

The Model Prover  
– a sequent-calculus based  
modal  $\mu$ -calculus model checker tool  
for  
finite control  $\pi$ -calculus agents

Fredrick B. Beste [fredrikb@sics.se](mailto:fredrikb@sics.se)

January 9, 1998

## Abstract

The  $\pi$ -calculus offers some very good properties for describing dynamic, distributed communicating systems. The modal  $\mu$ -calculus offers ways of describing properties for infinite processes and their behaviour. The Mobility Workbench (MWB) is equipped with methods of deciding bisimulation equivalences for  $\pi$ -calculus agents. We present a sequent-calculus based model checker tool for the MWB. We also introduce the basic theory therefore, the  $\lambda$ -calculus and CCS for algebraic description of processes, and modal and temporal logics for expressing properties of process behaviour, and also the classic sequent-calculus originating from the 1930's. Model checkers and bisimulation checkers are fundamental tools for the research area of *Formal Methods*, a branch of Computer Science that deals with verification of computer software by mathematical and automated reasoning as a way for designing and verifying software.

# Contents

<b>1</b>	<b>Acknowledgements</b>	<b>1</b>
<b>2</b>	<b>Introduction</b>	<b>2</b>
2.1	Background	2
2.2	Formal Methods	2
2.3	Previous Work	2
2.4	Outline	3
2.5	Notation	3
<b>I</b>	<b>Functions &amp; Processes</b>	<b>4</b>
<b>3</b>	<b>The <math>\lambda</math>-calculus</b>	<b>4</b>
3.1	Functional programming	4
3.2	Free and Bound names	5
3.3	Substitution	5
3.4	Abstraction	5
3.5	BNF Syntax for $\lambda$ -calculus	6
3.6	Rules for manipulation of $\lambda$ -terms	6
3.6.1	$\alpha$ -conversion	6
3.6.2	$\beta$ -reduction	6
3.7	Currying	7
3.8	Combinators	7
3.8.1	Identity Combinator	7
3.8.2	The Boolean Combinators	7
3.8.3	"If" Combinator	8
3.8.4	The Divergent Combinator	8
3.8.5	Fixed-point Combinator	8
3.9	A $\lambda$ -calculus example	9
3.10	Normal Forms	9

3.11	The Reduction Strategies . . . . .	10
3.11.1	Normal Order . . . . .	10
3.11.2	Lazy Reduction . . . . .	10
3.11.3	Applicative Reduction . . . . .	10
3.12	De Bruijn-indices . . . . .	10
<b>4</b>	<b>The <math>\pi</math>-calculus</b> . . . . .	<b>12</b>
4.1	Actions and names . . . . .	12
4.1.1	$\alpha$ -conversion . . . . .	12
4.1.2	Substitution . . . . .	13
4.2	Labeled Transitions . . . . .	13
4.3	Definition . . . . .	13
4.3.1	Summation . . . . .	13
4.3.2	Prefix . . . . .	14
4.3.3	Composition . . . . .	14
4.3.4	Replication . . . . .	15
4.3.5	Restriction . . . . .	15
4.3.6	Match . . . . .	15
4.3.7	Definition . . . . .	16
4.4	Binding of names . . . . .	16
4.5	Abstraction & Concretion . . . . .	16
4.6	Monadic / Polyadic . . . . .	16
4.7	Examples . . . . .	17
4.8	Agents, Structural Congruence . . . . .	17
4.8.1	Agents as a special case of processes . . . . .	17
4.8.2	Structural congruence for agents . . . . .	17
4.9	Normal form . . . . .	18
4.10	Application . . . . .	18
4.10.1	Pseudo-application . . . . .	18
4.11	Commitments . . . . .	18

4.12 Distinction . . . . .	19
4.13 Equivalence relations for processes . . . . .	19
4.14 Strong Bisimulation Equivalences . . . . .	20
4.14.1 Late . . . . .	20
4.14.2 Early . . . . .	20
4.14.3 Open . . . . .	21
4.14.4 Difference between bisimulation equivalences. . . . .	21
4.14.5 Strong bisimulation equivalence and congruence . . . . .	21
4.15 Weak Bisimulation Equivalences . . . . .	22
4.15.1 Late . . . . .	22
4.15.2 Early . . . . .	23
4.15.3 Open . . . . .	23
4.15.4 Difference between bisimulation equivalences. . . . .	23
4.16 Properties for Weak Bisimulation Equivalences . . . . .	23
<b>II Logic for processes</b>	<b>25</b>
<b>5 Sequent Calculus</b>	<b>25</b>
5.1 Proof System . . . . .	25
5.2 Proofs in the sequent calculus. . . . .	27
<b>6 Modal Logic</b>	<b>28</b>
6.1 Modal Operators . . . . .	28
6.2 Reduction Laws for modal operators . . . . .	29
6.3 Hennessy-Milner Logic . . . . .	29
6.3.1 Definition of HML [17] . . . . .	29
6.4 Modal Equivalence . . . . .	30
6.5 Limitation . . . . .	30
<b>7 Temporal Logic</b>	<b>31</b>
7.1 Liveness, Safety . . . . .	31

7.2	Fixed points . . . . .	31
7.3	The modal $\mu$ -calculus . . . . .	32
7.4	Meaning . . . . .	33
7.5	Liveness and Safety properties . . . . .	33
7.6	Tableau Proof System . . . . .	34
7.6.1	The tableau of rules . . . . .	34
7.7	Fold, Unfold, Discharge . . . . .	34
7.7.1	Unfold . . . . .	35
7.7.2	Fold . . . . .	35
7.7.3	Discharge, fail . . . . .	35
<b>III</b>	<b>Presentation</b>	<b>36</b>
<b>8</b>	<b>Model Checking</b>	<b>36</b>
8.1	The Problem . . . . .	36
8.2	Checker vs. Prover: Differences. . . . .	36
8.3	A Sequent calculus model checker . . . . .	37
<b>9</b>	<b>Components</b>	<b>38</b>
9.1	Atoms . . . . .	38
9.1.1	Names & Parameters . . . . .	38
9.1.2	Variables . . . . .	38
9.1.3	Actions . . . . .	38
9.1.4	Propositional Variables . . . . .	39
9.1.5	U - formula indexing . . . . .	39
9.1.6	I - Denotation parameter index . . . . .	39
9.2	Molecular Structures . . . . .	40
9.2.1	Name-Equations . . . . .	40
9.2.2	finite-control Agents . . . . .	40
9.2.3	Formulas . . . . .	40

9.2.4	Structure	40
9.2.5	Alternatives	41
9.2.6	Suspended Sequents	41
9.2.7	Visited Fixpoints	42
9.2.8	Conjunctive Sequents	43
<b>10 Model Prover BNF Description</b>		<b>44</b>
<b>11 The Tableau System of Rules</b>		<b>46</b>
11.1	Set and list denotations	46
11.2	Operators, predicates and functions	47
11.2.1	Suspended Variable	47
11.2.2	Quietness, Implication	47
11.2.3	Consistence	48
11.3	Axioms	48
11.3.1	STERM	48
11.3.2	FTERM	48
11.4	Structural Rules	48
11.4.1	T-LIT	49
11.4.2	F-LIT	49
11.5	Equivalence Rules	49
11.5.1	EQ1	49
11.5.2	EQ2	49
11.5.3	EQ3	49
11.5.4	EQ4	50
11.5.5	INEQ1	50
11.5.6	INEQ2	50
11.6	Logical Connectives	51
11.6.1	$\vee$ (OR)-intro	51
11.6.2	$\wedge$ (AND)-intro	51
11.7	Predicate Logic Quantifiers	51

11.7.1	$\forall$ -intro	51
11.7.2	$\exists$ -intro	51
11.8	Agent Rules	52
11.8.1	COND1	52
11.8.2	COND2	52
11.8.3	COND3	52
11.8.4	$\Sigma$ -intro	52
11.8.5	New-Rule	53
11.9	Modal Rules	53
11.9.1	DIAMOND1	53
11.9.2	DIAMOND2	53
11.9.3	DIAMOND3	53
11.9.4	Box1	54
11.9.5	BOX2	54
11.9.6	BOX3	54
11.10	Temporal Rules	54
11.10.1	LOOPCHECK	54
11.10.2	DISCHARGE	55
11.10.3	Fold	55
11.10.4	Unfold	55
11.11	Implicit Rules	55
11.11.1	Bar	55
11.11.2	Equivalence	56
11.11.3	Names	56
11.12	Suspension	56
11.13	Contraction	57
<b>12</b>	<b>Semantics</b>	<b>58</b>
12.1	Definitions	58
12.2	Tableau Rules Semantic definitions	58



<b>13 Soundness</b>	<b>60</b>
13.1 The axioms & structural rules . . . . .	60
13.2 Equivalence rules . . . . .	61
13.3 Logical rules . . . . .	61
13.4 Quantifiers . . . . .	61
13.5 Agent rules . . . . .	62
13.6 Summation & New rules . . . . .	62
13.7 Modal rules . . . . .	62
13.8 Fixpoint rules . . . . .	63
<b>14 Termination</b>	<b>64</b>
<b>15 Implementation</b>	<b>66</b>
15.1 de Bruijn-Indexing . . . . .	66
15.2 Back-Tracking . . . . .	66
<b>16 The Mobility Workbench with Model Prover</b>	<b>68</b>
16.1 Where? . . . . .	68
16.2 How? . . . . .	68
16.2.1 MWB Prover Syntax . . . . .	68
16.3 Examples . . . . .	70
16.3.1 Agent examples . . . . .	70
16.3.2 Formulae examples . . . . .	72
16.4 Some Model Prover run-through examples . . . . .	77
<b>17 Summary &amp; discussion</b>	<b>78</b>
<b>IV Appendix</b>	<b>81</b>
<b>A Implementation results</b>	<b>81</b>
<b>B References</b>	<b>84</b>

# 1 Acknowledgements

This is a Thesis submitted for the Degree of Master of Science in Computer Science at the *Department of Computer Science at Uppsala University*, Sweden, january 1998. The work has been carried out in the *Formal Description Techniques Group* at the *Swedish Institute Of Computer Science, SICS*, Kista, Stockholm. The text was written in L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>[28].

I would like to thank Björn Victor, Mads Dam, Lars-åke Fredlund and Torkel Franzén for their lectures, support, help, suggestions and incredible patience. I would also like to express gratitudes to Dilian Gurov for great company. The time at SICS was indeed an interesting and educational period in my life.

Fredrick Beste, Uppsala, January 1998.

**Dedicated to Lisa**

## 2 Introduction

### 2.1 Background

The  $\pi$ -calculus is a process algebra - a mathematical way of expressing *computer programs* (processes). It has some expressful "built-in" features, good for describing *distributed*, *communicating* and *dynamic* computer systems (systems where processes can change over time). A good example is the GSM digital telephone network, where the processes (telephones) can be turned on and off, and the telephone can move within the network. This is often referred to as *mobility*. A temporal logic is a language that can express properties for processes and their actions (possibly infinitely many - they may run forever). The temporal logic can describe properties like "it must always be possible to move the process to another state", (i.e. the process must never deadlock), or "it must react upon every possible input", and such specific behaviours. To combine a process, that we wish to control for a certain property, and a formula, that expresses the property, and calculate whether the formula holds for the process or not (if the process is a *model* for the formula) is known as *model checking*. In this paper, we will present an algorithm for a model checker, *The Model Prover*, for checking a specific class of  $\pi$ -calculus-processes ("finite control agents") against a formula, as expressed in the temporal logic known as *the modal  $\mu$ -calculus*.

### 2.2 Formal Methods

Model checkers are part of a growing sub-area of Computer Science: Formal Methods [1], an area that uses mathematical methods (mostly algebras and logic) for formal mathematical proofs of computer software. The demand for those methods increases as computer systems grow more complex; exhaustive testing and "trial-and-error" methods of debugging become obsolete, expensive and even impossible, due to the fact that computer programs may be infinite, or so huge (have so many program states) that a thorough testing would take extremely long time or go on forever. Formal methods approaches use well-known mathematical facts to reduce the time and effort it takes to fully check a program. The mathematical verification of a program can be done on a computer, the model checking is then a computer program.

### 2.3 Previous Work

The Mobility Workbench (MWB) is a tool for verifying if two processes are similar (*bisimulation equivalences*). It was written mainly by Björn Victor of the Department of Computer Systems at Uppsala University, as a licentiate thesis [12] 1994, and implemented in a Standard ML environment. It was later extended with a model checking tool, developed by Mads Dam of the Swedish Institute of Computer Science (SICS), Kista, Stockholm, in [14]. Later, another member of SICS, Torkel Franzén, developed another algorithm for the same model

checker, but this algorithm was based on the classical, logical work known as *the sequent calculus*, developed as early as in the 1930's for predicate logic. This alternative algorithm was presented briefly as an extended abstract in [11], and, developed in Prolog, was thus not an integrated part of the MWB.

## 2.4 Outline

This thesis augments the work from [11], with a brief presentation of the theoretical background for process algebras, mainly the  $\pi$ -calculus, and its ancestors,  $\lambda$ -calculus, (the closest ancestor, Calculus For Communicating Systems (CCS), is mentioned in the  $\pi$ -calculus chapter); for logic, we present modal logic as an extension of predicate logic, we mention HML-logic and temporal logic, especially the modal  $\mu$ -calculus. We also introduce the sequent calculus for predicate logic.

In part three, a full algorithm description for the Model Prover is presented, along with its semantics and we reason about its soundness and behaviour with respect to termination. We end with some conclusions, examples and implementation results.

Another part of this thesis work was to implement the prover as an integrated Standard ML-program in the MWB.

