation. □

Proposition 2.30 *$\approx_D$ is the largest indexed ground bisimulation which is closed under all respectful substitutions.*

PROOF: By contradiction. If $\mathcal{R}$ were larger than $\approx_D$, but still an indexed weak ground bisimulation, closed under all respectful substitutions, then (by proposition 2.29) it would be a indexed weak open bisimulation but larger than $\approx_D$, which is the largest weak open bisimulation. □

Lemma 2.31

1. *If $P \succ\!\!\succ_M \alpha.Q$, then for all σ there are M', α' and Q' such that $P\sigma \succ\!\!\succ_{M'} \alpha'.Q'$, where $M' \lhd\!\rhd M\sigma$, $\alpha' = \alpha\sigma$, and $Q' =_\alpha Q\sigma$*
2. *For all σ, if $P\sigma \succ\!\!\succ'_M \alpha'.Q'$, then there are M, α and Q such that $P \succ\!\!\succ_M \alpha.Q$, where $M' \lhd\!\rhd M\sigma$, $\alpha' = \alpha\sigma$, and $Q' =_\alpha Q\sigma$.*

PROOF:
CLAUSE 1: By examining all cases of α.

We choose to show here only the case where α is an input action. In this case, $P \succ\!\!\succ_M \alpha.Q$ means

$$P \succ_{N_1} \tau.P_1 \succ_{N_2} \tau.P_2 \succ_{N_3} \cdots \tau.P_n \succ_L \alpha.(\lambda\vec{x})P_{n+1}$$

$$P_{n+1} \succ_{K_1} \tau.P_{n+2} \cdots \tau.P_{n+m} \succ_{K_m} \tau.P_{n+m+1}$$

where $Q = (\lambda\vec{x})P_{n+m+1}$, for $n, m \geq 0$, where $M = N_1 \cdots N_n \cdot L \cdot K_1 \cdots K_m$, and $\mathrm{n}(K_1 \cdots K_m) \cap \vec{x} = \emptyset$. Using the strong version of this lemma (2.17(1)) for the implications, we can complete the following table (for $n, m \geq 0$).

$$\begin{array}{lcll}
P \succ_{N_1} \tau.P_1 & \Rightarrow & P\sigma \succ_{N_1'} \tau.P_1' & (N_1' \lhd\!\rhd N_1\sigma, P_1' =_\alpha P_1\sigma) \\
P_1 \succ_{N_2} \tau.P_2 & \Rightarrow & P_1\sigma \succ_{N_2'} \tau.P_2' & (N_2' \lhd\!\rhd N_2\sigma, P_2' =_\alpha P_2\sigma) \\
\vdots & & & \\
P_{n-1} \succ_{N_n} \tau.P_n & \Rightarrow & P_{n-1}\sigma \succ_{N_n'} \tau.P_n' & (N_n' \lhd\!\rhd N_n\sigma, P_n' =_\alpha P_n\sigma) \\
P_n \succ_L \alpha.(\lambda\vec{x})P_{n+1} & \Rightarrow & P_n\sigma \succ_{L'} \alpha'.(\lambda\vec{x})P_{n+1}' & \begin{pmatrix} L' \lhd\!\rhd L\sigma, \alpha' = \alpha\sigma \\ (\lambda\vec{x})P_{n+1}' =_\alpha (\lambda\vec{x})P_{n+1}\sigma \end{pmatrix} \\
P_{n+1} \succ_{K_1} \tau.P_{n+2} & \Rightarrow & P_{n+1}\sigma \succ_{K_1'} \tau.P_{n+2}' & \begin{pmatrix} K_1' \lhd\!\rhd K_1\sigma, \\ P_{n+2}' =_\alpha P_{n+2}\sigma) \end{pmatrix} \\
\vdots & & & \\
P_{n+m} \succ_{K_m} \tau.P_{n+m+1} & \Rightarrow & P_{n+m}\sigma \succ_{K_m'} \tau.P_{n+m+1}' & \begin{pmatrix} K_m' \lhd\!\rhd K_m\sigma, \\ P_{n+m+1}' =_\alpha P_{n+m+1}\sigma \end{pmatrix}
\end{array}$$

Reading the second column, bearing in mind that $P_i\sigma =_\alpha P_i'$, we see that for $Q' = (\lambda\vec{x})P_{n+m+1}'$, we have the desired result $P\sigma \succ\!\!\succ_{M'} \alpha'.Q'$, where $Q' =_\alpha Q\sigma$, $\alpha' = \alpha\sigma$, and $M' \lhd\!\rhd M\sigma$, where $M' = N_1' \cdots N_n' \cdot L \cdot K_1' \cdots K_m'$.

The other cases, where α is an output or silent action, are similar.

CLAUSE 2: For variety, we choose to study the case where α is an output action. Then $P\sigma \succ\!\!\succ_{M'} \alpha'.Q'$ means

$$P\sigma \succ_{N'_1} \tau.P'_1 \succ_{N'_2} \tau \cdots \tau.P'_n \succ_{L'} \alpha'.(\boldsymbol{\nu}\vec{x})[\vec{y}]P'_{n+1}$$

$$P'_{n+1} \succ_{K'_1} \tau \cdots \succ_{K'_m} \tau.P'_{n+m+1}$$

where $Q' = (\boldsymbol{\nu}\vec{x})[\vec{y}]P'_{n+m+1}$, for $n, m \geq 0$, where $M' = N'_1 \cdots N'_n \cdot L' \cdot K'_1 \cdots K'_m$ and $\mathrm{n}(K'_1 \cdots K'_m) \cap \vec{x} = \emptyset$, and we can complete the following table, this time using Lemma 2.17(2) for the implications:

$$\begin{array}{lll}
P\sigma \succ_{N'_1} \tau.P'_1 & \Rightarrow\ P \succ_{N_1} \tau.P_1 & (N'_1 \lhd\!\rhd N_1\sigma, P'_1 =_\alpha P_1\sigma) \\
P'_1 \succ_{N'_2} \tau.P'_2 & \Rightarrow\ P_1 \succ_{N_2} \tau.P_2 & (N'_2 \lhd\!\rhd N_2\sigma, P'_2 =_\alpha P_2\sigma) \\
\vdots & & \\
P'_{n-1} \succ_{N'_n} \tau.P'_n & \Rightarrow\ P_{n-1} \succ_{N_n} \tau.P_n & (N'_n \lhd\!\rhd N_n\sigma, P'_n =_\alpha P_n\sigma) \\
P'_n \succ_{L'} \alpha'.(\boldsymbol{\nu}\vec{x})[\vec{y}]P'_{n+1} & \Rightarrow\ P_n \succ_L \alpha.(\boldsymbol{\nu}\vec{x})[\vec{y}]P_{n+1} & \\
 & \multicolumn{2}{r}{\begin{pmatrix} L' \lhd\!\rhd L\sigma, \alpha' = \alpha\sigma \\ (\boldsymbol{\nu}\vec{x})[\vec{y}]P'_{n+1} =_\alpha (\boldsymbol{\nu}\vec{x})[\vec{y}]P_{n+1}\sigma \end{pmatrix}} \\
\vdots & & \\
P'_{n+m} \succ_{K'_m} \tau.P'_{n+m+1} & \Rightarrow P_{n+m} \succ_{K_m} \tau.P_{n+m+1} & \begin{pmatrix} K'_m \lhd\!\rhd K_m\sigma, \\ P'_{n+m+1} =_\alpha P_{n+m+1}\sigma \end{pmatrix}
\end{array}$$

Thus, reading the second column, for $Q = (\boldsymbol{\nu}\vec{x})[\vec{y}]P_{n+m+1}$ and $n, m \geq 0$, we have that $P \succ\!\!\succ_M \alpha.Q$, where $Q' =_\alpha Q\sigma$, $\alpha' = \alpha\sigma$, and $M' \lhd\!\rhd M\sigma$, where $M = N_1 \cdots N_n \cdot L \cdot K_1 \cdots K_m$.

Again, the other cases are similar. □

Lemma 2.32 *$\lesseqgtr_D$ is closed under respectful substitutions, i.e., $P \lesseqgtr_D Q$ implies $P\sigma \lesseqgtr_{D\sigma} Q\sigma$, for any σ which respects D.*

Proof: We show that

$$\mathcal{R} = \{\{(P\sigma, Q\sigma) : P \lesseqgtr_D Q\}_{D\sigma} : \sigma \text{ respects } D\}$$

is an indexed conditional weak open bisimulation (Definition 2.26).

Assume $P\rho\ \mathcal{S}_{D\rho}\ Q\rho$ for some respectful substitution ρ. If $P\rho$ has a commitment, by Lemma 2.17(2) we can write this (modulo $\lhd\!\rhd$ between match sequences) as $P\rho \succ_{M\rho} \alpha\rho.P'\rho$, for some M, α and P' such that $P \succ_M \alpha.P'$. Since $P\ \mathcal{S}_D\ Q$, we know that $Q \succ\!\!\succ_N \beta.Q'$, where

$$M \rhd N \tag{2.1}$$

$$\alpha\sigma_M = \beta\sigma_M \tag{2.2}$$

$$P'\sigma_M \lesseqgtr_{D\sigma_M} Q'\sigma_M \tag{2.3}$$

Using Lemma 2.31(1), this gives $Q\rho \succ\!\!\succ_{N\rho} \beta\rho.Q'\rho$, which matches the commitment of $P\rho$ if

$$M\rho \rhd N\rho \tag{2.4}$$

$$\alpha\rho\sigma_{M\rho} = \beta\rho\sigma_{M\rho} \tag{2.5}$$

$$P'\rho\sigma_{M\rho}\ \mathcal{S}_{D'}\ Q'\rho\sigma_{M\rho} \tag{2.6}$$

where $D' = D\rho\sigma_{M\rho}$.