This work is available from the FTP-source<sup>1</sup> along with the rest of the MWB. The work was carried out at SICS in Kista during the summer, autumn and winter of 1997. The thesis is written with the intention of being an introduction to formal methods for anyone with an interest therefore and some theoretical background in logic and algebra.

## 2.5 Notation

**Definition 1** *We will write  $\triangleq$  when meaning "assignment". In [8], this is written  $\stackrel{def}{=}$ . This is an operator, while different kinds of equivalences (e.g.  $=$ ,  $\equiv$ ,  $\sim$ , and  $\approx$ ) are relations.*

**Definition 2** *A congruence is an equivalence relation that is preserved by all introduced operators and in all possible contexts.*

---

<sup>1</sup>See chapter 16 for details.

## Part I

# Functions & Processes

## 3 The $\lambda$ -calculus

We will firstly present an overview of the classical  $\lambda$ -calculus [29]. It can be described as one of the first attempts to "catch" computable functions in an algebraic way. It was developed mainly by Alonzo Church during the 1930s, but H.B. Curry also participated <sup>2</sup>. Alongside Turing's machine and Kleene's recursive functions,  $\lambda$ -calculus is one of several attempts to formalize "the algorithmic".  $\lambda$ -calculus can be described as a formal system: it has a symbolic language, and rules for manipulating those "terms" of the language. It can also be regarded as the "mother" of several later process algebras/calculi, even though  $\lambda$ -calculus itself is not a process algebra (a function is not the same as a process<sup>3</sup> - especially when dealing with communication, concurrency, the difference is obvious). But  $\lambda$ -calculus has nevertheless interesting properties and notations, some inherited by subsequent calculi, that motivates a brief presentation here. A common proof method when developing new process algebras, one might want to show that the new calculus is "functionally complete". This is often done by showing that all the expressive power of  $\lambda$ -calculus is included in the new-born calculus.<sup>4</sup>

### 3.1 Functional programming

$\lambda$ -calculus can be viewed as the theoretical fundament for functional programming languages such as LISP, Scheme and ML. Some differences between functional and most imperative languages (Algol, Pascal, C, Java...), is that they only consist of *functions*, that *always* have return values, where imperative languages differ between functions and procedures<sup>5</sup> that can alter a memory location explicit without returning anything, e.g. the "void"-procedure of C. So imperative languages regard the memory as static, consisting of cells that can be manipulated during execution. In functional programming, the memory is regarded as dynamic in size, but the contents are the same. Imperative languages have a more machine-oriented way of treating the memory<sup>6</sup>, whilst functional languages use a much more mathematical approach.

---

<sup>2</sup>Although he later concentrated on "Combinatory Logic"

<sup>3</sup>But a function can be viewed as a special case of a process

<sup>4</sup>That  $\lambda$ -calculus is a "sub-calculus" of the new calculus.

<sup>5</sup>sometimes even "methods"

<sup>6</sup>although there are imperative languages that have a more non-direct access to memory, like automatic garbage collection.

### 3.2 Free and Bound names

Just as the quantifiers  $\exists$  and  $\forall$  in predicate logic binds variables in a logic expression [26],  $\lambda$  binds them in a function expression. We will write  $fv(X)$  as a denotation of the set of all free names in expression  $X$ . Analogously,  $bn(X)$  will refer to the bound names.  $n(X) = fv(X) \cup bn(X)$ , i.e. the set of *all* names in the expression  $X$ .

### 3.3 Substitution

Substitution is a very common operation in most formal systems. It is often denoted by  $\sigma$ . We will write  $X\{a/b\}$  when we mean "the expression  $X$  with all free occurrences of  $b$  replaced by  $a$ ". This notation is used throughout this thesis.

### 3.4 Abstraction

An idea of  $\lambda$ -calculus is to establish a universal way of denoting a function [5]. Some notations, like  $f(x)$ , could be regarded as ambiguous or not explicit enough (they may hide several functions in one operator, etc).

**Example 1** *Consider:*

$$f(x) = x^2 \tag{1}$$

$$f(x + 1) = x^2 + 2x + 1 \tag{2}$$

Are these two above expressing the same function? Does the right hand side of (2) denote the function  $x \mapsto x^2$  or perhaps the function  $x \mapsto x^2 + 2x + 1$ ? Not only is this ambiguous, it also gets very clumsy when the notation is applied to higher-order functions (functions taking functions as parameters). Church suggested the following<sup>7</sup>:

$$f = \lambda x.x^2 \tag{3}$$

$$\lambda x.E \tag{4}$$

**Definition 3** *The function  $\lambda x.E$  is created by abstracting out the dependent variable  $x$  from the expression  $E$ .*

---

<sup>7</sup>Although different ways of writing a function exist, this notation is adopted in all functional languages, for example  
`fn x => x*x` (in Standard ML)  
`lambda(x)(* x x)` (in LISP)

The  $\lambda$ -notation<sup>8</sup> removes ambiguity [4] and clearly signals what are the dependent variables of a function definition. The  $\lambda$  *binds* the name  $x$  in the expression  $E$ , just as  $f$  does in integral calculus, or  $\exists, \forall$  do in predicate logic.

### 3.5 BNF Syntax for $\lambda$ -calculus

The Backus-Naur Form [29] for the syntax of the  $\lambda$ -calculus is

$$\begin{aligned} \langle \lambda\text{-term} \rangle & ::= \langle \text{variable} \rangle & (5) \\ & | \lambda \langle \text{variable} \rangle \langle \lambda\text{-term} \rangle \\ & | \langle \lambda\text{-term} \rangle \langle \lambda\text{-term} \rangle \\ \langle \text{variable} \rangle & ::= x \mid y \mid z \mid \dots \end{aligned}$$

### 3.6 Rules for manipulation of $\lambda$ -terms

A  $\lambda$ -calculus expression can be altered in two ways:

#### 3.6.1 $\alpha$ -conversion

This is the same as "renaming bound variables":

##### Definition 4

$$\lambda x.E = \lambda y.E\{y/x\}, \quad y \notin \text{fv}(E) \quad (6)$$

If the lambda-expression  $E$  can be altered to the lambda-expression  $F$ , by only applying zero or more  $\alpha$ -conversions, we say they are  $\alpha$ -convertible, written  $E \equiv_{\alpha} F$ .

This is a strong type of equality between  $\lambda$ -expressions (it is a *congruence*).

#### 3.6.2 $\beta$ -reduction

Let  $E = \lambda x.f(x)$  denote an abstraction of the process  $E = f$ . Then  $f(a)$  or  $f a$  denotes the *application* of  $f$  on a value  $a$ , and  $f a = E\{a/x\}$ , i.e. the expression  $E$  with every occurrence of " $x$ " replaced by " $a$ ". Formally [2],

$$(\lambda x.E) F \rightarrow E\{F/x\} \quad (7)$$

---

<sup>8</sup>The original idea was to be  $\hat{x}.f(x)$  or  $\hat{x}.f(x)$ , but typographical reasons changed this into the greek letter "lambda",  $\lambda$ .

where  $E$  and  $F$  are expressions. This is also known as  $\beta$ -reduction. The subexpression  $(\lambda x.E)F$  is called a *redex* (reducible expression).

**Definition 5** Two expressions  $E$  and  $F$  are  $\beta$ -convertible,  $E =_{\beta} F$ , if:

- 1) they are  $\alpha$ -convertible,  $E \equiv_{\alpha} F$
- 2)  $E$  can be  $\beta$ -reduced to  $F$  or  $F$   $\beta$ -reduced to  $E$
- 3)  $E$  can be  $\beta$ -reduced to  $G$  and  $F$   $\beta$ -reduced to  $G$

$\beta$ -conversion can be regarded as an equivalence relation between function expressions.

### 3.7 Currying

Two functions can be combined. If  $M$  and  $N$  are  $\lambda$ -functions, so is  $M N$ . We say that  $M N$  is a *curried function*, since it consists of two joint functions, that take their arguments one at a time.<sup>9</sup>

### 3.8 Combinators

A combinator is a  $\lambda$ -term without free variables. An alternative way of using  $\lambda$ -calculus would be using only combinators (i.e. no usage of free variables at all).<sup>10</sup> Combinators can be used for building more complex  $\lambda$ -terms. Also, they are "constant" with respect to  $\beta$ -reduction, the only way to use them with "effect" is currying them. (Since  $\beta$ -reduction only substitutes free occurrences of variables). They are recognized by their abbreviations. Some of the more well-known combinators are:

#### 3.8.1 Identity Combinator

$$I = \lambda x.x \tag{8}$$

The combinator  $I$ , when applied to any other  $\lambda$ -term  $X$ , returns  $X$  (which is the identity of  $X$ ).

#### 3.8.2 The Boolean Combinators

Data types can be modelled in  $\lambda$ -calculus. Two of the basic ones are the boolean constants, that can be modelled by:

$$True = \lambda t.\lambda f.t \tag{9}$$

$$False = \lambda t.\lambda f.f \tag{10}$$

---

<sup>9</sup>named after Curry, although suggested by the mathematician Schönfinkel!

<sup>10</sup>This is the main idea of Combinatory logic, developed by H.B. Curry.



### 3.8.3 "If" Combinator

$$\text{If} = \lambda x. \lambda y. \lambda z. x y z \quad (11)$$

The "If"-combinator has the property that the expression  $\text{If } U V W$  reduces to  $V$  if  $U = \text{True}$  and  $W$  else.

### 3.8.4 The Divergent Combinator

$$\Omega = (\lambda x. x x) (\lambda x. x x) \quad (12)$$

which, when the reduction rules are used, always reduces to itself. (It cannot be reduced to anything less complex than itself.)

### 3.8.5 Fixed-point Combinator

The combinator  $Y$  can be used for the definition of recursive functions. Let  $F$  stand for an arbitrary  $\lambda$ -function.

$$Y = \lambda f. ((\lambda x. f(x x)) f(x x)) \quad (13)$$

$Y$  creates the fixed-point for any given term  $F$ . This can be expressed as:

$$\forall F : YF =_{\beta} F(YF) \quad (14)$$

If  $X$  is the fixed point of  $F$  and  $\Lambda$  is the class of all lambda-terms,

$$\forall F \in \Lambda, \exists X \in \Lambda : FX = X \quad (15)$$

This follows from *the Fixed-Point Theorem*<sup>11</sup> The fixed-points play an important rôle in computer science. They are fundamental in the semantic definition of recursion. An ordinary (non-recursive) function can have any number of fixed points (especially none at all), but, according to the fixed-point theorem, every recursive function *must* have a fixed point. Intuitively, the fixed-point is where a recursion "stops" or "saturates". For the function  $f = \lambda x. x^2$  a fixed point is 1, since this is a point where the function does not increase or decrease.<sup>12</sup> The identity function  $I$  has (naturally) a fixed-point for all possible values.

---

<sup>11</sup>Formally: the Knaster-Tarski Fixed Point Theorem.

<sup>12</sup>and the other possibility is  $\infty$

### 3.9 A $\lambda$ -calculus example

**Example 2** We wish to express the Factorial function in  $\lambda$ -calculus:  
 $Factorial(x) = \text{if } (x=0) \text{ then } 1 \text{ else } x \cdot Factorial(x-1)$ .

$$Factorial = \lambda x. \text{If } (x=0) \ 1 \ x \cdot Factorial(x-1) \quad (16)$$

where  $(x=0)$  is a predicate evolving to *True* or *False* respectively.  $Factorial(3)$  can now be calculated as:

$$\begin{aligned} & Factorial \ 3 \\ & \lambda x. \text{If } (x=0) \ 1 \ x \cdot Factorial(x-1) \ 3 \\ =_{\beta} & \text{If } (3=0) \ 1 \ 3 \cdot Factorial(3-1) \\ =_{\beta} & \text{If } \text{False} \ 1 \ 3 \cdot Factorial \ 2 \\ =_{\beta} & 3 \cdot Factorial \ 2 \\ =_{\beta} & 3 \cdot (\lambda x. \text{If } (x=0) \ 1 \ x \cdot Factorial(x-1) \ 2) \\ =_{\beta} & 3 \cdot (\text{If } (2=0) \ 1 \ 2 \cdot Factorial(2-1)) \\ =_{\beta} & 3 \cdot (\text{If } \text{False} \ 1 \ 2 \cdot Factorial \ 1) \\ =_{\beta} & 3 \cdot (2 \cdot Factorial \ 1) \\ =_{\beta} & 3 \cdot (2 \cdot (\lambda x. \text{If } (x=0) \ 1 \ x \cdot Factorial(x-1)) \ 1) \\ =_{\beta} & 3 \cdot (2 \cdot (\text{If } (1=0) \ 1 \ 1 \cdot Factorial(1-1))) \\ =_{\beta} & 3 \cdot (2 \cdot (\text{If } \text{False} \ 1 \ 1 \cdot Factorial \ 0)) \\ =_{\beta} & 3 \cdot (2 \cdot (Factorial \ 0)) \\ =_{\beta} & 3 \cdot (2 \cdot (\lambda x. \text{If } (x=0) \ 1 \ x \cdot Factorial(x-1)) \ 0) \\ =_{\beta} & 3 \cdot (2 \cdot (\text{If } (0=0) \ 1 \ 0 \cdot Factorial(0-1))) \\ =_{\beta} & 3 \cdot (2 \cdot (\text{If } \text{True} \ 1 \ 0 \cdot Factorial(0-1))) \\ =_{\beta} & 3 \cdot (2 \cdot (1)) \\ =_{\beta} & 6 \end{aligned} \quad (17)$$

### 3.10 Normal Forms

$\lambda$ -terms can be written on a Normal Form, which is a redex-free form.<sup>13</sup> The normal form is *unique* for every different function; if two  $\lambda$ -expressions,  $L$  and  $M$  can be reduced to the same normal form  $N$ , they are  $\beta$ -convertible.

**Definition 6** A  $\lambda$ -term, that is  $\beta$ -reduced until no further redexes exist, is said to be in *normal form*.

---

<sup>13</sup>The divergent operator lacks normal form, since it can always be reduced, but will never reach a redex-free form.

The normal form can be regarded as the "original" or "basic" form of an  $\lambda$ -term. One may view it as the fundamental "meaning" (in a semantic way) of a term.

### 3.11 The Reduction Strategies

When a  $\lambda$ -term contains several redexes, we can choose which order to reduce them. There are three major strategies when reducing a  $\lambda$ -term:

#### 3.11.1 Normal Order

Reduces the redex whose  $\lambda$  appears furthest to the left.

#### 3.11.2 Lazy Reduction

Reduce the leftmost redex if this is not itself in the body of another abstraction.

#### 3.11.3 Applicative Reduction

Always choose leftmost redex  $(\lambda x.E)F$  where  $F$  is *NOT* a redex itself.

**Example 3** *In the expression*

$$\lambda x (\lambda y.y) ((\lambda z.z) (s (\lambda t.t))) \tag{18}$$

*the normal-order evaluation would reduce the term  $(\lambda y.y) ((\lambda z.z) (s (\lambda t.t)))$  first, the applicative order would reduce the term  $((\lambda z.z) (s (\lambda t.t)))$  first, while lazy evaluation wouldn't reduce it at all.*

Normal and lazy-order reduction evaluation order corresponds to the "call-by-name" argument passing of programming languages such as Algol-60 (since they do not calculate the values of the expression(s) that are used as arguments before calling another function), whereas applicative order corresponds to "call-by-value" as in ML and Scheme (where no function call is done before all arguments are developed to values).

### 3.12 De Bruijn-indices

A difficulty when doing calculations in the  $\lambda$ -calculus is the problem of not mixing up free and bound occurrences of a variable. In  $(\lambda x.x) x$ ,  $x$  occurs both bound and free. Therefore, we reason about bound/free *occurrences* of a name, rather than bound/free names. As seen in the section regarding  $\alpha$ -conversion, the bound occurrence of a name can always be altered

to any other name, not already occurring in the expression, without the meaning of the expression being changed. It is obvious that the choice of a bound variable is unimportant. This led the dutch mathematician De Bruijn [35] to suggest that bound names should be represented as integers rather than names; the number index would show how far the bound name is from its binder (the "λ"). The bound name with its binder closest to it is thus given the "name" 0, the second closest is called 1, etc. Intuitively, the index tells how many binding λ:s one has to pass from name to binder.

**Example 4** *let  $\equiv_{DB}$  stand for the debruijn-indexed equivalence between λ-expressions:*

$$\lambda w.(\lambda x.(\lambda y.y)x)w)x \equiv_{DB} \lambda(\lambda.1\ 0\ 2)0\ 0 \quad (19)$$

Not only does this remove some ambiguity between free and bound name occurrences, (free occurrences are not replaced by an index), it turns out that  $\alpha$ -equivalent expressions always will look the same, *syntactically*. This means that no  $\alpha$ -conversion is necessary between DB-indexed expressions.

## 4 The $\pi$ -calculus

If every expression in  $\lambda$ -calculus denoted a function, every expression in  $\pi$ -calculus denotes a *process* [7]. A system of functions simply cannot model concurrent computing. We need another notion – the process. A process is a much more independent object than a function – it may have its own memory, variables and context, it may run independently, in parallel with other processes, and it may interact with other processes through communication. In  $\pi$ -calculus, the fundamental entity is a process, and the communication is represented as processes exchanging names over channels. Since a channel is named, the name of it can be passed from one process to another. The receiving process can then itself use the channel for passing names. This is the mechanism for changing the structure of a system – processes may be spawned off from other processes, channels may be set up (and destroyed) between them, and the whole system can change all of its processes and channels, (i.e. its entire structure) over time. This alongside ”ordinary” communication of data.

### 4.1 Actions and names

There are three types of actions:

- 1) Input actions, denoted by their names:  $a, b, c\dots$
- 2) Output actions, denoted by  $\bar{a}, \bar{b}, \bar{c}\dots$
- 3) The *silent* or ”perfect” or ”internal” or ”non-observable” action, denoted  $\tau$

**Definition 7** If  $\mathcal{L}$  is the set of names for a system, the set of actions,  $\mathcal{A}$  is  $\mathcal{L} \cup \bar{\mathcal{L}}$ . The entire set of actions for any system is then  $\mathcal{A} \cup \{\tau\}$ .

When using *value passing*, we write  $a(x)$  for meaning ”input a value for the name  $x$  on port  $a$ ”,  $\bar{a}(e)$  or  $\bar{a}e$  meaning ”output value  $e$  on port  $a$ ” where  $a \in \mathcal{A}$ , and  $x, e \in \mathcal{L}$ .

#### 4.1.1 $\alpha$ -conversion

If the processes  $P$  and  $Q$  only differs in their naming of bound variables, we say they are alpha-convertible, i.e. if  $P$  can be transformed to  $Q$  merely by acts of substitution on its bound variable names, we write  $P \equiv_\alpha Q$  as before. The  $\pi$ -calculus does not distinguish these processes.  $fn(P)$  ( $bn(P)$ ) denotes the set of free (bound) names in  $P$ . All names occurring in  $P$  is written  $n(P) = fn(P) \cup bn(P)$ . Similarly, actions can be free or bound, depending on whether they contain a free or a bound name.<sup>14</sup>

---

<sup>14</sup>This reasoning is not applicable for the silent action,  $\tau$ , since it does not contain a name at all, and thus cannot be neither free nor bound.

### 4.1.2 Substitution

A substitution is written as  $\{y/x\}$  and means "replacing all free occurrences of  $x$  with  $y$ ", as before.  $\sigma$  usually denotes a substitution.

## 4.2 Labeled Transitions

The semantics of the  $\pi$ -calculus can be described as a *labeled transition system*, "LTS", just as CCS [8]<sup>15</sup>. A simple example of a LTS is:

$$P \xrightarrow{\alpha} Q \tag{20}$$

meaning "process  $P$  can perform  $\alpha$  and then behave as process  $Q$ ", where  $\alpha$  is an *action*. We will always use capital letters like  $P, Q, R, S, \dots$  to denote processes,  $a, b, c, \dots$  to stand for *names*, and  $\alpha, \beta, \gamma, \dots$  to stand for *actions*.  $\hat{t} \in \mathcal{A} \cup \{\tau\}$  means the action sequence  $t$  with all occurrences of  $\tau$  removed. Thus,  $\hat{\tau}^n = \epsilon$ , the empty sequence, for any value of  $n$ .

$\xrightarrow{\alpha}$  is an totally specified action, the action  $\alpha$  can also be  $\tau$ .

$\xrightarrow{\hat{\alpha}}$  is an action *sequence* that at least specifies the  $\alpha$ -actions contained within.

$\xrightarrow{\hat{\alpha}}$  is an action sequence that says *nothing* about any  $\tau$ -actions that might be inside. Thus,

$\xrightarrow{\hat{\alpha}} = (\xrightarrow{\tau})^*$  if  $\alpha = \tau$ , and  $(\xrightarrow{\tau})^*(\xrightarrow{\alpha})(\xrightarrow{\tau})^*$  if  $\alpha \neq \tau$ .

A non-labelled arrow like  $\Rightarrow$  means  $(\xrightarrow{\tau})^*$ .

## 4.3 Definition

The syntax of the  $\pi$ -calculus can be described by a BNF-equation: [20]

$$P ::= \sum P_i \mid \alpha.P \mid P|Q \mid !P \mid (\nu x)P \mid [x = y]P \mid A(y_1, \dots, y_n) \tag{21}$$

### 4.3.1 Summation

$\sum_{i \in I} P_i$  (where  $I$  is a finite indexing set): Behaves as one of the agents  $P_i$ . If  $i=0$  we typically have the empty summation, inaction, also denoted  $\mathbf{0}$ .

$$SUM \frac{P_i \xrightarrow{\alpha} P'_i}{P_i + P_j + \dots + P_n \xrightarrow{\alpha} P'} \tag{22}$$

If  $P_i$  can perform  $\alpha$  and become  $P'_i$ , then  $P_i + P_j + \dots + P_n$  can perform  $\alpha$  and become  $P'_i$ . Summation typically stands for *alternative* ways of execution.

<sup>15</sup>CCS can be regarded as a static sub-calculus of the  $\pi$ -calculus.

### 4.3.2 Prefix

$\alpha.P$  performs the action  $\alpha$  and then behaves as agent  $P$ , where  $\alpha$  is one of the following actions:

(I) The *silent action*  $\tau$ , polarity: 0

$$TAU - ACT \frac{}{\tau.P \xrightarrow{\tau} P} \quad (23)$$

(II) The *free output action*,  $\bar{x}y$ , "transmit the name  $y$  on channel  $x$ ", polarity: minus.

(III) The *bound output action*,  $\bar{x}(y)$ , which is short for  $(\nu y)\bar{x}y$ .

$$OUTPUT - ACT \frac{}{\bar{x}y.P \xrightarrow{\bar{x}y} P} \quad (24)$$

(IV) The *input action*,  $x(y)$ , "receive a name on channel  $x$ , for the location  $y$ ", polarity: plus.

$$INPUT - ACT \frac{}{x(z).P \xrightarrow{x(w)} P\{w/z\}} w \notin fn((\nu z)P) \quad (25)$$

In (II) – (IV) above,  $x$  is said to be the *subject* and  $y$  is the *object* (parameter).

### 4.3.3 Composition

The agent  $P|Q$  behaves as the agents  $P$  and  $Q$  run in parallel. They may act independently, or communicate between each other.

$$PAR \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} bn(\alpha) \cap fn(Q) = \emptyset \quad (26)$$

$$COMM \frac{P \xrightarrow{\bar{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{P|Q \xrightarrow{\tau} P'|Q'\{y/z\}} \quad (27)$$

$$CLOSE \frac{P \xrightarrow{(\nu w)\bar{x}y} P' \quad Q \xrightarrow{x(w)} Q'}{P|Q \xrightarrow{\tau} (\nu w)(P'|Q')} \quad (28)$$

The CLOSE-rule shows *scope intrusion*, a bound output is run parallel to an input. The result is that the restriction on  $w$  is extended to incorporate the whole parallel agent  $P|Q$ . There are of course rules for the symmetrical cases of PAR, COMM, CLOSE.

#### 4.3.4 Replication

$$REP \quad !P \equiv P \mid !P \quad (29)$$

A parallel instance of process  $P$  can be "spawned" or "forked" from another process. The relation " $\equiv$ ", the structural congruence for processes, as defined as :

**Definition 8** *Let  $P$  and  $Q$  be two  $\pi$ -calculus processes. The structural congruence between them is written  $P \equiv Q$  and is a commutative and associative relation defined as:*

$$P \mid Q \equiv Q \mid P \quad (30)$$

$$(P \mid Q) \mid R \equiv P \mid (Q \mid R) \quad (31)$$

$$((\nu x)P) \mid Q \equiv (\nu x)(P \mid Q) \text{ if } x \notin \text{fn}(Q) \quad (32)$$

$$!P \equiv P \mid !P \quad (33)$$

The structural congruence will be extended to  $\pi$ -calculus agents (a special case of processes) in section 4.8.

#### 4.3.5 Restriction

$$RES \quad \frac{P \xrightarrow{\alpha} P'}{(\nu x)P \xrightarrow{\alpha} (\nu x)P} \quad x \notin n(\alpha) \quad (34)$$

The restriction<sup>16</sup>  $(\nu x)P$  denotes that the name  $x$  is *private* in process  $P$ . ("new  $x$  in  $P$ ") It is its own instance, other names  $x$  can exist outside of  $P$ , they then do not refer to the same name. The ports  $x$  and  $\bar{x}$  are prohibited as subjects in actions.

$$OPEN \quad \frac{P \xrightarrow{\bar{x}y} P'}{(\nu y)P \xrightarrow{\bar{x}(w)} P'\{w/y\}} \quad y \neq x, \quad w \notin \text{fn}(\nu y)P) \quad (35)$$

If process  $P$  emits the private name  $w$  on channel  $x$ , it thereafter must continue as an agent without the restriction for  $w$ . Since it has transmitted that name, it is now not private. This is a *scope extrusion*.

#### 4.3.6 Match

$$MATCH \quad \frac{P \xrightarrow{\alpha} P'}{[x = x]P \xrightarrow{\alpha} P'} \quad (36)$$

$[x = y]P$  behaves as  $P$  if  $x$  and  $y$  are identical names, as  $\mathbf{0}$  otherwise.

<sup>16</sup>Note that the Greek letter "nu" ( $\nu$ ) should not be confused with the maximum-fixpoint denotation or the  $\nu$ -calculus!



### 4.3.7 Definition

$$DEF \frac{P\{\tilde{y}/\tilde{x}\} \xrightarrow{\alpha} P'}{A(\tilde{y}) \xrightarrow{\alpha} P'} \quad A(\tilde{x}) \triangleq P \quad (37)$$

(We write  $\tilde{x}$  as shorthand for a whole vectore of names,  $x_1, \dots, x_n$ .) A defined agent  $A$ , of arity  $n$  is defined as  $A(x_1, \dots, x_n) \triangleq P$ . This definition of agent equations allows recursive agents. Any identifier  $A$  can occur in the defining equation of  $A$ . The names  $x_1, \dots, x_n$  are the only names allowed to occur free in  $P$ .  $A(y_1, \dots, y_n)$  behaves like  $P(\{y_1/x_1\} \dots \{y_n/x_n\})$ .