By Lemma 2.11(1), (2.4) follows from $M \rhd N$. For (2.5) and (2.6) we first express $\rho\sigma_{M\rho}$ in terms of σ_M. As Sangiorgi shows, $\rho\sigma_{M\rho} = \sigma_M\gamma$ for some γ, and thus what we want to show is that $\alpha\sigma_M\gamma = \beta\sigma_M\gamma$ (which is clear from $\alpha\sigma_M = \beta\sigma_M$), and that $P'\sigma_M\gamma\ \mathcal{S}_{D\sigma_M\gamma}\ Q'\sigma_M\gamma$, which is true from 2.3 and the construction of $\mathcal{R}$.

Let us turn to the case where $P\rho$ is a concretion. We can write this concretion in standard form as $\big((\boldsymbol{\nu}\vec{x})[\vec{y}]P'\big)\rho$, which can be rewritten as $(\boldsymbol{\nu}\vec{x})[\vec{y}\rho]P'\rho$, since w.l.o.g.[1] $\vec{x}\rho = \rho$. We know that $P \succapprox_D Q$, and thus that Q has standard form $(\boldsymbol{\nu}\vec{x})[\vec{y}]Q'$, and $P' \succapprox_{D'} Q'$ where $D' = D \cup \{\vec{x} \times \mathrm{fn}(P,Q)\}$.

Then $Q\rho$ has standard form $(\boldsymbol{\nu}\vec{x})[\vec{y}\rho]Q'\rho$, and we want to show that $P'\rho\ \mathcal{S}_{D''}\ Q'\rho$ where $D'' = D\rho \cup \{\vec{x} \times \mathrm{fn}(P\rho, Q\rho)\}$. D'' can be rewritten as

$$\begin{aligned} D\rho \cup \{\vec{x} \times \mathrm{fn}(P,Q)\rho\} &= \\ &= D\rho \cup \{\vec{x} \times \mathrm{fn}(P,Q)\}\rho \\ &= (D \cup \{\vec{x} \times \mathrm{fn}(P,Q)\})\rho \\ &= D'\rho \end{aligned}$$

that is, what we want to show is $P'\rho\ \mathcal{S}_{D'\rho}\ Q'\rho$ which is true because $P' \succapprox_{D'} Q'$ and because of the construction of $\mathcal{R}$.

The case where P and Q are abstractions is simpler. □

Lemma 2.33 *If $P\sigma_M \succ\!\!\succ \alpha.Q'$ then $P \succ\!\!\succ_N \beta.Q$ with*

- $M \rhd N$
- $\alpha = \beta\sigma_M$
- $Q' =_\alpha Q\sigma_M$

PROOF: In the case where α is an input action, $P\sigma_M \succ\!\!\succ \alpha Q'$ means

$$P\sigma_M \succ \tau.P'_1 \succ \cdots \succ \tau.P'_n \succ \alpha.(\lambda\vec{x})P'_{n+1}$$
$$P'_{n+1} \succ \tau \cdots \tau.P'_{n+m} \succ \tau.P'_{n+m+1}$$

where $Q' \equiv (\lambda\vec{x})P'_{n+m+1}$, for $n, m \geq 0$.

Again using the strong version of this lemma (2.19), we can make a nice table. Note however, that the match sequence N in the table is the condition for the *first* commitment, not necessary on a τ commitment as in this table. In each step after the first, we use Lemma 2.17(2) to connect with the previous step. Note also that we omit the empty substitutions $\sigma_\emptyset$.

$P\sigma_M \succ \tau.P'_1$	$\stackrel{2.19}{\Rightarrow}\ P \succ_N \tau.P_1$	$M \rhd N, P'_1 =_\alpha P_1\sigma_M$
$P'_1 \succ \tau.P'_2$	$\stackrel{2.19}{\Rightarrow}\ P'_1 \succ_\emptyset \tau.P_2$	$P'_2 =_\alpha P_2$
	$\Rightarrow\ P_1\sigma_M \succ_\emptyset \tau.P_2$	(since $P'_1 =_\alpha P_1\sigma_M$)
	$\stackrel{2.17(2)}{\Rightarrow}\ P_1 \succ_\emptyset \tau.P''_2$	$P''_2\sigma_M =_\alpha P_2 =_\alpha P'_2$
$P'_2 \succ \tau.P'_3$	$\stackrel{2.19}{\Rightarrow}\ P'_2 \succ_\emptyset \tau.P_3$	$P'_3 =_\alpha P_3$
	$\Rightarrow\ P''_2\sigma_M \succ_\emptyset \tau.P_3$	$(P'_2 =_\alpha P''_2\sigma_M)$
	$\stackrel{2.17(2)}{\Rightarrow}\ P''_2 \succ_\emptyset \tau.P''_3$	$P''_3\sigma_M =_\alpha P_3 =_\alpha P'_3$
$\vdots$		
$P'_n \succ \alpha.F'$	$\stackrel{2.19}{\Rightarrow}\ P'_n \succ_\emptyset \gamma.F$	$F' =_\alpha F, \gamma = \alpha$
	$\Rightarrow\ P''_n\sigma_M \succ_\emptyset \gamma.F$	$(P'_n =_\alpha P''_n\sigma_M)$
	$\stackrel{2.17(2)}{\Rightarrow}\ P''_n \succ_\emptyset \beta.F''$	$F''\sigma_M =_\alpha F =_\alpha F'$,
		$\beta\sigma_M = \gamma = \alpha$
		$F \equiv (\lambda\vec{x})P_{n+1}$
$\vdots$		
$P'_{n+m} \succ \tau.P'_{n+m+1}$	$\stackrel{2.19}{\Rightarrow}\ P'_{n+m} \succ_\emptyset \tau.P_{n+m+1}$	$P'_{n+m+1} =_\alpha P_{n+m+1}$
	$\Rightarrow\ P_{n+m}\sigma_M \succ_\emptyset \tau.P_{n+m+1}$	$(P'_{n+m} =_\alpha P_{n+m}\sigma_M)$
	$\stackrel{2.17(2)}{\Rightarrow}\ P_{n+m} \succ_\emptyset \tau.P''_{n+m+1}$	$P''_{n+m+1}\sigma_M =_\alpha P_{n+m+1}$
		$P_{n+m+1} =_\alpha P'_{n+m+1}$

[1] "without loss of generality"

Thus, by connecting each step in the second column with the next, we get the desired result: $P \succ\!\!\succ_N \beta.Q$, where $Q \equiv (\lambda\vec{x})P_{n+m+1}$, $Q' =_\alpha Q\sigma_M$, $\alpha = \beta\sigma_M$, and $M \rhd N$.

The other cases are similar. □

Lemma 2.34 *If $P \succ\!\!\succ_M \alpha.P'$ then $P\sigma_M \succ\!\!\succ \alpha\sigma_M.P'\sigma_M$.*

Proof: Looking again at the case where α is an input action, we by now know that $P \succ\!\!\succ_M \alpha.P'$ means

$$P \succ_{N_1} \tau.P_1 \succ_{N_2} \tau.P_2 \succ_{N_3} \cdots \tau.P_n \succ_L \alpha.(\lambda\vec{x})P_{n+1}$$

$$P_{n+1} \succ_{K_1} \tau.P_{n+2} \cdots \tau.P_{n+m} \succ_{K_m} \tau.P_{n+m+1}$$

where $P' = (\lambda\vec{x})P_{n+m+1}$, for $n, m \geq 0$, but let us just point out that $M = N_1 \cdots N_n \cdot L \cdot K_1 \cdots K_m$, and thus $M \rhd N_i$, $M \rhd L$, and $M \rhd K_i$, for each i. Using the strong version of this lemma (2.20), and Lemmas 2.21 and 2.11(3), we can complete the following table:

$$\begin{array}{lcl}
P \succ_{N_1} \tau.P_1 & \stackrel{2.20}{\Rightarrow} & P\sigma_{N_1} \succ \tau.P_1\sigma_{N_1} \\
 & \stackrel{2.21}{\Rightarrow} & P\sigma_{N_1}\rho_1 \succ \tau.P_1\sigma_{N_1}\rho_1 \\
 & \stackrel{2.11(3)}{\Rightarrow} & P\sigma_M \succ \tau.P_1\sigma_M \text{ (since } M \rhd N_1) \\
P_1 \succ_{N_2} \tau.P_2 & \stackrel{2.20}{\Rightarrow} & P_1\sigma_{N_1} \succ \tau.P_2\sigma_{N_2} \\
 & \stackrel{2.21}{\Rightarrow} & P_1\sigma_{N_2}\rho_2 \succ \tau.P_2\sigma_{N_2}\rho_2 \\
 & \stackrel{2.11(3)}{\Rightarrow} & P_1\sigma_M \succ \tau.P_2\sigma_M \text{ (since } M \rhd N_2) \\
\vdots & & \\
P_n \succ_L \alpha.F & \stackrel{2.20}{\Rightarrow} & P_n\sigma_L \succ \alpha\sigma_L.F\sigma_L \\
 & \stackrel{2.21}{\Rightarrow} & P_n\sigma_L\rho_n \succ \alpha\sigma_L\rho_n.F\sigma_L\rho_n \\
 & \stackrel{2.11(3)}{\Rightarrow} & P_n\sigma_M \succ \alpha\sigma_M.F\sigma_M \text{ (since } M \rhd L) \\
\Big(F \equiv (\lambda\vec{x})P_{n+1} & \stackrel{(\star)}{\Rightarrow} & F\sigma_M \equiv (\lambda\vec{x})(P_{n+1}\sigma_M)\Big) \\
P_{n+1} \succ_{K_1} \tau.P_{n+2} & \stackrel{2.20}{\Rightarrow} & P_{n+1}\sigma_{K_1} \succ \tau.P_{n+2}\sigma_{K_1} \\
 & \stackrel{2.21}{\Rightarrow} & P_{n+1}\sigma_{K_1}\rho_{n+1} \succ \tau.P_{n+2}\sigma_{K_1}\rho_{n+1} \\
 & \stackrel{2.11(3)}{\Rightarrow} & P_{n+1}\sigma_M \succ \tau.P_{n+2}\sigma_M \text{ (since } M \rhd K_1) \\
\vdots & & \\
P_{n+m} \succ_{K_m} \tau.P_{n+m+1} & \stackrel{2.20}{\Rightarrow} & P_{n+m}\sigma_{K_m} \succ \tau.P_{n+m+1}\sigma_{K_m} \\
 & \stackrel{2.21}{\Rightarrow} & P_{n+m}\sigma_{K_m}\rho_{n+m} \succ \tau.P_{n+m+1}\sigma_{K_m}\rho_{n+m} \\
 & \stackrel{2.11(3)}{\Rightarrow} & P_{n+m}\sigma_M \succ \tau.P_{n+m}\sigma_M \text{ (since } M \rhd K_m)
\end{array}$$

Finally, by connecting the steps in the second column, we get $P\sigma_M \succ\!\!\succ \alpha\sigma_M.P'\sigma_M$ as desired.

The interesting step here is the $\star$-marked implication, which is only valid if (as we assume on page 10) σ_M is an identity on $\vec{x}$ (otherwise we alpha-convert the term). So this is why we needed the constraint $\mathrm{n}(K_1 \cdots K_m) \cap \vec{x} = 0$; otherwise, the alternative semantics could make weak commitments resulting in *different* derivatives under the accumulated condition M than the standard semantics could under the corresponding substitution σ_M.

As usual, the cases where α is not an input action are similar. □

Theorem 2.35 $\approx_D$ *implies* $\approxeq_D$

Proof: Following Sangiorgi [San93b], we show that

$$\mathcal{R} = \{\{(P, Q) : P\mathcal{S}_D Q\}_D : \mathcal{S}_D \in \mathcal{T}\}$$

is a weak conditional open D-bisimulation, where $\mathcal{T}$ is the weak open D-bisimulation witnessing $P \approx_D Q$.

Suppose that $P \succ_M \alpha.P'$. By Lemma 2.20 we know that $P\sigma_M \succ \alpha\sigma_M.P'\sigma_M$, and since $P \approx_D Q$, using Proposition 2.30 and the fact that σ_M respects D (since we know that M does), we have that

$$P\sigma_M \approx_{D\sigma_M} Q\sigma_M$$

and thus $Q\sigma_M \ggcurly \beta.Q''$, where $\alpha\sigma_M = \beta$, $P'\sigma_M \approx_{D\sigma_M} Q''$. By Lemma 2.33, we have $Q \ggcurly_N \gamma.Q'$ with $M \rhd N$, $\gamma\sigma_M = \beta$, and $Q'\sigma_M =_\alpha Q''$. Thus $Q \ggcurly_N \gamma.Q'$ with

- $M \rhd N$
- $\gamma\sigma_M = \alpha\sigma_M$
- $P'\sigma_M \; \mathcal{S}_{D\sigma_M} \; Q'\sigma_M$

which is what is required of a weak conditional open D-bisimulation.

Now suppose that P is a concretion with standard form $(\boldsymbol{\nu}\vec{x})[\vec{y}]P'$. Then, since $P \approx_D Q$, Q has standard form $(\boldsymbol{\nu}\vec{x})[\vec{y}]Q'$, and $P' \approx_{D'} Q'$, where $D' = D \cup \{\vec{x} \times \mathrm{fn}(P,Q)\}$, which is the correct D' for the definition of weak conditional open D-bisimulation.

The case where P is an abstraction is similar. $\square$

Theorem 2.36 $\lessapprox_D$ *implies* $\approx_D$

PROOF: Again following Sangiorgi, we show that

$$\mathcal{S} = \{\{(P,Q) : P\mathcal{S}_D Q\}_D : \mathcal{S}_D \in \mathcal{T}\}$$

is a weak open D-bisimulation, where $\mathcal{T}$ is the weak conditional open D-bisimulation witnessing $P \lessapprox_D Q$.

By Lemma 2.32, $\mathcal{S}$ is closed under respectful substitutions, so by Proposition 2.29, we only need to show that $\mathcal{S}$ is an indexed weak ground bisimulation.

Suppose $P \succ \alpha.P'$. By Lemma 2.19, $P \succ_\emptyset \alpha.P'$, and since $P \lessapprox_D Q$, we have that $Q \ggcurly_\emptyset \beta.Q'$, where $\alpha = \beta$ and $P' \lessapprox_D Q'$. By Lemma 2.34, we know that $Q \ggcurly \beta.Q'$, which closes up the bisimulation.

Now suppose P is a concretion with standard form $(\boldsymbol{\nu}\vec{x})[\vec{y}]P'$. Because $P \lessapprox_D Q$, we know that Q has standard form $(\boldsymbol{\nu}\vec{x})[\vec{y}]Q'$ and $P' \lessapprox_{D'} Q'$, where $D' = D \cup \{\vec{x} \times \mathrm{fn}(P,Q)\}$, which is the appropriate distinction for an indexed weak ground bisimulation.

The case where P is an abstraction is similar. $\square$

This concludes the proof of Theorem 2.27. We note in passing that the monadic calculus is a subcalculus of the polyadic, so the theorem holds in the monadic calculus as well.

Chapter 3

The Algorithm

The usual technique for deciding bisimilarity of (finite-state) systems is to construct the state spaces of the two systems in question and then perform a partition refinement algorithm to try to distinguish the two start states. However, this technique is inapplicable in the case of the π-calculus due to the problems of name instantiation.

When dealing with the polyadic π-calculus we need a finer notion than state spaces; we consider "agent spaces", which is the set of agents (processes, abstractions and concretions) we need to examine when checking the equivalence.

For example, the two terms

$$\begin{aligned} P &\stackrel{def}{=} (\lambda x)x.(\lambda y)\overline{y}.[y]P(y) \\ Q &\stackrel{def}{=} (\lambda x)x.(\lambda y)\Big(\overline{y}.[y]Q(y) \;+\; [x=y]\overline{y}.[y]Q(y)\Big) \end{aligned}$$

are clearly open bisimilar. When comparing them for equality, with the open equivalence it is enough to instantiate the name bound in an abstraction or restriction with a single fresh name, in contrast with the late and early equivalences, where it is necessary to instantiate with each free name plus one fresh name (a name x is *fresh* with respect to an abstraction $(\lambda z)A$ or a restriction $(\boldsymbol{\nu} z)A$ if $x \notin \mathrm{fn}(A) - \{z\}$).

We cannot generate the complete agent space of P or Q, since they are infinite, but by equating alpha-equivalent agents we can generate a minimal agent space. When generating the minimal agent space for P, we get five agents, namely P, $P(x)$ (by instantiating x with x), $(\lambda y)\overline{y}.[y]P(y)$, $\overline{x}.[x]P(x)$ (by instantiating y with x), and $[x]P(x)$. However, when we compare the two agents, we cannot simply compare their minimal agent spaces, because when generating the minimal agent space of Q, the second abstraction of Q cannot instantiate y to x, due to the appearance of x in the ensuing agent. To match the two, the agent space of P must be extended to match that of Q.

Thus the implementation of the bisimulation algorithm is by necessity an *"on-the-fly"* algorithm [FM91]: the agent spaces of the two systems in question are generated together during the construction of the candidate bisimulation relation which equates them.

Beyond this, the algorithms implemented in the MWB closely follow Definitions 2.12 and 2.26 (the strong and weak versions): given two agents P and Q in standard form and a relation $\mathcal{R}$, check if (P, Q) is already in the relation. If so, return the relation unchanged. Otherwise, if P and Q are processes, then for each commitment of P, find a (strong or weak) commitment of Q such that the conditions match appropriately and the actions are equivalent under the substitution σ induced by the first, larger, condition. Assuming that P and Q are equivalent (by adding (P, Q) to the relation), apply the substitution σ to the derivatives and recurse over them using the extended relation. This either fails, which causes the current recursion to try the next commitment of Q (or to fail if no such commitment exists), or returns a relation relating the two derivatives, which is used in subsequent recursive calls to match the other commitments of P. Finally, when all commitments of P have been matched by Q, match each commitment of Q with a (strong or weak) commitment of P in the same way, returning the resulting relation. If P and Q are abstractions of the same arity, instantiate the same bound names (alpha-converting the bodies of the abstractions), and recurse over the bodies of the abstractions. If P and Q are concretions of the same arity, then if they are restricted instantiate the same names, and if the concretions match, recurse over the bodies of the abstractions; otherwise fail.

(This is the approach in the absence of distinctions; in the more general case, distinctions are handled in a suitable fashion; see Figure 3.1.)

The algorithm is described in more detail in Figure 3.1, where we present pseudo-code for the algorithm for weak equivalence. For the strong equivalence, simply change the framed occurrences of Pcw and Qcw to Pc and Qc respectively. In Appendix A we prove that the algorithm is correct.

The implementation of the algorithm is of course somewhat more complicated than presented in Figure 3.1, but not much. An extra optimization is provided by the addition of an inequality set (inspired by [FM91]), which records all triples (P, Q, D) for which we have already found that $P \not\approx_D Q$, and thus we do not need to check them for equivalence. Of course we also make sure that we record new triples in a standard order, so that we do not add (Q, P, D) if the symmetric case (P, Q, D) is already in R.