## 4.4 Binding of names

Two operators *bind* names; the input prefix  $x(y).P$  binds the name  $y$  in  $P$ , and the restriction operator,  $(\nu x)P$  binds the name  $x$  in  $P$ . Names that are not bound by one of those operators are considered free. Note that sometimes  $(\nu y)\bar{x}y$  is abbreviated " $\bar{x}(y)$ " as an action<sup>17</sup>.

## 4.5 Abstraction & Concretion

An abstraction is another way of writing the input prefix:

$$x(y).P \triangleq x.(\lambda y)P \quad (38)$$

where  $x$  is considered the location of the value-to-be input, and  $(\lambda y)$  the abstraction of the value  $y$  from  $P$ . A major difference to the  $\lambda$ -calculus abstraction is that in  $\pi$ -calculus, **only names** can be abstracted.<sup>18</sup> When communication is involved, we may say that  $x$  has to synchronize with its co-name of another agent before any value-passing can take place. Also note that in abstraction, we regard  $x$  as prefixing the abstraction. A concretion is analogously a way of re-writing the output operation:

$$\bar{x}y_1 \dots y_n.P \triangleq \bar{x}.[y_1 \dots y_n]P \quad (39)$$

where  $\bar{x}$  is the co-location and  $[y_1 \dots y_n]$  is the concretion of agent  $P$ , where each  $y_i$  is a datum (name) of the concretion, and the arity of the concretion is  $n$ . Communication can only take place when the arities of the abstraction agent and concretion agent are the same.

## 4.6 Monadic / Polyadic

If the input and output operators only are capable of receiving *exactly* one name at a time, the calculus is said to be *monadic*. Likewise, if they instead are capable of interchanging a

<sup>17</sup>Not in the agent prefix form.

<sup>18</sup>In  $\lambda$ -calculus, we saw that arbitrary expressions could be the object of abstraction.

whole vector of zero names or more at a time (this is considered an *atomic*<sup>19</sup> transaction),

$$x(\vec{y}).P \mid \overline{xz}.Q \xrightarrow{\tau} P\{z_1/y_1, \dots, z_n/y_n\} \mid Q \quad (40)$$

where each  $z_i/y_i$  denotes a substitution of every occurrence of  $y_i$  with  $z_i$ , the calculus is said to be *polyadic*.

## 4.7 Examples

**Example 5** *A very illuminating example of a simple  $\pi$ -calculus process is the memory cell:*

$$MEM(x) \triangleq \overline{out}x.MEM(x) + in(y).MEM(y) \quad (41)$$

*The memory cell  $MEM$  can either output its contents,  $x$  and then continue as  $MEM(x)$  (i.e. as itself), or input another value,  $y$ , and then continue as  $MEM(y)$ , as itself but with another content.*

## 4.8 Agents, Structural Congruence

### 4.8.1 Agents as a special case of processes

Introducing abstractions and concretions (the polyadic  $\pi$ -calculus), we now distinguish between *agents* and *processes*. An *agent* is thus defined as either an abstraction or a concretion: Let  $N$  stand for "normal process",  $P$  for *any* process,  $F$  for abstraction and  $C$  for concretion, and  $A$  for agent. The difference between these notions is explained by:

$$\begin{aligned} N & ::= \alpha.A \mid \mathbf{0} \mid \mathbf{N} + \mathbf{N} \\ P & ::= N \mid P \mid P \mid !P \mid (\nu x)P \\ F & ::= P \mid (\lambda x)F \mid (\nu x)F \\ C & ::= P \mid [x]C \mid (\nu x)C \\ A & ::= F \mid C \end{aligned} \quad (42)$$

### 4.8.2 Structural congruence for agents

If we wish to reason algebraically about agents, we need some rules for simplifying/re-writing agent expressions. Let  $A, F, C$  denote agents as above.

**Definition 9** *1) If  $A$  and  $B$  are  $\alpha$ -congruent,  $A \equiv_\alpha B$ , then they are structurally congruent, written  $A \equiv B$ .*

---

<sup>19</sup>Atomic means that all substitutions are regarded as taking place simultaneously.

- 2)  $!P \equiv P$
- 3)  $(\nu x)\mathbf{0} \equiv \mathbf{0}$
- 4)  $(\nu x)(\nu y)A \equiv (\nu y)(\nu x)A$
- 5) If  $x \notin fn(P)$ , then  $(\nu x)(P|Q) \equiv P|(\nu x)Q$
- 6)  $(\nu y)(\lambda x)F \equiv (\lambda x)(\nu y)F$ , ( $x \neq y$ )
- 7)  $(\nu x)[x]C \equiv [x](\nu y)C$ , ( $x \neq y$ )
- 8)  $(\nu x)(\nu x)A \equiv (\nu x)A$

## 4.9 Normal form

The structural congruence rules can be used for re-writing an agent expression into a *normal form*. For abstractions, this is done by "pushing restriction inwards", and, for restrictions that is a part of a concretion, these are pushed outwards. So, the normal forms for the abstraction  $F \equiv (\lambda \tilde{x})P$ , and for the concretion  $C \equiv (\nu \tilde{x})[\tilde{y}]P$ ,  $\tilde{x} \leq \tilde{y}$ , and, if the concretion lacks any private names,  $C \equiv [\tilde{x}]P$ , For aspects of normal forms, the same as for  $\lambda$ -calculus function expressions is applicable here.

## 4.10 Application

Application on  $\pi$ -calculus agents look very much like that for the  $\lambda$ -calculus:

$$((\lambda x)F)y \triangleq F\{y/x\} \quad (43)$$

### 4.10.1 Pseudo-application

Let  $\vec{x}, \vec{y}$  denote *vectors* of names. The pseudo-application  $A \bullet B$  of an abstraction and a concretion, both on normal form, is defined as:

Let  $A \equiv (\lambda \vec{x})P$  and  $B \equiv (\nu \vec{z})[\vec{y}]Q$  where  $\vec{x} \cap \vec{z} = \emptyset$  and  $|\vec{x}| = |\vec{y}|$ . Then:

$$A \bullet B \triangleq (\nu \vec{z})(P\{\vec{y}/\vec{x}\} | Q) \quad (44)$$

The pseudo-application is useful in the rule *COMM* of the *commitments* (below).

## 4.11 Commitments

If  $P$  can perform the action  $\alpha$  (and then behave as  $A$ ), we say that this is a *commitment* for  $P$  to act upon  $\alpha$ .

$$P \succ \alpha.A \quad (45)$$

is interpreted as "P can commit to  $\alpha.A$ ". The rules for the commitment relation are:

**Definition 10**

$$SUM \frac{-}{\dots + \alpha.A} \succ \alpha.A \quad (46)$$

$$COMM \frac{P \succ x.F \quad Q \succ \bar{x}.C}{(P \mid Q) \succ \tau.(F \bullet C)} \quad (47)$$

$$PAR \frac{P \succ \alpha.A}{P \mid Q \succ \alpha.A} \quad (48)$$

$$RES \frac{P \succ \alpha.A}{(\nu x)P \succ \alpha.(\nu x)A} \text{ if } \alpha \notin \{x, \bar{x}\} \quad (49)$$

$$STRUCT \frac{Q \equiv P \quad P \succ \alpha.A \quad A \equiv B}{Q \succ \alpha.B} \quad (50)$$

The commitment relation will be of much use later. The way  $\pi$ -calculus agents are represented in the model checking algorithm is as a set of commitments. Note the usage of pseudo-application in the COMM-rule. This is needed since parallel composition is not defined for commitments. The value-passing that can take place in parallel composition is not a part of the commitment relation, which only deals with the subjects of agents (actions including  $\tau$ ). The pseudo-application, " $\bullet$ " deals with this; value-passing can only take place when the arities of the abstraction and the concretion are the same. But any two agents can be parallel-composed, and, if the arities are not equal, no value-passing takes place.

## 4.12 Distinction

**Definition 11** *A distinction is a symmetric, irreflexive binary relation between names.*

It is irreflexive since a name cannot be distinct from itself. Let  $D$  range over Distinctions. Any substitution  $\sigma$  preserves a distinction if

$$\forall x, y \in D : x\sigma \neq y\sigma \quad (51)$$

Distinctions will be of use when we reason about bisimulation equivalences later on; for some equivalences to hold, certain names must be kept apart. This is then done by using a distinction. We will write  $=_D$ , meaning an equivalence relation that holds under the distinction  $D$ , e.g.  $\sim_{\{x, y\}}$  for a strong bisimulation equivalence when  $x \neq y$ , as seen in the next section.

## 4.13 Equivalence relations for processes

We have seen that  $\alpha$ -congruence and structural congruence are two equivalence relations between  $\pi$ -calculus processes. Another type of equivalence relation, that also existed in CCS, is the notion of *bisimulation equivalence*. It is roughly divided into strong and weak

types. Intuitively, if process  $P$  can do everything<sup>20</sup> that process  $Q$  can,  $P$  is similar to  $Q$ . If the opposite also holds,  $P$  and  $Q$  are *bi*-similar. Here we first introduce three different *strong*<sup>21</sup> bisimulation equivalences:

## 4.14 Strong Bisimulation Equivalences

### 4.14.1 Late

Strong Late Bisimulation Equivalence  $\sim_{\mathcal{L}}$  is defined as:

**Definition 12**  $\mathcal{L}_{\mathcal{S}}$  is a late simulation if  $(P, Q) \in \mathcal{L}_{\mathcal{S}}$  implies that ( $\alpha = \text{free action}$ )

$$\text{If } P \xrightarrow{\alpha} P' \text{ then } \exists Q'. Q \xrightarrow{\alpha} Q' \text{ and } (P', Q') \in \mathcal{L}_{\mathcal{S}} \quad (52)$$

$$\text{If } P \xrightarrow{x(y)} P' \text{ then } \exists Q' : Q \xrightarrow{x(y)} Q' \text{ and } \forall w : (P'\{w/y\}, Q'\{w/y\}) \in \mathcal{L}_{\mathcal{S}} \quad (53)$$

$$\text{If } P \xrightarrow{(\nu y)\bar{x}y} P' \text{ and } y \notin n(P, Q) \text{ then } \exists Q'. Q \xrightarrow{(\nu y)\bar{x}y} Q' \text{ and } (P', Q') \in \mathcal{L}_{\mathcal{S}} \quad (54)$$

If both  $\mathcal{L}_{\mathcal{S}}$  and its inverse are simulations, then  $\mathcal{L}_{\mathcal{S}}$  is a *bisimulation*. We write  $P \sim_{\mathcal{L}} Q$  if  $P$  and  $Q$  are strong late bisimilar, i.e. related by a strong late bisimulation.

### 4.14.2 Early

Strong *Early* bisimulation Equivalence,  $\sim_{\mathcal{E}}$  is defined as:

**Definition 13**  $\mathcal{E}_{\mathcal{S}}$  is an early simulation if  $(P, Q) \in \mathcal{E}_{\mathcal{S}}$  implies that (for the free action  $\alpha$ ):

$$\text{If } P \xrightarrow{\alpha} P' \text{ then } \exists Q'. Q \xrightarrow{\alpha} Q' \text{ and } (P', Q') \in \mathcal{E}_{\mathcal{S}} \quad (55)$$

$$\text{If } P \xrightarrow{x(y)} P' \text{ then } \forall w : \exists Q' : Q \xrightarrow{x(y)} Q' \text{ and } (P'\{w/y\}, Q'\{w/y\}) \in \mathcal{E}_{\mathcal{S}} \quad (56)$$

$$\text{if } P \xrightarrow{(\nu y)\bar{x}y} P' \text{ and } y \notin n(P, Q) \text{ then } \exists Q'. Q \xrightarrow{(\nu y)\bar{x}y} Q' \text{ and } (P', Q') \in \mathcal{E}_{\mathcal{S}} \quad (57)$$

If both  $\mathcal{E}_{\mathcal{S}}$  and its inverse are simulations, then  $\mathcal{E}_{\mathcal{S}}$  is a bisimulation. We write  $P \sim_{\mathcal{E}} Q$  if  $P$  and  $Q$  are related by a strong early bisimulation. This variant (commuting the quantifiers in step 2) is weaker<sup>22</sup> than strong late bisimulation, since the rule only requires that for each instance of the object, there exists an simulating transition, while, in the late bisimulation, an simulating input transition must be able to simulate *all* possible instances.

---

<sup>20</sup>Perform all actions

<sup>21</sup>The notion of *weak* bisimulation equivalence is investigated in the next section.

<sup>22</sup>"weaker" here means that the relation is larger, i.e. the conditions for being in the relation are relaxed

### 4.14.3 Open

Let  $\sigma$  stand for a substitution of names.  $x$  and  $y$  can be vectors of names.  $D$  is a distinction. The strong *open* bisimulation equivalence is a symmetric binary relation and is defined as:

**Definition 14** A symmetric binary relation  $\mathcal{O}_{\mathcal{S}_D}$  is an indexed open bisimulation if  $(P, Q) \in \mathcal{O}_{\mathcal{S}_D}$  implies that for all substitutions  $\sigma$ , and the free action  $\alpha$ :

$$P\sigma \xrightarrow{\alpha} P', \exists Q'. Q\sigma \xrightarrow{\alpha} Q' \text{ and } (P', Q') \in \mathcal{O}_{\mathcal{S}_D} \quad (58)$$

$$P\sigma \xrightarrow{a(x)} P', \exists Q'. Q\sigma \xrightarrow{a(x)} Q', \text{ and } (P', Q') \in \mathcal{O}_{\mathcal{S}_D} \quad (59)$$

$$P\sigma \xrightarrow{(\nu x)a(x)} P', \exists Q'. Q\sigma \xrightarrow{(\nu x)a(x)} Q', \text{ and } (P', Q') \in \mathcal{O}_{\mathcal{S}_{D'}}, \text{ where } D' = D\sigma \cup (x \times \text{fn}(P\sigma, Q\sigma)) \quad (60)$$

If both  $\mathcal{O}_{\mathcal{S}_D}$  and its inverse are simulations, then  $\mathcal{O}_{\mathcal{S}_D}$  is a *bisimulation*. We write  $P \sim_{\mathcal{O}_D} Q$  if  $P$  and  $Q$  are strong open bisimilar, i.e. related by a strong open bisimulation.

### 4.14.4 Difference between bisimulation equivalences.

The relation between the three different types of bisimulation equivalence can be expressed as

$$\mathcal{O}_{\mathcal{S}} \subset \mathcal{L}_{\mathcal{S}} \subset \mathcal{E}_{\mathcal{S}} \quad (61)$$

The congruent open bisimulation is the strongest (smallest) relation. Intuitively, the difference between them can be described as:

The *late* bisimulation equivalence instantiates a bound name *after* the synchronization has taken place<sup>23</sup>, the *early* does this just *before* the synchronization will take place. Thus, the early bisimulation equivalence will have more options than the late for selecting actions in the other agent, in order for the search to go on further into respective agents, to check whether the relation will still hold. As for the *open*, it does not instantiate any bound names at all until necessary.

### 4.14.5 Strong bisimulation equivalence and congruence

Early and Late bisimulation equivalences are not preserved when substitution of names is concerned.

**Example 6** Let  $P \triangleq x$  and  $Q \triangleq \bar{y}$ . Is  $P\sigma|Q\sigma \sim x.\bar{y} + \bar{y}.x$ ? Yes, if  $\sigma$  does not contain the substitution  $\{y/x\}$ . Then  $P\sigma|Q\sigma \sim x.\bar{y} + \bar{y}.x + \tau$ . We see that keeping names apart (not applying a substitution) is a way to make the bisimulation equivalence hold.

<sup>23</sup>Synchronization means here "the selection of an action"

**Example 7** As a result of above are these equations:

$$\bar{x} \mid y \overset{\bullet}{\sim} \bar{x}.y + y.\bar{x} \quad (62)$$

$$\bar{x} \mid x \not\overset{\bullet}{\sim} x.\bar{x} + \bar{x}.x \quad (63)$$

$$\bar{x} \mid x \overset{\bullet}{\sim} x.\bar{x} + \bar{x}.x + \tau \quad (64)$$

The notation " $\overset{\bullet}{\sim}$ " is here either the early or late bisimulation equivalence. Putting a dot over the relation symbol is a way of denoting a *non-congruent* relation.

The open bisimulation equivalence, on the other hand, is defined as an equivalence holding in all contexts, which makes it a congruent relation.

Distinctions, keeping names apart, is another way of strengthening a bisimulation relation:

**Example 8**

$$\bar{x} \mid y \sim_{\{x,y\}} \bar{x}.y + y.\bar{x} \quad (65)$$

*This holds since the two names  $x, y$  must be kept distinct. An internal communication resulting in a  $\tau$ -action cannot take place. The relation  $\sim_{\{x,y\}}$  is a relation under a distinction, as before.*

## 4.15 Weak Bisimulation Equivalences

As CCS, the  $\pi$ -calculus has a notion of *weak* bisimulation equivalence. The difference to strong is the view upon *silent actions*,  $\tau$ . These are ignored in the definition of the relation. There are weak versions of all three previously presented strong bisimulation equivalences. Sometimes, one may wish to ignore the internal actions of a system when deciding whether two processes are equal or not. Then the weak bisimulation equivalence is a useful relation.

### 4.15.1 Late

Weak Late Bisimulation Equivalence  $\approx_{\mathcal{L}}$  is defined as:

**Definition 15**  $\mathcal{L}_{\mathcal{W}}$  is a late weak simulation if  $(P, Q) \in \mathcal{L}_{\mathcal{W}}$  implies that

$$\text{If } P \xrightarrow{\alpha} P' \text{ then } \exists Q'. Q \overset{\hat{\alpha}}{\Rightarrow} Q' \text{ and } (P', Q') \in \mathcal{L}_{\mathcal{W}} \quad (66)$$

$$\text{If } P \xrightarrow{x(y)} P' \text{ then } \exists Q' : Q \Rightarrow \xrightarrow{x(y)} Q' \text{ and } \forall w : (P'\{w/y\}, Q'\{w/y\}) \in \mathcal{L}_{\mathcal{W}} \quad (67)$$

$$\text{If } P \xrightarrow{(\nu y)\bar{x}y} P' \text{ and } y \notin n(P, Q) \text{ then } \exists Q'. Q \Rightarrow \xrightarrow{(\nu y)\bar{x}y} Q' \text{ and } (P', Q') \in \mathcal{L}_{\mathcal{W}} \quad (68)$$

If both  $\mathcal{L}_{\mathcal{W}}$  and its inverse are simulations, then  $\mathcal{L}_{\mathcal{W}}$  is a *bisimulation*. We write  $P \approx_{\mathcal{L}} Q$  if  $P$  and  $Q$  are weak late bisimulation equivalent.

#### 4.15.2 Early

Weak *early* bisimilarity,  $\approx_{\mathcal{E}_W}$  is defined as:

**Definition 16**  $\mathcal{E}_W$  is a weak early simulation if  $(P, Q) \in \mathcal{E}_W$  implies that

$$\text{If } P \xrightarrow{\alpha} P' \text{ then } \exists Q'. Q \xrightarrow{\bar{\alpha}} Q' \text{ and } (P', Q') \in \mathcal{E}_W \quad (69)$$

$$\text{If } P \xrightarrow{x(y)} P' \text{ then } \forall w : \exists Q' : Q \xrightarrow{x(y)} Q' \text{ and } (P'\{w/y\}, Q'\{w/y\}) \in \mathcal{E}_W \quad (70)$$

$$\text{If } P \xrightarrow{(\nu y)\bar{x}y} P' \text{ and } y \notin n(P, Q) \text{ then } \exists Q'. Q \xrightarrow{(\nu y)\bar{x}y} Q' \text{ and } (P', Q') \in \mathcal{E}_W \quad (71)$$

If both  $\mathcal{E}_W$  and its inverse are simulations, then  $\mathcal{E}_W$  is a bisimulation. We write  $P \approx_{\mathcal{E}} Q$  if  $P$  and  $Q$  are weak early bisimulation equivalent.

#### 4.15.3 Open

Let  $\sigma$  stand for a substitution of names.  $x$  and  $y$  can be vectors of names. The *open* bisimulation equivalence is a symmetric binary relation and is defined as:

**Definition 17** A symmetric binary relation  $\mathcal{O}_{W_D}$  is an indexed open bisimulation if  $(P, Q) \in \mathcal{O}_{W_D}$  implies that for all substitutions  $\sigma$ , and the free action  $\alpha$ :

$$P\sigma \xrightarrow{\alpha} P', \exists Q'. Q\sigma \xrightarrow{\bar{\alpha}} Q' \text{ and } (P', Q') \in \mathcal{O}_{W_D} \quad (72)$$

$$P\sigma \xrightarrow{a(x)} P', \exists Q'. Q\sigma \xrightarrow{a(x)} Q', \text{ and } (P', Q') \in \mathcal{O}_{W_D} \quad (73)$$

$$P\sigma \xrightarrow{(\nu x)a(x)} P', \exists Q'. Q\sigma \xrightarrow{(\nu x)a(x)} Q', \text{ and } (P', Q') \in \mathcal{O}_{W_{D'}}, \text{ where } D' = D\sigma \cup (x \times \text{fn}(P\sigma, Q\sigma)) \quad (74)$$

If both  $\mathcal{O}_{W_D}$  and its inverse are simulations, then  $\mathcal{O}_{W_D}$  is a bisimulation. We write  $P \approx_{\mathcal{O}_D} Q$  if  $P$  and  $Q$  are weak open bisimilar, i.e. related by a weak open bisimulation.

#### 4.15.4 Difference between bisimulation equivalences.

The exact relation between the three different types of weak bisimulation equivalence is still unexplored research topics.

### 4.16 Properties for Weak Bisimulation Equivalences

We see that the difference between strong and weak bisimulation equivalence is the relaxed condition for the "simulating" process ( $Q$ ); in strong bisimulation equivalence, there must exist exactly one corresponding step in  $Q$  for every step in  $P$ , in the weak version, there can



be one or several steps. One result is  $P \approx \tau.P$ , while  $P \not\approx \tau.P$ . On the other hand, no weak bisimulation equivalence is preserved by unguarded<sup>24</sup> SUMMATION:

$$\alpha + \beta \not\approx \tau.\alpha + \beta \tag{75}$$

This is due to the fact that the right agent, having done a  $\tau$ , is not able to simulate the left agent, which can do  $\alpha$  as well as  $\beta$ .

---

<sup>24</sup>Guarded=under a prefix

## Part II

# Logic for processes

## 5 Sequent Calculus

This is an introduction to the classical sequent calculus for predicate logic, originating from the 1930's [25],[26],[34]. The rules of this calculus are based on purely syntactical manipulation, but they are shaped in a way that semantical reasoning from them can be deduced.

**Definition 18** *In sequent calculus, we adopt the idea of a sequent, a structure denoted as*

$$\Gamma \rightarrow \Delta \tag{76}$$

where  $\Gamma$  and  $\Delta$  contains formulas, and the arrow is called a sequent arrow<sup>25</sup>.  $\Gamma$  is called the antecedent and  $\Delta$  the succedent of the sequent.

**Statement 1** *The goal for the sequent calculus is to make all formulas in the antecedent valid and all formulas in the succedent invalid.*

This is done by applying the rules from a *tableau system* of the calculus and assign names to all free variables of the formulas. If this succeeds, i.e. there exists a substitution of names such that all formulas of  $\Gamma$  are valid and all formulas in  $\Delta$  are invalid, the whole sequent is valid.

### 5.1 Proof System

**Definition 19** *A proof system for predicate logic sequent calculus [34] (without identity)<sup>26</sup>, and  $\mathcal{X}$  stands for a failed sequent:*

$$\frac{\Gamma \rightarrow \Delta}{\Gamma \rightarrow \perp, \Delta} \rightarrow \perp \tag{77}$$

$$\frac{\mathcal{X}}{\Gamma, \perp \rightarrow \Delta} \perp \rightarrow \tag{78}$$

$$\frac{\Gamma \rightarrow \Delta, \phi \quad \Gamma \rightarrow \Delta, \psi}{\Gamma \rightarrow \phi \wedge \psi, \Delta} \rightarrow \wedge \tag{79}$$

---

<sup>25</sup>When dealing with logical connectives, the logical *implication* is written as " $\supset$ " to avoid confusion.

<sup>26</sup>Identity would have to deal with formulas such as  $a = b$ ,  $a \neq b$  and calculate the truth values for such formulas. It is a minor extension of the here presented sequent calculus

$$\frac{\Gamma, \phi, \psi \rightarrow \Delta}{\Gamma, \phi \wedge \psi, \rightarrow \Delta} \wedge \rightarrow \quad (80)$$

$$\frac{\Gamma \rightarrow \Delta, \phi, \psi}{\Gamma \rightarrow \Delta, \phi \vee \psi} \rightarrow \vee \quad (81)$$

$$\frac{\phi, \Gamma \rightarrow \Delta \quad \psi, \Gamma \rightarrow \Delta}{\phi \vee \psi, \Gamma \rightarrow \Delta} \vee \rightarrow \quad (82)$$

$$\frac{\phi, \Gamma \rightarrow \Delta}{\Gamma \rightarrow \Delta, \neg \phi} \rightarrow \neg \quad (83)$$

$$\frac{\Gamma \rightarrow \Delta, \phi}{\neg \phi, \Gamma \rightarrow \Delta} \neg \rightarrow \quad (84)$$

$$\frac{\Gamma, \phi \rightarrow \Delta, \psi}{\Gamma \rightarrow \Delta, \phi \supset \psi} \rightarrow \supset \quad (85)$$

$$\frac{\Gamma \rightarrow \phi, \Delta \quad \psi, \Gamma \rightarrow \Delta}{\phi \supset \psi, \Gamma \rightarrow \Delta} \supset \rightarrow \quad (86)$$

$$\frac{\phi, \Gamma \rightarrow \Delta, \psi \quad \psi, \Gamma \rightarrow \Delta, \phi}{\Gamma \rightarrow \Delta, \psi \subset \supset \psi} \rightarrow \subset \supset \quad (87)$$

$$\frac{\phi, \psi, \Gamma \rightarrow \Delta \quad \Gamma \rightarrow \Delta, \phi, \psi}{\phi \subset \supset \psi, \Gamma \rightarrow \Delta} \subset \supset \rightarrow \quad (88)$$

$$\frac{\Gamma \rightarrow \Delta, \phi(y)}{\Gamma \rightarrow \Delta, \forall x \phi(x)} \rightarrow \forall \text{ where } y \text{ is a new variable} \quad (89)$$

$$\frac{\phi(t), \forall x \phi(x), \Gamma \rightarrow \Delta}{\forall x \phi(x), \Gamma \rightarrow \Delta} \forall \rightarrow \text{ where } t \text{ is any term} \quad (90)$$

$$\frac{\Gamma \rightarrow \Delta, \exists x \phi(x), \phi(t)}{\Gamma \rightarrow \Delta, \exists x \phi(x)} \rightarrow \exists \text{ where } t \text{ is any term} \quad (91)$$

$$\frac{\phi(y), \Gamma \rightarrow \Delta}{\exists x \phi(x), \Gamma \rightarrow \Delta} \exists \rightarrow \text{ where } y \text{ is a new variable} \quad (92)$$

Where  $\perp$  is the universally false statement,  $\supset$  is logical implication,  $\neg$  is logical negation and  $\subset \supset$  is logical equivalence,  $\phi$  and  $\psi$  are logical formulae,  $\Gamma$  and  $\Delta$  are sets of logical formulae, as described above.