The problem of deciding bisimulation equivalence of data-independent agents in value-passing CCS is known to be NP-hard in the size of the agents [JP93], which implies that the decision problem for π-calculus is at least this hard.

$$
\begin{array}{l}
\mathit{bisimilar}(P,Q)\ D = (\mathit{add}(P,Q,D)\ \emptyset) \neq \emptyset \\
\\
\mathit{add}(P,Q,D)\ R = \\
\qquad \text{if}\ \ \exists(P',Q',D') \in R \text{ such that } P = P' \wedge Q = Q' \wedge D' \subseteq D \\
\qquad\quad \text{then return } \{(P,Q,D)\} \cup R \\
\qquad \text{else if is\_process}(P) \wedge \text{is\_process}(Q) \text{ then} \\
\qquad\qquad \mathit{match}\ Qc\ \boxed{Pcw}\ D \\
\qquad\qquad\qquad (\mathit{match}\ Pc\ \boxed{Qcw}\ D\ (\{(P,Q,D)\} \cup R)) \\
\qquad\qquad \text{where } Pc = \big\{(m,\alpha,P') \mid P \succ_m \alpha.P' \wedge m \text{ respects } D\big\} \\
\qquad\qquad\qquad Qc = \big\{(m,\alpha,Q') \mid Q \succ_m \alpha.Q' \wedge m \text{ respects } D\big\} \\
\qquad\qquad\qquad Pcw = \big\{(m,\alpha,P') \mid P \ggcurly_m \alpha.P' \wedge m \text{ respects } D\big\} \\
\qquad\qquad\qquad Qcw = \big\{(m,\alpha,Q') \mid Q \ggcurly_m \alpha.Q' \wedge m \text{ respects } D\big\} \\
\qquad \text{else if}\ \ P \equiv (\lambda\vec{x})P' \wedge Q \equiv (\lambda\vec{y})Q' \wedge |\vec{x}| = |\vec{y}| \\
\qquad\qquad \text{then } \mathit{add}(P'',Q'',D)\ (\{(P,Q,D)\} \cup R) \\
\qquad\qquad\qquad \text{where } \vec{n} = \text{freshnames}(|\vec{x}|, fn(P,Q)) \\
\qquad\qquad\qquad\qquad P'' = P'\{\vec{n}/\vec{x}\} \\
\qquad\qquad\qquad\qquad Q'' = Q'\{\vec{n}/\vec{y}\} \\
\qquad \text{else if}\ \ P \equiv (\boldsymbol{\nu}\vec{x})[\vec{y}]P' \wedge Q \equiv (\boldsymbol{\nu}\vec{z})[\vec{w}]Q' \wedge |\vec{x}| = |\vec{z}| \wedge \vec{y}\{\vec{n}/\vec{x}\} \equiv \vec{w}\{\vec{n}/\vec{z}\} \\
\qquad\qquad \text{then } \mathit{add}(P'',Q'',D')\ (\{(P,Q,D)\} \cup R) \\
\qquad\qquad\qquad \text{where } \vec{n} = \mathit{freshnames}(|\vec{x}|, fn(P,Q)) \\
\qquad\qquad\qquad\qquad P'' = P'\{\vec{n}/\vec{x}\} \\
\qquad\qquad\qquad\qquad Q'' = Q'\{\vec{n}/\vec{z}\} \\
\qquad\qquad\qquad\qquad D' = \big(D\lceil\text{fn}(P,Q)\big) \cup \{\vec{n} \times \text{fn}(P,Q)\} \\
\qquad \text{else return } \emptyset \\
\\
\mathit{match}\ Pc\ Qc\ D\ \emptyset = \emptyset \\
\mathit{match}\ \emptyset\ Qc\ D\ R = R \\
\mathit{match}\ Pc\ \emptyset\ D\ R = \emptyset \\
\mathit{match}\ (\{(m,\alpha,P)\} \cup Pc)\ Qc\ D\ R = \\
\qquad \text{find}\ \ (n,\beta,Q) \in Qc \text{ such that} \\
\qquad\qquad m \rhd n \wedge \alpha\sigma_m = \beta\sigma_m \\
\qquad\qquad \wedge((\mathit{add}(P\sigma_m, Q\sigma_m, D\sigma_m)\ R) \neq \emptyset) \\
\qquad\qquad \text{and return } \mathit{match}\ Pc\ Qc\ D\ (\mathit{add}(P\sigma_m, Q\sigma_m, D\sigma_m)\ R) \\
\qquad \text{else return } \emptyset \\
\\
\mathit{freshnames}(n,S) \text{ returns } n \text{ names not in the set } S
\end{array}
$$

Figure 3.1: The verification algorithm (weak case)

Chapter 4

The Implementation

The MWB is implemented in Standard ML [MTH90], and currently runs under the New Jersey SML compiler (version 0.93). The command parser is implemented using SML-Lex and SML-Yacc. The code for the whole tool is currently approximately 3800 lines (a bit more than in Figure 3.1, admittedly), in 16 different modules.

When the first implementation of the MWB was done [VM94], very little attention was paid to efficiency. π-calculus names were implemented by the ASCII strings (e.g the name x was represented by the string `"x"`), and lots of α-conversion was done, among other things resulting in a lot of copying of π-calculus terms. The transitions of agents were computed every time they were needed, which was very costly for complex agents. Much of this inefficiency has since been overcome.

4.1 Representation

In the present implementation, we use de Bruijn style indices [dB72] to represent names, thereby removing the need for α-conversion. The encoding of the polyadic π-calculus and some examples of its use are described below.

Every name is represented by an index (a number) representing its *reference depth*, saying how far it is to its binding. Binders (λ and $\boldsymbol{\nu}$) no longer take arguments, but only indicate that a binding is taking place. For example, the agent $(\lambda x)(\boldsymbol{\nu} y)\overline{x}\,.[y]\mathbf{0}$ is represented as $\lambda\boldsymbol{\nu}\overline{1}\,.[0]\mathbf{0}$. All alpha-equivalent agents have the same representation; e.g., $(\lambda a)(\boldsymbol{\nu} b)\overline{a}\,.[b]\mathbf{0}$ also has the representation $\lambda\boldsymbol{\nu}\overline{1}\,.[0]\mathbf{0}$. Below we will use double square brackets $[\![x]\!]$ to mean "the representation of x".

A little more complicated is the example

$$[\![(\lambda a)(\lambda b)a.(\lambda x)b.(\lambda y)[x = y]a.[b]\mathbf{0}]\!] = \lambda\lambda 1.\lambda 1.\lambda[1 = 0]3.[2]\mathbf{0}$$

An occurrence of a name is *free* in an agent if its reference depth is greater than or equal to its level (the number of binding operators surrounding the occurrence). For example, the name c is free in

$$P \stackrel{def}{=} (\lambda a)a.(\lambda b)[b = c]\overline{b}\,.[a]\mathbf{0}$$

and the index in its representation (framed for convenience) is equal to the number of binding operators (2) surrounding it:

$$[\![P]\!] = \lambda 0.\lambda[0 = \boxed{2}]\overline{0}\,.[1]\mathbf{0}$$

Names which are free on the top level are represented in correspondence to this, using an ordered list of names as a "global binding". So the first free name gets represented by 0 (at top level), the second by 1, etc.

With this representation, much is gained since we no longer have to do alpha-conversion. However, care must be taken when implementing the various syntactic rewrite rules. For example, consider parallel composition of abstractions (Definition 2.5)

$$\big((\lambda\vec{x})P \mid (\lambda\vec{y})Q\big) \text{ rewrites to } (\lambda\vec{x}\vec{y})\big(P \mid Q\big)$$

provided $\vec{x} \not\subseteq \mathrm{n}(Q) \cup \vec{y}$ and $\vec{y} \not\subseteq \mathrm{n}(P) \cup \vec{x}$. Using the "naïve" representation, we may have to alpha-convert one of the agents to avoid name clashes. Using de Bruijn indices however, this becomes

$$\big((\lambda)^n P \mid (\lambda)^m Q\big) \text{ rewrites to } (\lambda)^{n+m}\big(P' \mid Q'\big)$$

where P' and Q' are P and Q with the indices changed appropriately: in P' the indices corresponding to names in $\vec{x}$ are increased by m, since the distance to their binding has increased by that amount, and also the indices to names free in $(\lambda\vec{x})P$ are increased by the same amount. In Q' however, only the indices corresponding to names free in $(\lambda\vec{y})Q$ should be increased (by n).

Another situation is the application of the RES rule:

$$\text{RES}: \quad \frac{P \succ_M \alpha.P'}{(\boldsymbol{\nu}b)P \succ_M \alpha.(\boldsymbol{\nu}b)P'} \quad b \notin \mathrm{n}(\alpha, M)$$

In the naïve representation, we can simply compare b and $\mathrm{n}(\alpha, M)$, but how do we do this in our de Bruijn representation, where we no longer have the name of b? Consider the agent $[\![(\boldsymbol{\nu}x)\overline{x}\,.[x]\mathbf{0}]\!] = \boldsymbol{\nu}\overline{0}\,.[0]\mathbf{0}$, which should have no commitment by the RES rule. When applying the rule, we compute the commitments of the "body" of the restriction, which is the single commitment $\succ_\emptyset \overline{0}\,.[0]\mathbf{0}$. When computing the commitment of the restriction, because we have looked "inside" a binding, we have to reduce the indices of the names of the action and match sequence coming out of this, since the distance to their bindings is one less outside the $\boldsymbol{\nu}$. In this case, the reduced index of the name of the commitment becomes -1, which is easily recognizable, indicating that the commitment is not valid. Compare this to the (complete) commitment of $[\![(\boldsymbol{\nu}x)\overline{y}\,.[x]\mathbf{0}]\!] = \boldsymbol{\nu}\overline{1}\,.[0]\mathbf{0}$ which is $\succ_\emptyset \overline{0}\,.\boldsymbol{\nu}[0]\mathbf{0}$, a perfectly legal commitment.

4.2 Recording of commitments

Computing the commitments of a π-calculus process can be quite costly if the structure of the process is complex. To reduce the amount of work we have to do, we use the ALPHA rule from the efficient commitment relation (Figure 2.3), which tells us that alpha equivalent agents have the same commitments. Since all alpha equivalent agents have the same representation when we use our de Bruijn representation, checking for alpha equivalence is easy and efficient, and we can use the ALPHA rule to its fullest.

Every time we compute the commitments of a process, we record them in a hash table. Whenever we want to find the commitments of a process, we first look up the process in the hash table, and only if there are no recorded commitments do we actually go to the trouble of computing them. The hash tables are at present kept at most 30% full, and automatically resize and rehash when they become fuller.

Using preliminary measurements, this is where the big time saving comes from, compared to the original implementation of the MWB. In the original implementation, verification of the handover procedure of [OP92] ran for over 230 CPU hours on a SPARCstation 10 before it was (prematurely) terminated, while in the new implementation, it ran to completion in less than 11 CPU *minutes*!