## 5.2 Proofs in the sequent calculus.

The sequent calculus yields a *proof tree*, in which each node is a *sequent*. Some of the rules force the proof tree into *branching*, i.e. from that point and on, the tree is divided into several sub-trees. A node is marked  $\mathcal{X}$  means no proof is available at that particular node. The rules for the predicate logic connectives are motivated as: ( for  $\forall \rightarrow$  and  $\exists \rightarrow$ ): The formulas to the left of the sequent arrow are to be validated and the formulas to the right are to be falsified by the calculus. If we wish to prove the formula  $\forall x\phi(x)$  true, we will have to prove  $\phi(y)$  true for every possible element  $y$  of the structure we are looking for. The formula has to be true even higher up in the proof tree, therefore it is saved along with an instantiation of itself in the premise, of the form  $\phi(t)$ . The same reasoning is valid for  $\rightarrow \exists$ , to make the formula  $\exists x\phi(x)$  false, we will have to make all possible elements,  $\phi(t)$  false.<sup>27</sup> For the rules  $\rightarrow \forall$  and  $\exists \rightarrow$ , we simply state that if the formula  $\forall x\phi(x)$  is to be falsified, we find one formula,  $\phi(y)$  that is false in order for the whole formula to be false. The same reasoning can be done for the  $\rightarrow \exists$  rule; to make it false, it suffices to find a new element,  $\phi(y)$  that is false.

**Theorem 1** *For each one of the rules, the conclusion (the lower sequent) is valid iff all sequents above (the premises) are valid.[34]*

A disproof structure can be read at a leaf node of the proof tree when:

- 1) The structure only contains atomic formulas, that cannot be applied the rules of the calculus further.
- 2) The  $\forall$ -formulas in the antecedent have been applied the  $\forall \rightarrow$  rule on all activated terms,  $t$ .
- 3) The  $\exists$ -formulas in the succedent have been applied the  $\rightarrow \exists$  rule on all activated terms,  $t$ .

If all branches are closed, i.e. they are all closed by  $\mathcal{X}$ , a disproof structure could not be produced by the sequent calculus, and we are allowed to draw the conclusion that the original sequent in fact was valid.

---

<sup>27</sup>The repetition of a formula in the premise is called *contraction* and will be investigated further in the proof system of the model prover.

## 6 Modal Logic

Modal<sup>28</sup> logic [6] & [19] is an extension of regular predicate logic with *modalities*. In formal methods, this gives us the potential to reason about properties for *different program states*, not just properties for an entire program (process, agent). Modal properties can be described as different types of "local" properties.

### 6.1 Modal Operators

Unlike predicate logic, modal logic distinguishes between different types of true and false propositions. There are propositions that are necessarily true and others that are merely true. The same is valid for false statements.

**Definition 20** *Modal logic contains different modal notions. Its propositions can be:*

- 1) *necessarily true (always true, cannot be false)*
- 2) *Impossible (never true, no matter the circumstances)*
- 3) *Possibly true or false*

The key in this reasoning is *different worlds*. While predicate logic always refers to a single world when making a statement with its quantifiers ( $\exists$ ,  $\forall$ ), modal logic refers to several worlds at the same time. Modal logic extends predicate (or propositional) logic with two new operators, the necessity operator, and the possibility operator:

$$\Box p \tag{93}$$

$$\Diamond p \tag{94}$$

( $p$  is a proposition). Meaning: "p is true in all worlds"<sup>29</sup> and "p is true in some world"<sup>30</sup>, respectively. Just like the existential quantifier is defined in terms of the universal, the possibility operator is defined in terms of the necessity operator:

$$\Diamond p \equiv \neg \Box \neg p \tag{95}$$

$$\Box p \equiv \neg \Diamond \neg p \tag{96}$$

Modal operators are not true-functional in the ordinary logic way, i.e. there are no truth tables. Intuitively, since modal operators reason about different worlds, they can be defined as:

---

<sup>28</sup>It is called *modal* since in mediaeval logic, necessities etc were thought of as being the modes in which a proposition could be true or false.

<sup>29</sup>read: "box-p"

<sup>30</sup>read: "diamond-p"

**Definition 21** *Let  $W$  be the set of all possible worlds. Let  $\Phi$  denote a formula in ordinary predicate logic. Then,*

$$\Box\Phi \equiv \forall w \in W : \Phi \quad (97)$$

$$\Diamond\Phi \equiv \exists w \in W : \Phi \quad (98)$$

These are attractive features when describing computer programmes or processes; a "world" would in that sense mean a program *state*.

## 6.2 Reduction Laws for modal operators

$$\Box p \equiv \Diamond\Box p \quad (99)$$

$$\Diamond p \equiv \Box\Diamond p \quad (100)$$

$$\Box p \equiv \Box\Box p \quad (101)$$

$$\Diamond p \equiv \Diamond\Diamond p \quad (102)$$

## 6.3 Hennessy-Milner Logic

Hennessy and Milner [18] proposed a simple modal logic for CCS [8], for expressing properties of concurrent and non-deterministic programs. They first defined observational equivalence:

### 6.3.1 Definition of HML [17]

let  $\mathcal{A}$  be the set of all possible actions and  $\alpha$  an action such that  $\alpha \in \mathcal{A}$ . Let  $\Phi$  denote a formula in HML.

$$\Phi ::= \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid [\alpha]\Phi \mid \langle \alpha \rangle \Phi \mid T \mid F \quad (103)$$

That process  $P$  satisfies  $\Phi$  is denoted  $P \models \Phi$  and means semantically:

$$P \models T \quad (104)$$

$$P \not\models F \quad (105)$$

$$P \models \Phi_1 \wedge \Phi_2 \Leftrightarrow P \models \Phi_1 \text{ and } P \models \Phi_2 \quad (106)$$

$$P \models \Phi_1 \vee \Phi_2 \Leftrightarrow P \models \Phi_1 \text{ or } P \models \Phi_2 \quad (107)$$

$$P \models [\alpha]\Phi \Leftrightarrow \{\forall \alpha. P \xrightarrow{\alpha} P' \text{ and } P' \models \Phi\} \quad (108)$$

$$P \models \langle \alpha \rangle \Phi \Leftrightarrow \{\exists \alpha. P \xrightarrow{\alpha} P' \text{ and } P' \models \Phi\} \quad (109)$$

We can combine the modal operators. Then we are equipped with a powerful tool for expressing properties of processes.

**Example 9**

$$P \models \langle \alpha \rangle T \text{ (it is possible for process } P \text{ to perform } \alpha) \quad (110)$$

$$P \models [s]F \text{ (} P \text{ is deadlocked if } \alpha \text{ is performed)} \quad (111)$$

$\langle \alpha \rangle T$  expresses an *capability* for performing  $\alpha$ ,

$[\alpha]F$  expresses an *inability* for doing  $\alpha$ .

$[-]$  is a way of denoting any *observable* action in the action set, (anything but  $\tau$ ). If silent actions also are concerned, the modal operators can be written as

$$P \models [[\alpha]]\Phi \Leftrightarrow \{\forall \alpha. P \xrightarrow{\alpha} P' \text{ and } P' \models \Phi\} \quad (112)$$

$$P \models \langle\langle \alpha \rangle\rangle\Phi \Leftrightarrow \{\exists \alpha P \xrightarrow{\alpha} P' \text{ and } P' \models \Phi\} \quad (113)$$

where  $\xrightarrow{\alpha}$  means  $(\xrightarrow{\tau})^* \xrightarrow{\alpha} (\xrightarrow{\tau})^*$ , i.e. an action  $\alpha$  can be performed, disregarding all possible following or preceding  $\tau$ -actions.

## 6.4 Modal Equivalence

There is another way of describing equivalences between processes: We may say that two processes  $P$  and  $Q$  are similar if they have the same modal properties<sup>31</sup>. In fact, there is an intimate relationship between having the same modal properties and being bisimulation equivalent (strong or weak, respectively, according to in which of the modal logics from above we choose to express the properties).

**Definition 22** *The input modality for the diamond operator of the late and early strong bisimulation equivalence of the  $\pi$ -calculus are written as:*

$$P \models \langle\langle x(y) \rangle\rangle^{\mathcal{L}} \varphi \Leftrightarrow \exists P'. \forall z. \xrightarrow{x(y)} P' \text{ and } P'\{z/y\} \models \varphi \quad (114)$$

$$P \models \langle\langle x(y) \rangle\rangle^{\mathcal{E}} \varphi \Leftrightarrow \forall z. \exists P'. \xrightarrow{x(y)} P' \text{ and } P'\{z/y\} \models \varphi \quad (115)$$

where  $\varphi$  is a logic formula, written in HML.

## 6.5 Limitation

An disadvantage with modal logic for processes is its finite model property, i.e. we can only build a finite "chain" of modal operators in a formula, and thus express a finite behaviour model. If a process, on the other hand, is *infinite*, its properties may not be "caught" by modal logic.

---

<sup>31</sup>if they satisfy the same modal formulas

## 7 Temporal Logic

Although powerful, modal logic lacks the ability to express enduring capabilities. Only "immediate" properties are possible to formulate using the ordinary modal logic. Statements like "property  $P$  is *always* possible" or "action  $a$  will *eventually* happen" are out of its scope, due to the fact that modal logic formulas are always *finite* - and a process might not be. For this, we need a more powerful tool - temporal logic, that can be regarded as "modal logic with recursion". And with recursion follows the notion of fixed-points [17].

### 7.1 Liveness, Safety

Of special interest is the ability to express liveness and safety properties for a process:

**Definition 23** *Liveness: something good will eventually happen.*

**Definition 24** *Safety: nothing bad will ever happen.*

Temporal logic expresses properties of processes by expressing features of some or all of their runs (while modal logic only concerns processes properties as their behavior change through transitions). Also, modal logic naturally deals with finite processes, since its only ability in finite formulas stating properties of finite "chains" of actions.

### 7.2 Fixed points

We recapture the definition of *fixed-points* for functions from section 3.9.5. Every recursive function has a fixed-point. Suppose now that we wish to express an infinite property in modal logic for a process, e.g. "process  $P$  must *always* be able to perform the action  $\alpha$ ". Writing  $P \models \langle \alpha \rangle T$  or  $P \models \langle \alpha \rangle \langle \alpha \rangle T$  solves this for one or two states in  $P$ , but for infinity? The answer is a *recursive formula*. We could say that a process satisfying the formula  $X \triangleq \langle \alpha \rangle X$  means a process that can perform  $\langle \alpha \rangle$  and then satisfy  $X$  again, i.e. in a "loop". In this fashion, an infinite behaviour of a process can be captured. With recursive formulas come the notion of fixed points. i.e. a configuration, or *state*, as described by the formula, to which the process might return, analogously to the fixed points of an ordinary numeric function. These fixed points of formulas express interesting properties, as we shall see.

**Definition 25** *A complete partial order is any set of elements,  $C$  with a relation, usually written  $\sqsubseteq$  that is a partial order  $c \times c$  between the elements in  $C$ . Let  $c, x, y, a, b$  stand for elements, all in  $C$ . The relation  $\sqsubseteq$  is reflexive, anti-symmetric and transitive in the ordinary meaning:*

$$x \sqsubseteq x \tag{116}$$



$$x \sqsubseteq y, y \sqsubseteq x \rightarrow x = y \quad (117)$$

$$x \sqsubseteq y, y \sqsubseteq z \rightarrow x \sqsubseteq z \quad (118)$$

Let  $X$  be a subset of  $C$ . i.e.  $X \subseteq C$ . An element,  $y \in C$ , is called the least upper bound of  $X$  if,  $\forall x \in X : x \sqsubseteq y$ , and  $\neg \exists z \in X. y \sqsubseteq z$  or  $x \sqsubseteq z \sqsubseteq y$ . Note that  $y$  does not have to be a member of  $X$ , but of  $C$ . Analogously, if the same conditions apply, but  $y \sqsubseteq x$ , and  $\neg \exists z \in X. z \sqsubseteq y$  or  $y \sqsubseteq z \sqsubseteq x$ , we call  $y$  the greatest lowest bound of  $X$ . The least upper bound of  $X$  is also called the supremum of  $X$ , and is written  $\text{lub}(X)$  or  $\sqcup X$ . The greatest lower bound is also called the infimum of  $X$ , and can also be written  $\text{glb}(x)$  or  $\sqcap X$ .  $C$  is a complete partial order, CPO, if all elements  $c \in C$  are ordered by the relation  $\sqsubseteq$ . There is also a "bottom" element, usually written  $\perp$ , that has the property  $\forall c \in C : \perp \sqsubseteq c$ . Every chain of ordered elements of a CPO has a supremum (an empty chain would have  $\perp$  as supremum).

A function,  $f$  from elements in  $C$  to elements in  $C$ , i.e.  $f : C \mapsto C$ , is monotonic if  $\forall x, y \in C : x \sqsubseteq y \rightarrow f(x) \sqsubseteq f(y)$ . The element  $a \in C$  is a fixed-point to  $f$ , if  $f(a) = a$ . Also, if  $\forall b \in C$ , if  $b$  also is a fixed-point of  $f$ , and  $a \sqsubseteq b$ , and  $\neg \exists c \in C. c \sqsubseteq a$  and  $f(c) = c$ , then  $a$  is the least fixed-point of  $f$ , written  $\text{lfp}(f)$  or  $\mu f$ . Analogously, if  $b \sqsubseteq a$ , and  $\neg \exists c \in C. a \sqsubseteq c$  and  $f(c) = c$ , we say that  $a$  is the greatest fixed-point of  $f$ , written  $\text{gfp}(f)$  or  $\nu f$ . Kleene's recursion-theorem states that every monotonic function  $f$  on a CPO has a least fixed-point.

This notion of least/greatest fixed-points can be used for our recursion-defined modal logic formulas and the set of agents that satisfy these formulas.

**Definition 26** We can regard the set of agents,  $\mathcal{A}$  satisfying a formula  $\Phi$  as a CPO. The relation  $\sqsubseteq$  would be subset, i.e.  $\subseteq$ . Then the least fixed-point of a formula  $\Phi$  ( $\mu X.\Phi$ ) is the set of agents  $\bigcap \{A' \subseteq \mathcal{A} . \|\Phi\| \subseteq A'\}$ , where  $\|\Phi\|$  is the set of agents satisfying the formula  $\Phi$ . Analogously, the greatest fixed-point of a formula, written  $\nu X.\Phi$ , is the set  $\bigcup \{A' \subseteq \mathcal{A} . A' \subseteq \|\Phi\|\}$ .

The temporal operators can also be defined in terms of each other, a dual property similar to the relations between  $\exists, \forall$  or  $\diamond, \square$ :

$$\mu X.\Phi \triangleq \neg \nu X'. \neg \Phi \{ \neg X' / X \} \quad (119)$$

Adding these fixed-point operators to modal logic results in *the modal  $\mu$ -calculus*.

### 7.3 The modal $\mu$ -calculus

The modal  $\mu$ -calculus is an extension of HML with fixpoint notation and can be written as:

$$\Phi ::= \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid [\alpha]\Phi \mid \langle \alpha \rangle \Phi \mid \nu Z.\Phi \mid \mu Z.\Phi \mid T \mid F \quad (120)$$

## 7.4 Meaning

We can think of fixed-points as recursion points of a process, a state (or "configuration") to which the process may return. A maximum fixed-point formula expresses a configuration to which the process must return. Otherwise, the process does not satisfy the formula. A minimum fixed-point expresses a configuration, to which the process must NOT return, if it satisfies the formula. Invariant properties<sup>32</sup> can be expressed by using maximum fixed-point formulas. Intuitively, the maximum fixed-point formula  $\nu X.\Phi$  can be read as "always  $\Phi$ " and  $\mu X.\Phi$  "eventually  $\Phi$ ".

**Example 10** Let  $P \triangleq a.b.P$  be a recursive process, that can perform  $a$ , then  $b$  and then behave as  $P$  again, i.e. the language of  $P$  is  $a, ab, aba, abab, ababa, ababab, \dots$ . An invariant for this process is "it can always perform  $a$  or perform  $b$ ". This would be written

$$P \models \nu X.(\langle a \rangle X \vee \langle b \rangle X) \quad (121)$$

in the modal  $\mu$ -calculus.

## 7.5 Liveness and Safety properties

**Example 11** A maximal fixpoint formula expresses a safety property. If  $K$  contains all unsafe actions and  $k \in K$ , then if a system satisfies

$$\nu Z.(\langle k \rangle F \wedge [-]Z) \quad (122)$$

then the system is safe, "it is always the case that we cannot perform  $k$  and we can perform any (other) action and this formula holds".

**Example 12** If  $\Phi$  captures all "good" states, then the formula

$$\mu Z.(\Phi \vee (\langle - \rangle T \wedge [-]Z)) \quad (123)$$

means that the system is always kept at liveness<sup>33</sup> - "we will eventually reach a state where either  $\Phi$  holds or any action can be performed, either resulting in termination or that this formula holds again".

The formulas can express either terms of actions or terms of states.

---

<sup>32</sup>Invariant property = A property that holds throughout the entire execution of a process

<sup>33</sup>Can always change state

## 7.6 Tableau Proof System

The modal  $\mu$ -calculus<sup>34</sup> can be used in proof systems by using proof rules to reduce a formula to axiomatic level and then decide if it is valid. These rules are often presented as a "tableau". The tableaux can be read from top to bottom (premiss to conclusion) or bottom-up. Similar to the *sequent calculus*, we can develop proof trees, by applying the appropriate proof rule from the tableau on each new result from the former application of a rule. This application will stop when we have reached a leaf node, where an axiomatic formula  $T$  or  $F$  can be seen. An *successful* tableau has no  $F$ -leaves at all. In that case, we conclude the original process was indeed a model for the formula.

### 7.6.1 The tableau of rules

Let  $\sigma \in \{\mu, \nu\}$ . Let  $P$  be a process and  $\Phi$  a formula in the modal  $\mu$ -calculus. Let  $U$  range over propositional constants. Let  $\alpha$  be an action in the action set,  $\mathcal{A}$ . Then the tableau proof system for the modal  $\mu$ -calculus can be presented as:

$$\wedge \frac{P \vdash \Phi \wedge \Psi}{P \vdash \Phi \quad P \vdash \Psi} \quad (124)$$

$$\vee \frac{P \vdash \Phi \vee \Psi}{P \vdash \Phi} \quad \vee \frac{P \vdash \Phi \vee \Psi}{P \vdash \Psi} \quad (125)$$

$$[\alpha] \frac{P \vdash [k]\Phi}{P_1 \vdash \Phi \dots P_n \vdash \Phi} \{P' : P \xrightarrow{\alpha} P' \wedge \alpha \in \mathcal{A}\} = \{P_1 \dots P_n\} \quad (126)$$

$$\langle k \rangle \frac{P \vdash \langle \alpha \rangle \Phi}{P' \vdash \Phi} P \xrightarrow{\alpha} P' \wedge \alpha \in \mathcal{A} \quad (127)$$

$$\sigma Z \frac{P \vdash \sigma Z.\Phi}{P \vdash U} U \triangleq \sigma Z.\Phi \text{ and } U \text{ fresh} \quad (128)$$

$$U \frac{P \vdash U}{P \vdash \Phi\{U/Z\}} U \triangleq \sigma Z.\Phi \quad (129)$$

## 7.7 Fold, Unfold, Discharge

The track-keeping of fixpoints is a difficult task. Several methods have been suggested. Here we will present and use the suggestion of [16], a solution that uses *special constants* for keeping track of fixpoint formulas.

---

<sup>34</sup>Since we later will be using both minimum and maximum fixed-point formulas, even nested, the "modal  $\mu$ -calculus" and " $\nu$ -calculus" [15] are interchangeable terms here.

### 7.7.1 Unfold

" $\sigma Z.\Phi$ ", where  $\sigma \in \{\mu, \nu\}$

In the rules above, the notion of *special constants* ( $U$ ) is used as a method of keeping track of visited fixpoint formulas. Every time a fixpoint configuration is encountered, a new constant is substituted for it, and the proof rule application can continue. This is known as *Unfolding* the fixpoint formula.

### 7.7.2 Fold

" $U$ "

If we re-encounter a fixed point formula (i.e. a constant), but the process  $P$  is not equal to the process of the original Folding, this new configuration cannot be a re-visited fixpoint formula, despite the constant. In this case, we will have to substitute the original formula for the constant, and continue the proof rule applications. This is known as *Folding* a fixed point formula.

### 7.7.3 Discharge, fail

When a constant is found, we now that this is a previously visited point. We now have to decide whether it is a success or a failure. A maximum fixpoint must be re-visited, a minimum fixpoint must not. If the constant represented a maximum fixpoint, we can stop the application here, and consider the branch True. This is known as *Discharging* of a fixed point. Likewise, we can fail a minimum fixed point. Note that these rules are valid through the *induction hypothesis*; encountering a maximum fixed point formula once again, we assume it is indeed a maximum fixed point of the formula. Since the discharge procedure stops further proof search, we will not know if repeated search will return to the fixed point again, but that is validly assumed by the hypothesis.

## Part III

# Presentation

## 8 Model Checking

A model checker is a tool for checking whether a process, an *agent*,  $A$  satisfies a certain logic formula  $\Phi$ . The agent is expressed in a suitable process algebra therefore, e.g. the  $\pi$ -calculus. This is usually written as  $A \models \Phi$ .

### 8.1 The Problem

Model checking has always been associated with huge amounts of computation, and, when performed by computers, leading to complex programs demanding vast amount of memory space and other resources. For instance, when trying to prove that an agent satisfies a formula, the "brute force" way of doing so would be to let the agent run through all of its possible states and then check whether each of them satisfied the formula. This becomes non-efficient on larger agents with perhaps billions of states, and impossible on infinite-behavioural agents. Other approaches must then be used, that correctly reduces the number of states that has to be checked. One way is to implement a checker like in [11], based on the classic logic sequent calculus. An implementation was done in SICStus PROLOG v3. We will here present a total algorithm of this latter algorithm (hereafter called *Model Prover* or just *prover* to distinguish it from the earlier *checker*).

### 8.2 Checker vs. Prover: Differences.

The prover was influenced by earlier works in theorem proving, e.g. the intuitionistic predicate logic theorem prover of [9]. A main difference between the two model checking algorithms of [11] and [14] is the treatment of names; in checker, this is always done *explicit*, in prover we use a *symbolic* approach; Other entities ("parameters" and "variables") represent sets of names. The implementation of checker uses *binding modalities*<sup>35</sup>, i.e. a modal logic operator binds the name of the action in the adjacent formula. The prover uses non-binding modal formulas, as described in section 6 of this thesis. The treatment of fixpoint formulas are almost the same in both algorithms, the "special constant" approach, as described earlier. Another difference between the two algorithms is the prover's usage of *suspension* of certain proof goals as an efficiency improvement mechanism.

---

<sup>35</sup>This was a change between the presentation of [14] and the actual implementation.

### 8.3 A Sequent calculus model checker

In the Model Prover, we will write

$$\Gamma \rightarrow A : \Phi \tag{130}$$

for a sequent, where  $\Gamma$  is a set of *name-equations* (as explained later),  $A$  an agent (in the *finite control*<sup>36</sup>  $\pi$ -calculus), and  $\Phi$  a formula of the modal  $\mu$ -calculus. Intuitively, we can think of the Model prover as a way of trying to make the stated agent *not* satisfy the stated formula under the current nameequation and the current assignment of names to variables. Indeed, as soon as we have found a deduction of this being the fact, the algorithm terminates immediately. When looking for a proof of the sequents *validity*, we will often have to search longer ways to find this.

---

<sup>36</sup>see section 9.2.2 for details

## 9 Components

The components of the algorithm can be described as divided in two parts: atomic and molecular items.

### 9.1 Atoms

#### 9.1.1 Names & Parameters

A name is the most basic unit of the algorithm. They are always written in lower case:

$$x, y, z, out, in... \quad (131)$$

It is assumed invariantly that the supply of names to the algorithm is infinite, i.e. that new names can always be created.

A *parameter* is a meta-name, i.e. a name created by the algorithm at certain points where a "new" name, a name that is not already mentioned inside the structure is needed.<sup>37</sup> The difference between names and these (annotated) parameters is the fact that parameters carry indices, indicating at what stage of the algorithm they were created, a parameter with a lower index was created *before* a parameter with a higher index. Parameters will thus be denoted

$$p_1, p_2... \quad (132)$$

#### 9.1.2 Variables

A *variable* is a placeholder for a name. It is always unique throughout the entire structure and will be denoted by capital letters. Variables always carry indices as well as parameters,

$$V_1, X_2, Y_3 \quad (133)$$

and are always considered uninstantiated. When instantiated, they are substituted for a name.

#### 9.1.3 Actions

Actions are either unbarred, barred or the silent action  $\tau$  which is distinct from all other actions and is not barred or unbarred. The barred and unbarred actions can contain a name/parameter or a variable:

$$x, y, z_4, X_1, Y_2, Z_7 \text{ (unbarred)} \quad (134)$$

$$\bar{x}, \bar{y}, \bar{z}_4, \bar{X}_1, \bar{Y}_2, \bar{Z}_7 \text{ (barred)} \quad (135)$$

$$\tau \text{ (silent)} \quad (136)$$

---

<sup>37</sup>this is called "fresh" in [14] et al

#### 9.1.4 Propositional Variables

Propositional variables stand for a *point of recursion* in a formula and will be written

$$A, B, \dots \quad (137)$$

(Always *without* indexing.) They appear in *fixed-point* formulas and when those have been unfolded.

**Example 13** *The maximum fixed-point formula*

$$\Phi = \nu X (\langle \alpha \rangle X \vee \langle \bar{\beta} \rangle X) \quad (138)$$

*expresses the invariant property of a process that satisfies  $\Phi$  to always be able to perform  $\alpha$  or to perform  $\bar{\beta}$  and then continue from the point  $X$  in the formula, i.e. the same property again, in a loop.*

Intuitively, the prop.variables denote the point of recursion for recursive formulas.

#### 9.1.5 U - formula indexing

A formula constant  $U_i$  will stand for a subformula of a previously visited fixpoint formula in the algorithm. Whenever we encounter such a formula, it is replaced by a fresh constant  $U_i$  and the search continues. When we encounter a formula that *only* contains a constant, we know that this is a re-visited fixpoint formula, and we can then perform the appropriate actions according to the rules of the algorithm. This technique is adopted from [16] and ensures correct treatment also of even nested fixpoint formulas. A key condition is that every new constant  $U_i$  is unique throughout the structure.