Chapter 5

Examples

5.1 A Simple Example

In Figure 5.1 we have a sample session which demonstrates some simple usage. Note that we render $\boldsymbol{\nu}$ as `^`, λ as `\`, and $\overline{x}$ as `'x` when typing in ASCII format, and that the parser accepts input both in the "traditional" style where action prefixes look like `a(x)` or `'x<b>`, as well as a style with explicit abstractions and concretions `a.(\x)` or `'x.[b]`.

In the sample session, we first define an agent `Buf1` implementing a one-place buffer, then another, `Buf2`, implementing a two-place buffer by composing two instances of `Buf1`, and finally three agents, `Buf20`, `Buf21` and `Buf22`, together implementing a two-place buffer without parallel composition.

We proceed with this example by comparing the two implementations for weak equality. The MWB responds by saying that they are equivalent and that it found a bisimulation relation with 18 tuples, and asks us if we want to inspect it. We respond positively and the MWB prints out the relation as a list of pairs of agents with associated distinction sets.

We then simulate the behaviour of the agent `Buf2(i,o)`. The MWB presents the possible commitments, including their least necessary conditions (if not trivial), and prompts the user to select one of them. When the user selects a commitment whose derivative is an abstraction or concretion, the bound names are instantiated automatically to the least index available. After having a single choice on the first two steps, we then get a choice of three commitments; the first which is possible only if the names `i` (represented by 0) and `o` (represented by 1) are the same.

Next, we change the definition of `Buf22` to introduce a possible deadlock and again check for weak equivalence between `Buf2` and `Buf20`, this time as abstractions, without instantiating their arguments. We find that they are not equivalent, and proceed by trying to equate `Buf2(i,o)` and `Buf20(i,o)` under the proviso that `i` is different from all other free names of the two agents (namely `o`). Under this distinction, there are no deadlocks, and the MWB reports that they are once again equivalent.

5.2 An Extended Example: Mobile Telephones

As a case study, we have specified and verified the core of the handover protocol intended to be used in the GSM Public Land Mobile Network (PLMN) proposed by the European Telecommunication Standards Institute (ETSI). The formal specification of the protocol, and its service specification, are due to

```
The Mobility Workbench
(Polyadic version 0.97)

MWB> agent Buf1(i,o) = i(x).'o<x>.Buf1(i,o)
MWB> agent Buf2(i,o) = (^m)(Buf1(i,m) | Buf1(m,o))
MWB> agent Buf20(i,o) = i(x).Buf21(i,o,x)
MWB> agent Buf21(i,o,x) = i(y).Buf22(i,o,x,y) + 'o<x>.Buf20(i,o)
MWB> agent Buf22(i,o,x,y) = 'o<x>.Buf21(i,o,y)

MWB> weq Buf2(i,o) Buf20(i,o)
The two agents are related.
Relation size = 18.  Do you want to see it?  (y or n) y
R= < (^)(_1.(\)'_1.[_0]Buf1<_2,_1> | _0.(\)'_3.[_0]Buf1<_1,_3>),
      _0.(\)Buf21<_1,_2,_0> >  {}
      ...

MWB> step Buf2(i,o)
   0:  |>_0.(\)(^)('_0.[_1]Buf1<_2,_0> | _0.(\)'_4.[_0]Buf1<_1,_4>)
Step> 0
  (\_2)
   0:  |>t.(^)(_1.(\)'_1.[_0]Buf1<_2,_1> | '_2.[_3]Buf1<_0,_2>)
Step> 0
   0:  |[_0=_1]>t.(^)('_0.[_3]Buf1<_1,_0> | _0.(\)'_3.[_0]Buf1<_1,_3>)
   1:  |>_0.(\)(^)('_0.[_1]Buf1<_2,_0> | '_3.[_4]Buf1<_0,_3>)
   2:  |>'_1.[_2](^)(_1.(\)'_1.[_0]Buf1<_2,_1> | _0.(\)'_3.[_0]Buf1<_1,_3>)
Step> 1
  (\_3)
   0:  |>'_1.[_2](^)('_0.[_4]Buf1<_1,_0> | _0.(\)'_3.[_0]Buf1<_1,_3>)
Step> quit

MWB> agent Buf22 = (\i,o,x,y)('o.[x]Buf21(i,o,y) + [i=o]t.0)

MWB> weq Buf2 Buf20
The two agents are NOT related.

MWB> weqd (i) Buf2(i,o) Buf20(i,o)
The two agents are related.
Relation size = 8.  Do you want to see it?  (y or n) y
R = < (^)(_1.(\)'_1.[_0]Buf1<_2,_1> | _0.(\)'_3.[_0]Buf1<_1,_3>),
      _0.(\)Buf21<_1,_2,_0> >  {_0#_1}
      ...
```

Figure 5.1: A simple sample session with the MWB.

Orava and Parrow [OP92], who also verified the protocol algebraically. Fredlund and Orava [FO92] later verified the protocol automatically by specifying the protocol in LOTOS [vEVD89], which was translated to labelled transition systems using the Cæsar tool [GS90], which were in turn minimized using the Aldébaran tool [Fer89], and finally compared using Aldébaran. The following informal presentation of the protocol is based on the presentation in [FO92].

The PLMN is a cellular system which can be seen as consisting of Mobile Stations (MSs), Base Stations (BSs), and Mobile Switching Centres (MSCs). The MS, mounted in e.g. a car, provides service to an end user. The BS manages the interface between the MS and a stationary network, controlling all radio communication within a geographical area (a cell). All communication with the MS in a cell is routed through the BS responsible for the cell. The MSC manages a set of BSs, and communicates with them and with other MSCs using a stationary network.

When a MS moves across a cell boundary, the handover procedure changes the communication partner of the MS from the BS of the old cell to the BS of the new cell, ensuring that the MS is constantly in contact with the MSC. The MSC initiates the handover by transmitting a *handover command* message to the MS via the old BS. The handover command message contains parameters enabling the MS to locate the new BS. When transmitting this message the MSC suspends transmission of all messages except for messages related to the handover procedure. When the MS receives the handover command message, it disconnects the old radio links and initiates the new radio links. To establish these connections the MS sends *handover access* messages to the new BS, in order to synchronize with the new BS. When the connections are successfully established, the BS sends a *handover complete* message to the MSC via the new BS. When this message has been received, the network resumes normal operations and releases the old radio links, which are now free and can be allocated to another MS.

In Figure 5.2 we present a π-calculus specification of the protocol. For a discussion on the modelling of the protocol, see [OP92], from which this is taken. In previous work [VM94], we omitted the failure handling aspects of the protocol, but here we present (and verify) the *full* protocol. In Figure 5.4 we present the MWB code for this specification, which differs from Figure 5.2 in that the argument lists of agent identifier definitions must contain all free names which appear in the agent. (This can sometimes be unconvenient, e.g., in this Figure we had to cut the argument lists to fit in the paper width available).

Correctness of this specification would come from showing that it matched some (ideally simple) service specification which would clearly define the desired behaviour of the system. In Figures 5.3 and 5.5, we present the service specification of the handover protocol and its rendering into MWB code, respectively.

When checking the protocol specification (`System`) against the more abstract service specification (`Spec`), we must express the fact that the parameters `in`, `out`, `ho_acc`, `ho_com`, `data`, `ho_cmd`, `ho_fail` and `ch_rel` are constants, i.e. they are distinct from all other free names. This is done by using the `weqd` (weak open bisimulation with distinctions) command of the MWB, in the following way:

```
weqd (i,o,a,co,d,cm,f,r) Spec(i,o) System(i,o,a,co,d,cm,f,r)
```

The bisimulation found by the MWB has 327 tuples. Running on a Sun SPARCstation 10, it took less than 11 CPU minutes (user mode). The ML heap size was at most 295 Mb. The memory requirement is quite extreme, and because of the use of hash tables, the locality of references is very poor, which

$$
\begin{array}{lcl}
CC(f_a, f_p, l) & \stackrel{def}{=} & \mathtt{in}(v).\overline{f_a}\,\mathtt{data}.\overline{f_a}\,v.CC(f_a, f_p, l) \\
& & +l(m_{new}).\overline{f_a}\,\mathtt{ho\_cmd}.\overline{f_a}\,m_{new}. \\
& & \quad \big(\; f_p(c).[c = \mathtt{ho\_com}]\overline{f_a}\,\mathtt{ch\_rel}.f_a(m_{old}). \\
& & \qquad \bar{l}\,m_{old}.CC(f_p, f_a, l) \\
& & \quad +f_a(c).[c = \mathtt{ho\_fail}]\bar{l}\,m_{new}.CC(f_a, f_p, l)\big) \\
HC(l, m) & \stackrel{def}{=} & \bar{l}\,m.l(m).HC(l, m) \\
MSC(f_a, f_p, m) & \stackrel{def}{=} & (\boldsymbol{\nu} l)\big(HC(l, m) \mid CC(f_a, f_p, l)\big) \\
BS_a(f, m) & \stackrel{def}{=} & f(c).\Big(\;[c = \mathtt{data}]f(v).\overline{m}\,\mathtt{data}.\overline{m}\,v.BS_a(f, m) \\
& & \quad +[c = \mathtt{ho\_cmd}]f(v).\overline{m}\,\mathtt{ho\_cmd}.\overline{m}\,v. \\
& & \qquad \big(\; f(c).[c = \mathtt{ch\_rel}]\overline{f}\,m.BS_p(f, m) \\
& & \qquad +m(c).[c = \mathtt{ho\_fail}]\overline{f}\,\mathtt{ho\_fail}.BS_a(f, m)\big)\Big) \\
BS_p(f, m) & \stackrel{def}{=} & m(c).[c = \mathtt{ho\_acc}]\overline{f}\,\mathtt{ho\_com}.BS_a(f, m) \\
MS(m) & \stackrel{def}{=} & m(c).\Big(\;[c = \mathtt{data}]m(v).\overline{\mathtt{out}}\,v.MS(m) \\
& & \quad +[c = \mathtt{ho\_cmd}]m(m_{new}). \\
& & \qquad \big(\; \overline{m_{new}}\,\mathtt{ho\_acc}.MS(m_{new}) \\
& & \qquad +\overline{m}\,\mathtt{ho\_fail}.MS(m)\big)\Big) \\
P(f_a, f_p) & \equiv & (\boldsymbol{\nu} m)\big(MSC(f_a, f_p, m) \mid BS_p(f_p, m)\big) \\
Q(f_a) & \equiv & (\boldsymbol{\nu} m)\big(BS_a(f_a, m) \mid MS(m)\big) \\
System & \equiv & (\boldsymbol{\nu} f_a)(\boldsymbol{\nu} f_p)\big(P(f_a, f_p) \mid Q(f_a)\big)
\end{array}
$$

Figure 5.2: A formal specification of the handover procedure.

means that unless plenty of physical memory is available, a great deal of real time will be spent paging.

$$
\begin{array}{lcl}
Spec & \stackrel{def}{=} & \mathtt{in}(v).S_1(v) \;+\; \tau.Spec \\
S_1(v) & \stackrel{def}{=} & \mathtt{in}(v).S_2(v_1, v) \;+\; \overline{\mathtt{out}}\,v_1.Spec \;+\; \tau.\overline{\mathtt{out}}\,v_1.Spec \\
S_2(v_1, v_2) & \stackrel{def}{=} & \mathtt{in}(v).S_3(v_1, v_2, v) \;+\; \overline{\mathtt{out}}\,v_1.S_1(v_2) \;+\; \tau.\overline{\mathtt{out}}\,v_1.\overline{\mathtt{out}}\,v_2.Spec \\
S_3(v_1, v_2, v_3) & \stackrel{def}{=} & \overline{\mathtt{out}}\,v_1.S_2(v_2, v_3)
\end{array}
$$

Figure 5.3: A formal specification of the service specification.

```
agent CC(fa,fp,l,in,data,ho_cmd,ho_com,ho_fail,ch_rel) =
     in(v).'fa<data>.'fa<v>.CC(fa,fp,l,in,data,...)
   + l(mnew).'fa<ho_cmd>.'fa<mnew>.
       (fp(c).[c=ho_com]'fa<ch_rel>.fa(mold).'l<mold>.CC(fp,fa,l,...)
       +fa(c).[c=ho_fail]'l<mnew>.CC(fa,fp,l,...)

agent HC(l,m) = 'l<m>.l(m).HC(l,m)

agent MSC(fa,fp,m,in,data,ho_cmd,ho_com,ho_fail,ch_rel) =
     (^l)(HC(l,m) | CC(fa,fp,l,in,data,ho_cmd,ho_com,ho_fail,ch_rel))

agent BSa(f,m,ho_acc,ho_com,data,ho_cmd,ch_rel,ho_fail) =
     f(c).( [c=data]f(v).'m<data>.'m<v>.BSa(f,m,ho_acc,...)
          + [c=ho_cmd]f(v).'m<ho_cmd>.'m<v>.
               (f(c).[c=ch_rel]'f<m>.BSp(f,m,ho_acc,...)
               +m(c).[c=ho_fail]'f<ho_fail>.BSa(f,m,...)

agent BSp(f,m,ho_acc,ho_com,data,ho_cmd,ch_rel,ho_fail) =
     m(c).[c=ho_acc]'f<ho_com>.BSa(f,m,ho_acc,ho_com,data,ho_cmd,...)

agent MS(m,data,ho_cmd,out,ho_acc,ho_fail) =
     m(c).( [c=data]m(v).'out<v>.MS(m,data,ho_cmd,out,ho_acc,ho_fail)
           +[c=ho_cmd]m(mnew).
               ('mnew<ho_acc>.MS(mnew,data,ho_cmd,out,ho_acc,ho_fail)
               +'m<ho_fail>.MS(m,data,ho_cmd,out,ho_acc,ho_fail)))

agent P(fa,fp,in,ho_acc,ho_com,data,ho_cmd,ch_rel,ho_fail) =
     (^m)( MSC(fa,fp,m,in,data,ho_cmd,ho_com,ho_fail,ch_rel)
         | BSp(fp,m,ho_acc,ho_com,data,ho_cmd,ch_rel,ho_fail))

agent Q(fa,out,ho_acc,ho_com,data,ho_cmd,ch_rel,ho_fail) =
     (^m)( BSa(fa,m,ho_acc,ho_com,data,ho_cmd,ch_rel,ho_fail)
         | MS(m,data,ho_cmd,out,ho_acc,ho_fail))

agent System(in,out,ho_acc,ho_com,data,ho_cmd,ch_rel,ho_fail) =
     (^fa)(^fp)( P(fa,fp,in,ho_acc,ho_com,data,ho_cmd,ch_rel,ho_fail)
               | Q(fa,out,ho_acc,ho_com,data,ho_cmd,ch_rel,ho_fail))
```

Figure 5.4: The MWB code for the protocol specification.

```
agent Spec(in,out) = in(v).S1(v,in,out) + t.Spec(in,out)

agent S1(v1,in,out) =
   in(v).S2(v1,v,in,out) + 'out<v1>.Spec(in,out)
  + t.'out<v1>.Spec(in,out)

agent S2(v1,v2,in,out) =
   in(v).S3(v1,v2,v,in,out) + 'out<v1>.S1(v2,in,out)
  + t.'out<v1>.'out<v2>.Spec(in,out)

agent S3(v1,v2,v3,in,out) = 'out<v1>.S2(v2,v3,in,out)
```

Figure 5.5: The MWB code for the service specification.

Chapter 6

Related Work

The work described in this thesis is of course related to much other work on the π-calculus, but I will only mention the standard works: Milner, Parrow and Walker's original paper [MPW92], which was first published in technical reports at Edinburgh University in 1989, is the first standard reference on the π-calculus. It describes the monadic calculus, and the late and early bisimulation equivalences. In [Mil93a], Milner describes the polyadic calculus, the replication operator and the notion of sortings.

The MWB (Mobility Workbench), being a tool for a process algebra, is related to many other tools for process algebras (for a survey see [Mad92]). Its relation to the CWB (Concurrency Workbench) [CPS93] may seem strong; both tools were developed in collaboration with people at Edinburgh University, both are written in Standard ML, and one of the modules of the MWB (16 lines for "variables") was actually written for the CWB by Joachim Parrow in September 1986! However, their principles are quite different. The algorithms used by the CWB (with some exceptions) generate the state space of the agent before starting the analysis, while the MWB cannot do this (for reasons explained in Chapter 3). The CWB is of course a much more mature tool than the MWB.

The only other π-calculus tool known is PICT of Pierce, Rémy and Turner [PRT93], a typed concurrent higher-order object-oriented programming language based on the Higher-Order π-calculus [San92]. However, PICT is a tool *using* the π-calculus, rather than a tool for reasoning *about* the π-calculus. It is quite impressive and includes a graphical library for the X Window System.

Jonsson and Parrow [JP93] studied the problem of deciding bisimulation equivalence between programs in value-passing CCS, with the restriction that the programs must be *data-independent*, i.e. their behaviour does not depend on the values passed. Through the use of *schematic names* they reduced this problem to finite-state, thus making it possible to use standard algorithms for finite-state programs (e.g. [PT87] or [KS90]).

Hennessy and Lin [HL92] use *symbolic transition graphs* for value-passing CCS to define *symbolic bisimulations*, parameterized on boolean expressions and corresponding to standard bisimulations (late and early). They also give an algorithm for finding the most general boolean expression under which two agents are symbolically bisimilar. However, the algorithm only works for a subclass of the finite control agents, where re-use of names does not occur. Recently, Boreale and De Nicola [BDN94] extended this work to the π-calculus.

Dam [Dam93] developed a temporal logic for the polyadic π-calculus, and by relativizing the correctness assertions to conditions on names in the spirit of Hennessy and Lin, defined a proof system and decision procedure for finite-control π-calculus agents. His logic characterizes the late bisimulation equivalence.

In a recent paper, Liu [Liu94] describes a proof technique for the late bisimulation congruence of the π-calculus, using *symbolic D-bisimulations* and conditional commitments, also inspired by Hennessy and Lin.

Chapter 7

Conclusion

In this thesis we presented an automated tool for the polyadic π-calculus, along with the theory of the equivalences which the tool implements. The theoretical work is based on [San93b], but here we presented the equivalences in a polyadic setting, and gave an efficient characterization for the weak equivalence. This is the first time the weak open bisimulation equivalence has been treated formally.

The MWB is the first automated tool to appear for the π-calculus, and its second generation is presented here, with vastly improved performance. It is still too early to know if its use will be as widespread as that of, for example, the CWB, but hopefully it will be an inspiration for others to make better tools.

7.1 Future work

Although the performance of the current version of the MWB is much better than that of the prototype version reported in [VM94], it could be improved in many ways. One crucial part of the MWB is the use of hash tables; these could be tuned for better performance by adjusting the sizes, the degree of fullness, improving the hash function, etc. The hash tables grow rather quickly in size; it may also be interesting to experiment with caching instead of hashing, meaning that we only record a certain number of the latest commitments instead of all commitments, thereby reducing the size of the tables and increasing the locality of the process.

To increase the functionality of the MWB, we intend on adding sort handling, based on e.g the sort inference algorithm of Gay [Gay93]. We also intend on adding the model checking algorithm of Dam [Dam93], although his logic characterizes the late equivalence.

There are also plans to integrate the MWB with the Pi derivation editor of Eriksson [Eri93], where the model checking proof rules of Dam have been implemented. This would let the user choose to prove parts of a formula automatically using the MWB, and other parts interactively using the Pi editor.

The weak open equivalence should be further investigated: its axiomatization and relation to the weak late equivalence, minimization with respect to it, and alternative characterizations.

We would also like to develop a temporal logic characterizing the open equivalence, with an accompanying model checking algorithm. It should also be possible to generate distinguishing formulae, a logic expression describing how two agents differ.

Algorithms should be sought for finding the "minimal" distinction and substitution, respectively, under which two agents are equal. This is strongly related to the work of Hennessy & Lin [HL92] and Boreale & De Nicola [BDN94].

We would like to see graphical interfaces based on Sangiorgi's tree representation [San93b], Parrow's Interaction Diagrams [Par93], and Milner's π-nets [Mil93b]. The latter two could make use of animation techniques, and actually *show* the behaviour of the user's π-calculus specification.

Acknowledgements

The work described in this thesis was partially supported by the Swedish Board for Industial and Technical Development (NUTEK) as part of their contribution to the ESPRIT Basic Research Action project no. 6454, "CONFER".

I would like to thank my advisor, Joachim Parrow, for introducing me to the π-calculus, and for coming up with so many ways to improve the MWB and ideas on how to implement them. He turned out to be more of a hacker than I had ever thought. I owe big thanks to Faron Moller, who formulated the original verification algorithm one morning when I came in late in Edinburgh, who answers all my questions on the ML type and module systems, and who is a great friend. Faron also read and commented on the final manuscript. Lars-åke Fredlund provided some model ML code, from which I learned a lot of ML. I am also very grateful to Davide Sangiorgi, who has always encouraged me in my work. I thank Lars-Henrik Eriksson who told me how de Bruijn β-reduction really works, and Mads Dam for inspiring discussions. Finally I express my deep gratitude to Fredrik Orava for reading and commenting on late versions of this manuscript.

I also thank Ellen for sleeping so well at night, and Helena for her love and patience.

Bibliography

[BDN92] M. Boreale and R. De Nicola. Testing equivalences for mobile processes. In R. Cleaveland, editor, *CONCUR'92, 3^{rd} Intl. Conference on Concurrency Theory*, volume 630 of *Lecture Notes in Computer Science*, pages 2–16. Springer-Verlag, 1992.

[BDN94] M. Boreale and R. De Nicola. A symbolic semantics for the π-calculus. In J. Parrow, editor, *CONCUR'94, 5^{th} Intl. Conference on Concurrency Theory*, Lecture Notes in Computer Science. Springer-Verlag, 1994. To appear.

[BW90] J. Baeten and W. Weijland. *Process algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

[CPS93] R. Cleaveland, J. Parrow and B. Steffen. The Concurrency Workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[Dam93] M. Dam. Model checking mobile processes. In E. Best, editor, *CONCUR'93, 4^{th} Intl. Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 22–36. Springer-Verlag, 1993. Full version in Research Report R94:01, Swedish Institute of Computer Science, Kista, Sweden.

[dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[Eri93] L.-H. Eriksson. *Finitary Partial Inductive Definitions and General Logic*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 1993.

[Fer89] J.-C. Fernandez. Aldébaran: A tool for verification of communicating processes. Technical Report RTC 14, IMAG, Laboratoire de Génie Informatique, 1989.

[FM91] J.-C. Fernandez and L. Mounier. "On-the-fly" verification of behavioural equivalences and preorders. In K. G. Larsen and A. Skou, editors, *Proceedings of CAV'91*, volume 575 of *Lecture Notes in Computer Science*, pages 181–191. Springer-Verlag, 1991.

[FO92] L. Fredlund and F. Orava. Modelling dynamic communication structures in LOTOS. In K. R. Parker and G. A. Rose, editors, *Proceedings of FORTE'91*, pages 185–200. North-Holland, 1992.

[Gay93] S. J. Gay. A sort inference algorithm for the polyadic π-calculus. In *Proceedings, 20th ACM Symposium on Principles of Programming Languages*. ACM Press, 1993.

[GS90] H. Garavel and J. Sifakis. Compilation and verification of LOTOS specifications. In *Proceedings of PSTV X*, 1990.

[HL92] M. Hennessy and H. Lin. Symbolic bisimulations. Technical Report 1/92, School of Cognitive and Computing Sciences, University of Sussex, UK, April 1992.

[Jon93] C. Jones. A pi-calculus semantics for an object based design notation. In E. Best, editor, *CONCUR'93, 4^{th} Intl. Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 158–172. Springer-Verlag, 1993.

[JP93] B. Jonsson and J. Parrow. Deciding bisimulation equivalences for a class of non-finite-state programs. *Journal of Information and Computation*, 107(2):272–302, December 1993.

[KS90] P. Kanellakis and S. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Journal of Information and Computation*, 86(1):43–68, May 1990.

[Li94] B. Li. A π-calculus specification of Prolog. In *Proceedings of European Symposium on Programming*, Edinburgh, 1994. Springer-Verlag.

[Liu94] X. Liu. Characterizing bisimulation congruence in the π-calculus. In J. Parrow, editor, *CONCUR'94, 5^{th} Intl. Conference on Concurrency Theory*, Lecture Notes in Computer Science. Springer-Verlag, 1994. To appear.

[Mad92] E. Madelaine. Verification tools from the CONCUR project. *Bulletin of the European Association of Theoretical Computer Science*, 47:110–126, June 1992.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil90] R. Milner. Sorts and types in the π-calculus. Handwritten notes, December 1990.

[Mil92] R. Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[Mil93a] R. Milner. The polyadic π-calculus: a tutorial. In F. L. Bauer, W. Brauer and H. Schwichtenberg, editors, *Logic and Algebra of Specification*. Springer-Verlag, 1993.

[Mil93b] R. Milner. Action structures for the π-calculus. Technical Report ECS-LFCS-93-264, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, May 1993.

[Mil93c] R. Milner. Action structures and the pi calculus. To be published in *Proceedings of the NATO Summer School on Logic and Computation*, Marktoberdorf, 1993.

[MPW92] R. Milner, J. Parrow and D. Walker. A calculus of mobile processes, Parts I and II. *Journal of Information and Computation*, 100:1–77, September 1992.

[MTH90] R. Milner, M. Tofte and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[OP92] F. Orava and J. Parrow. An algebraic verification of a mobile network. *Journal of Formal Aspects of Computing*, 4:497–543, 1992.

[Ora94] F. Orava. *On the Formal Analysis of Telecommunication Protocols.* PhD thesis, Department of Computer Systems, Uppsala University, Sweden, May 1994.

[Par93] J. Parrow. Interaction diagrams. Research Report R93:06, Swedish Institute of Computer Science, Kista, Sweden, 1993. Also invited presentation for REX'93, to appear in Springer Verlag Lecture Notes in Computer Science.

[PRT93] B. C. Pierce, D. Rémy and D. N. Turner. A typed higher-order programming language based on the pi-calculus. Submitted for publication; an earlier version was presented as an invited lecture at the Workshop on Type Theory and its Application to Computer Systems, Kyoto University, July, 1993.

[PS93] J. Parrow and D. Sangiorgi. Algebraic theory for name-passing calculi. Technical Report ECS-LFCS-93-262, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, April 1993. Accepted for publication in *Journal of Information and Computation.*

[PT87] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, 1987.

[Ros93] B. Ross. A π-calculus semantics logical variables and unification. In S. Purushothaman and A. Zwarico, editors, *Proceedings of First North American Process Algebra Workshop, Stony Brook, 1992, Workshops in Computing*, pages 216–230. Springer-Verlag, 1993.

[San92] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis, Department of Computer Science, University of Edinburgh, UK, 1992.

[San93a] D. Sangiorgi. From π-calculus to Higher-Order π-calculus — and back. In *Proceedings of TAPSOFT'93*, volume 668 of *Lecture Notes in Computer Science*, pages 151–166. Springer-Verlag, 1993.

[San93b] D. Sangiorgi. A theory of bisimulation for the π-calculus. Technical Report ECS-LFCS-93-270, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, June 1993. Extended abstract in E. Best, editor, *CONCUR'93, 4^{th} Intl. Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 127-142. Springer-Verlag, 1993.

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

[VM94] B. Victor and F. Moller. The Mobility Workbench — a tool for the π-calculus. In D. Dill, editor, *Proceedings of CAV'94*, Lecture Notes in Computer Science. Springer-Verlag, 1994.

[Wal90] D. Walker. π-calculus semantics of object-oriented programming languages. Technical Report ECS-LFCS-90-122, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, 1990.

[Wal92] D. Walker. Objects in the π-calculus. Technical Report RR217, Department of Computer Science, University of Warwick, UK, 1992. To appear in *Journal of Information and Computation*.

Appendix A

Correctness of the algorithm

In this appendix we prove total correctness of the algorithm presented in Chapter 3, Figure 3.1. We choose to study the algorithm for the strong case; the proof for the weak case is essentially the same, instead using the definitions of the weak equivalence, commitment relation, and algorithm.

Lemma A.1 *If $D' \subseteq D$, then $P \asymp_{D'} Q$ implies $P \asymp_D Q$.*
PROOF: Along the lines of [MPW92, San93b]. □

For example,

$$P \asymp_{\emptyset} Q \quad \Rightarrow \quad P \asymp_{\{(x,y)\}} Q$$

Lemma A.2 *If $P \asymp_D Q$ and $A = \mathrm{fn}(P, Q)$ then $P \asymp_{D \lceil A} Q$*
PROOF: Along the lines of [MPW92]. □

The following lemma expresses that provided R only contains triples (P, Q, D) such that $P \asymp_D Q$, then *add* only adds triples (P', Q', D') such that $P' \asymp_{D'} Q'$. In the following, we write $\phi(\mathcal{R})$ for $\forall (P, Q, D) \in \mathcal{R} : P \asymp_D Q$.

Lemma A.3 *If R is a set of triples such that $\phi(R)$ holds, then if $S = add(P, Q, D)\ R$ is non-empty, then $\phi(S)$ holds.*
PROOF: We prove the lemma by induction on the number of recursions, n, left until *add* terminates.

Case for $n = 0$: If $add(P, Q, D)\ R$ terminates and returns a non-empty set, it is because one of the following is true:

1. P and Q are processes but have no commitments, i.e. $P \equiv Q \equiv \mathbf{0}$.
   Then $S = \{(P, Q, D)\} \cup R$, and $P \asymp_D Q$.

2. $\exists (P', Q', D') \in R$ s.t. $P' = P$, $Q' = Q$, $D' \subseteq D$
   Then since $P' \asymp_{D'} Q'$ and $D' \subseteq D$, by Lemma A.1, $P \asymp_D Q$, and we return $\{(P, Q, D)\} \cup R$.

Case for $n+1$: If $S = add(P, Q, D)\ R$ did not return immediately, one of the following is true:

1. P and Q are processes and all commitments of P have been matched by Q in a way appropriate for an open conditional D-bisimulation, and each of the derivatives are by induction open conditional D-bisimilar. S is successively built up by the recursive calls matching the derivatives.

   We show that (P, Q, D) indeed do fulfill the requirements in the definition of an open conditional D-bisimulation (Definition 2.12).

   $\forall(\alpha_i, m_i, P_i) \in \{(\alpha, m, P') : P \succ_m \alpha.P', m \text{ respects } D\}$, $\exists \beta_i, n_i, Q_i : Q \succ_{n_i} \beta_i.Q_i$ and

   - $m_i \rhd n_i$
   - $\alpha_i \sigma_{m_i} = \beta_i \sigma_{m_i}$
   - $S_i = (add(P_i\sigma_{m_i}, Q_i\sigma_{m_i}, D\sigma_{m_i})\ R_i) \neq \emptyset$ where $R_0 = \{(P, Q, D)\} \cup R$ and $R_{i+1} = S_i$ and by induction, provided R_i is a set of triples such that $\phi(R_i)$ holds, then $\phi(S_i)$ also holds.
     Note that we could use $R_{i+1} = R_0$, but by using the result accumulated by the other branches, we don't have to recurse over agents we already have considered.

   Since *all* commitments of P are matched in a proper way, and all derivatives are D-bisimilar, $P \asymp_D Q$ if $\phi(R)$ holds. We return the last S_i, which is the largest of the sets.

   Also all commitments of Q are matched by P as required.

2. $P \equiv (\lambda\vec{x})P', Q \equiv (\lambda\vec{x})Q'$ and $S' = add(P', Q', D)\ (\{(P, Q, D)\} \cup R) \neq \emptyset$

3. $P \equiv (\boldsymbol{\nu}\vec{x})[\vec{y}]P', Q \equiv (\boldsymbol{\nu}\vec{x})[\vec{y}]Q'$ and $S' = add(P', Q', D')\ (\{(P, Q, D)\} \cup R) \neq \emptyset$, where $D' = (D\lceil \text{fn}(P, Q)) \cup \{\vec{x} \times \text{fn}(P, Q)\}$

and by induction, and Lemma A.2, in the last two cases $\phi(S')$ holds if $\phi(R)$ holds, and by definition of an open conditional D-bisimulation, $P \asymp_D Q$. In these cases $S = \{(P, Q, D)\} \cup S'$. □

Proposition A.4 *If $S = add(P, Q, D)\ \emptyset$ is non-empty, then $\phi(S)$ holds.*
PROOF: Follows from Lemma A.3, using $R = \emptyset$. □

Proposition A.5 *When $S = add(P, Q, D)\ R = \emptyset$, $P \not\asymp_D Q$.*
PROOF: By examining the code in Figure 3.1, we see that $\emptyset$ is returned when there are commitments which cannot be matched in accordance with the definition of an open conditional D-bisimulation, or when we cannot match abstractions or concretions properly. Note especially the first case of *match*, which makes sure $\emptyset$ is returned when the "inner" call to *match* in *add* returns $\emptyset$. □

Proposition A.6 *For agents P, Q with finite control, $add(P, Q, D)\ R$ always terminates.*
PROOF: Agents with finite control are built up by parallel composition of sequential agents (containing no parallel composition) with finite branching. The sequential agents have a minimal agent space which is finite, and the parallel composition of these also has a finite minimal agent space. When comparing two agents with finite minimal agent spaces, we may have to extend the agent spaces (as described in Chapter 3), but they will still be finite.

Since we examine finite agent spaces, and we record and check whether we have already checked the agents in question for each recursion (and if so return immediately), the algorithm always terminates. □

Now, since the algorithm always terminates and returns an answer, and the answer can only be either empty or non-empty, and the result in either case is correct, the algorithm is totally correct.

Appendix B

The MWB User's Guide

B.1 Introduction

The Mobility Workbench (MWB) is a tool for manipulating and analyzing mobile concurrent systems described in the π-calculus [MPW92, Mil93a], developed by Björn Victor[1] and Faron Moller[2], in collaboration with Davide Sangiorgi. It is written in Standard ML, and currently runs under the New Jersey SML compiler.

In the current version, the basic functionality is to decide the *open bisimulation equivalences* of Sangiorgi [San93b], for agents in the polyadic π-calculus with the original positive match operator. This is decidable for π-calculus agents with *finite control*, corresponding to CCS finite-state agents, which do not admit parallel composition within recursively defined agents.

The algorithm is based on the alternative "efficient" characterizations of the equivalences described in [San93b] and in Chapter 2 of this thesis, and (necessarily) generates the state space "on the fly". Algorithms for both the strong and weak equivalences are implemented. There are also commands e.g. for finding deadlocks and interactively simulating an agent.

B.2 Syntax of agents

The syntax of agents is given by the following grammar:

$$\begin{array}{rcl} P & ::= & \mathtt{0} \;\Big|\; \alpha.P \;\Big|\; \mathit{pfx}.P \;\Big|\; [a=b]P \;\Big|\; P_1 \mid P_2 \;\Big|\; P_1 + P_2 \;\Big| \\ & & \mathit{Id}(\mathit{nlist}) \;\Big|\; (\hat{}\,\mathit{nlist})P \;\Big|\; (\backslash \mathit{nlist})P \;\Big|\; [\mathit{nlist}]P \;\Big|\; (P) \end{array}$$

where *nlist* is a (possibly empty) comma-separated list of names, *Id* is a name starting with an upper-case letter, `^` is used for $\boldsymbol{\nu}$, `\` is used for λ; and α is τ (written `t`) or a name `x` or a co-name `'x`, and *pfx* is an abbreviated prefix: an input action $a(\mathit{nlist})$ or an output action written $'a{<}\mathit{nlist}{>}$, which are equivalent to $a.(\backslash \mathit{nlist})$ and $\overline{a}.[\mathit{nlist}]$, respectively. Names must start with a lowercase letter but can after that include the characters _, $, ', letters and digits.

[1]Department of Computer Systems, Uppsala University, Box 325, S-751 05 Uppsala, Sweden; email: `Bjorn.Victor@DoCS.UU.SE`. Work supported by the ESPRIT BRA project 6454 "CONFER"

[2]SICS, Box 1263, S-164 28 Kista, Sweden; email: `fm@sics.se`. Work supported by the ESPRIT BRA projects 7166 "CONCUR2" and 6454 "CONFER".

B.3 Commands of the MWB

B.3.1 `help`

gives a help text. `?` (questionmark) is a synonym for this command.

B.3.2 `quit`

terminates the program.

B.3.3 `agent`

defines an agent identifier. Two equivalent examples:

```
agent P(x,y) = (^z)'x<y,z>.y(x,y).P(y,x)
agent P = (\x,y)(^z)'x.[y,z]y.(\ x,y)P(y,x)
```

An agent definition must be closed, i.e., its free names must be a subset of the argument list.

B.3.4 `clear`

removes agent identifier definitions. `clear P` removes the definition of the agent identifier P, while `clear` without an argument removes all definitions.

B.3.5 `deadlocks` *agent*

finds and describes deadlocks in the agent given as argument. It displays the agent in which the deadlock is found, and a trace of commitments from the initial agent ending in the deadlocked agent.

The deadlocks are displayed as they are found, which makes the command useful even if the state space is infinite.

B.3.6 `env`

prints all agent definitions in the environment.

B.3.7 `eq` $agent_1$ $agent_2$

checks whether $agent_1$ and $agent_2$ are strong open bisimulation equivalent.

B.3.8 `eqd` `(`$name_1$,...,$name_n$`)` $agent_1$ $agent_2$

checks whether $agent_1$ and $agent_2$ are strong open bisimulation equivalent given the distinction formed by making $name_1, \ldots, name_n$ distinct from all free names in $agent_1$ and $agent_2$. Thus $\{name_1, \ldots, name_n\}$ should be a subset of the free names of $agent_1$ and $agent_2$. (Names not free in $agent_1$ or $agent_2$ are meaningless and are simply removed).

B.3.9 `input "`*filename*`"`

reads commands from the file named *filename*. The double quotes are part of the syntax but not of the filename.

B.3.10 `step` *agent*

interactively simulates the agent, by presenting the possible commitments of the agent and letting the user select one, and repeating this until there are no possible commitments.

B.3.11 `weq` $agent_1$ $agent_2$

checks whether $agent_1$ and $agent_2$ are *weak* open bisimulation equivalent.

B.3.12 `weqd` ($name_1$,...,$name_n$) $agent_1$ $agent_2$

checks whether $agent_1$ and $agent_2$ are weak open bisimulation equivalent given the distinction formed by making $name_1,\ldots,$ $name_n$ distinct from all free names in $agent_1$ and $agent_2$. Thus $\{name_1, \ldots, name_n\}$ should be a subset of the free names of $agent_1$ and $agent_2$.

B.3.13 `debug` *n*

sets the debugging level of the program. n should be a non-negative integer; the only values we expect to be valuable to users other than the developers are 0 (meaning debugging is turned off), and 1 (which gives a timing printout at the end of some commands). The use of this command for higher values of n is discouraged, and as such is left undocumented here.

B.4 Getting the MWB

The polyadic version of the MWB is available on request.

The monadic version of the MWB is available by anonymous FTP from the host `ftp.docs.uu.se` in the directory `pub/mwb`. The file `README` contains further directions and information. An up-to-date version of this guide is always part of the distribution.

Binary executables are provided for some architectures and operating systems. Source code is also provided which can be compiled with the SML-NJ compiler. SML-NJ is currently available from the host `research.att.com`, directory `dist/ml` and the host `princeton.edu`, directory `pub/ml`.

Any bug reports, queries, feedback etc should be sent to:

email: `Bjorn.Victor@DoCS.UU.SE`
fax: +46 18 550225
mail: Björn Victor
Dept. of Computer Systems
Uppsala University
Box 325
S-751 05 Uppsala
SWEDEN