**Example 14** *The Formula constant for the maximum fixpoint formula of the previous example would be*

$$U_0 = (\langle \alpha \rangle U_0 \vee \langle \bar{\beta} \rangle U_0) \quad (139)$$

#### 9.1.6 I - Denotation parameter index

The structure of the algorithm will at every point carry a *current index* of the latest created parameter. This will be written as

$$i \text{ where } i \in \mathcal{N} \quad (140)$$



## 9.2 Molecular Structures

### 9.2.1 Name-Equations

A Name-equation will be denoted  $\Gamma$  and is a set of equalities and *in*-equalities between names (and parameters). Note that neither variables nor actions cannot occur in name-equations. A name-equation is interpreted as a conjunction, i.e. all of its expressions are valid simultaneously:

$$\Gamma = \{a = b \wedge b \neq c \wedge b \neq d \wedge \dots \wedge \dots\} \quad (141)$$

### 9.2.2 finite-control Agents

Agents are expressed in the  $\pi$ -calculus of section 6, with one major difference: The agents have to be of *finite-control* type.

**Definition 27** *A finite-control agent has the property that no use of the parallel combinator " $|$ " is allowed in recursively defined agents. Thus, the agent  $A(x) \triangleq x.(A(x)|A(x))$  is NOT finite-control.*

This condition ensures that the agent will have a "finite" behaviour and that the algorithm therefore can terminate.<sup>38</sup>

### 9.2.3 Formulas

Formulas are of the modal  $\mu$ -calculus and atomic items are name-equalities and name-inequalities expressions.

### 9.2.4 Structure

A structure is a 8-tuple of the form

$$\langle \Gamma \rightarrow A : \Phi, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi \rangle \quad (142)$$

where

$\Gamma$  is a name-equation as described in 9.4.1,

$A$  is an agent as described in section 9.4.2,

$\Phi$  is a logic formula as described in section 9.4.3,

$\Theta$  is a *set* of alternatives as described in section 9.4.5,

$\Upsilon$  is a *set* of visited fixpoint formulas as described in section 9.4.7,

---

<sup>38</sup>It does the whole model checking decidable.

$i$  is the current parameterindex as described in section 9.3.6,  
 $\mathcal{S}$  is a *set* of suspended sequents as described in section 9.4.6,  
 $\Xi$  is a *set* of conjunctive structures as described in section 9.4.8.

The structure is the top-level of the algorithm. At each point of execution, we have a structure that can be altered according to the tableau rules into another structure until we reach a structure that allows us to terminate.

### 9.2.5 Alternatives

An *alternative* to the current sequent is a 5-tuple on one of two possible forms

$$\text{Bar}\langle A, \exists x\Phi, \Upsilon, i, \mathcal{S} \rangle \quad (143)$$

$$\text{Alt}\langle A, \Phi, \Upsilon, i, \mathcal{S} \rangle \quad (144)$$

where

$A$  is an agent as described in section 9.4.2,  
 $x$  is a name or parameter as described in section 9.3.1,  
 $\Phi$  is a logic formula as described in section 9.4.3,  
 $\Upsilon$  is a *set* of visited fixpoint formulas as described in section 9.4.7,  
 $i$  is the current parameter index as described in section 9.3.6,  
 $\mathcal{S}$  is a *set* of suspended sequents as described in section 9.4.6.

An alternative to the current sequent  $\Gamma \rightarrow A : \Phi$  is invoked when the current sequent fails to be validated by the rules. If the (possibly empty) list of alternatives of the structure contains another alternative, this one is invoked instead, and execution continues. This way, *at least* one of the alternative sequents from the alternative list has to yield a valid sequent for the algorithm to be able to validate the structure. Some of the rules of the algorithm add alternative sequents to the current structure. An "active" alternative is marked *Alt...* and can be used without restriction in the search of a proof. Another, "locked" alternative, marked *Bar...* also exists. The Barred alternatives come from contraction of  $\exists$ -formulas, and cannot be used in the proof search unless they are "unlocked" (i.e. transformed into an *Alt*-alternative), and this can only take place at certain points in the algorithm (mostly when new information has been added to the name-equations).

### 9.2.6 Suspended Sequents

A suspended sequent is a 6-tuple of the form

$$\langle \Gamma, A, \Phi, \Theta, \Upsilon, i \rangle \quad (145)$$

where

$\Gamma$  is a name-equation as described in 9.4.1,

$A$  is an agent as described in section 9.4.2,  
 $\Phi$  is a logic formula of one of the forms

$$V_i \neq W_j \tag{146}$$

$$V_i \neq x \tag{147}$$

$$[V_i]\Phi \tag{148}$$

(The symmetrical cases of (146) and (147) are omitted here.) where  
 $V_i, W_j$  are variables indexed with  $i$ , as described in section 9.3.2,  
 $x$  is a name or parameter as described in section 9.3.1,  
 $i, j$  are indexing parameters as described in section 9.3.6,  
 $\Phi$  is a logic formula as described in section 9.4.3,  
 $\Theta$  is an alternative as described in section 9.4.5,  
 $\Upsilon$  is a set of visited fixpoint formulas as described in section 9.4.7,  
 $i$  is the current parameter index as described in section 9.3.6.

A suspended sequent is one for which the validating efforts have ceased and it is put into a list of suspended sequents. This is because the formula has a variable in a vital place, as seen above, and the proof search therefore needs this variable to be unified with a name in order for the algorithm to be able to continue its search. If no unification is possible, the suspended sequent is considered proved valid. This is done by pure efficiency reasons. By assuming that there exists a name substitution so that all suspended sequents can be validated, we simply leave those unvalidated if no key variables of those sequents get instantiated. See section 12.12 for further reasoning about the suspension mechanism.

### 9.2.7 Visited Fixpoints

An *Visited fixpoint* is a 5-tuple on one of the forms

$$Max\langle U_j, \Gamma, A, \Phi, i \rangle \tag{149}$$

$$Min\langle U_j, \Gamma, A, \Phi, i \rangle \tag{150}$$

where

$U_j$  is a formula constant as described in section 9.3.5.  
 $\Gamma$  is a name-equation as described in 9.4.1,  
 $A$  is an agent as described in section 9.4.2,  
 $\Phi$  is a logic formula as described in section 9.4.3,  
 $i$  is the current parameter index as described in section 9.3.6,

The list of visited fixpoints is keeping track of all previous visited fixpoint configurations. This information is used whenever another fixpoint configuration is discovered in the algorithm. At that time, we can decide whether to discharge the current configuration (successful

validation of a maximal fixpoint), loopcheck (failing a minimum fixpoint) or to continue unfolding (the execution continues since the point was not re-visited after all).

### 9.2.8 Conjunctive Sequents

The conjunctive sequents  $\Xi$  of a structure is a set of zero or more structures as described in section 9.4.4. Some of the rules of the algorithm add sequents to the set of conjunctive sequents, and these have all to be proved in conjunction to the current sequent in order for the algorithm to validate the structure.

## 10 Model Prover BNF Description

**Definition 28**  $i, j \in \mathcal{N}$  are indices from an indexing set (the natural numbers),

**Definition 29**  $n$  is a name,  $\alpha$  an action,  $V$  is a variable,  $U_i$  a formula constant and  $X$  a prop.variable as defined in previous sections.

$$Structure ::= \langle \Gamma, A, \Phi, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi \rangle \mid \emptyset \quad (151)$$

$$\Gamma ::= n = n \wedge \Gamma \mid n \neq n \wedge \Gamma \mid \emptyset \quad (152)$$

$$Sequent ::= \Gamma \rightarrow A : \Phi \quad (153)$$

$$\begin{aligned} \Theta ::= & Alt\langle A, \Phi, \Upsilon, \iota, \mathcal{S} \rangle, \Theta \\ & | Bar\langle A, \exists n \Phi, \Upsilon, \iota, \mathcal{S} \rangle, \Theta \\ & | \emptyset \end{aligned} \quad (154)$$

$$\begin{aligned} \Upsilon ::= & Max\langle U_i, \Gamma, A, \Phi, \iota \rangle, \Upsilon \\ & | Min\langle U_i, \Gamma, A, \Phi, \iota \rangle, \Upsilon \\ & | \emptyset \end{aligned} \quad (155)$$

$$\begin{aligned} \mathcal{S} ::= & \langle \Gamma, A, V \neq n, \Theta, \Upsilon, \iota \rangle, \mathcal{S} \\ & | \langle \Gamma, A, V \neq X, \Theta, \Upsilon, \iota \rangle, \mathcal{S} \\ & | \langle \Gamma, A, [V]\Phi, \Theta, \Upsilon, \iota \rangle, \mathcal{S} \\ & | \emptyset \end{aligned} \quad (156)$$

$$\Xi ::= Structure, \Xi \mid \emptyset \quad (157)$$

$$A ::= (\lambda n)A \mid [n]A \mid cond(n = n), A, A \mid P \quad (158)$$

$$P ::= 0 \mid P+P \mid P|P \mid (\nu n)P \mid \alpha.A \quad (159)$$

$$\begin{aligned} \Phi ::= & n = n \mid n \neq n \mid True \mid False \mid \exists n \Phi \\ & | \forall n \Phi \mid \langle \alpha \rangle \Phi \mid [\alpha] \Phi \mid \nu X. \Phi \mid \mu X. \Phi \mid U_i \end{aligned} \quad (160)$$

$$\alpha ::= n \mid \bar{n} \mid \tau \quad (161)$$

$$n ::= x, y, z, \dots \mid x_i, y_j, \dots \tag{162}$$

## 11 The Tableau System of Rules

The algorithm can be presented as a set of sequent rules. Each rule will be of the form

$$RULENAME \frac{premise}{conclusion} \\ (explanation)$$

The given input to the algorithm, in form of a name-equation, an agent and a formula, is considered to be the *conclusion* of a calculus. The task for the prover algorithm is to find a valid premise on axiomatic level, that would yield the requested conclusion in zero or more steps. This is a *bottom-up* approach, used widely in model checking. Each of the sequent rules should thus be read from bottom and up, and a whole proof consists of a *tree*, where the conclusion (the input) is placed at the root and the axiomatic level(s) are at the leaf nodes. The proof tree will perhaps branch, since some of the rules produce several new sequents as output. Other rules create different new proof trees that will have to be proved along the current one. For the whole structure to be valid, at least one of the leaves in each created proof tree must contain a valid premiss at axiomatic/atomic level.

Also note that some of the rules are not explicit; i.e. they are "theoretical" in the sense that they are implemented quite differently than the stipulated rule expresses (this is true for *FTERM* and *NAMES*), and that others are "built-in" into the mechanism (this is true for *THINNING*, a rule that can be omitted due to *de Bruijn-indexing* (see section 15.1).

The algorithm applies the appropriate sequent rule to the starting structure and then applies another rule to the resulting structure of the previous application (if necessary). This way, the algorithm is purely deterministic and can be fully described by presenting the rules.

### 11.1 Set and list denotations

We will also use a denotation of lists similar to what is known in programming languages such as ML:

$$Head :: Tail$$

will denote a list consisting of *at least* the element *Head*. *Tail* may possibly be empty, or consist of other elements. The sets of the algorithm can be regarded as lists (in fact, that is how they are implemented!)

**Definition 30** *We will use the operator "•" for set addition (concatenation), i.e. for adding an element to a set (where the element is always assumed having the same type as the set/list).*

## 11.2 Operators, predicates and functions

**Definition 31** We will use the operator " := " when dealing with assignment, for example of a name to a variable.

**Definition 32** The operator " $\setminus$ " denotes set subtraction.

**Definition 33** The functions " $n(x)$ " and " $v(x)$ " return the set of all names (variables) of their argument, be it an agent, formula or a molecular structure.

**Definition 34** A substitution of every free occurrence of  $x$  in  $S$  for  $y$  is denoted  $S\{y/x\}$ . Bound occurrences can only be substituted when  $\alpha$ -conversion is made.

Commitments ( $\succ$ ) are explained in section 4.12.

Distinctions ( $\succ_{\{x,y\}}$ ) are explained in section 4.13.

### 11.2.1 Suspended Variable

The function *suspendedVariable*, written  $\mathcal{SV}$ , is defined as:

$\mathcal{SV}(\langle \Gamma, A, V_i \neq x, \Theta, \mathcal{X}, \iota \rangle) = V_i$  and

$\mathcal{SV}(\langle \Gamma, A, [V_i] \Phi, \Theta, \mathcal{X}, \iota \rangle) = V_i$ .

$\mathcal{SV}$  is undefined elsewhere.

### 11.2.2 Quietness, Implication

[9],[23] The relations quiet ( $\trianglelefteq$ ) and implies ( $\supset$ ) are defined as:

$$\begin{aligned}
 & \Gamma \text{ quiet } \Gamma' \ (\Gamma \trianglelefteq \Gamma') \text{ iff :} \\
 & \forall x, y \in \Gamma' : \text{ if } (x, y) \in \Gamma, \text{ then} \\
 & \text{ if } \Gamma' \models x = y \text{ then } \Gamma \models x = y \\
 & \text{ if } \Gamma' \models x \neq y \text{ then } \Gamma \models x \neq y
 \end{aligned} \tag{163}$$

$$\begin{aligned}
 & \Gamma \text{ implies } \Gamma' \ (\Gamma \supset \Gamma') \text{ iff : } \forall (x, y) \in \Gamma' : \\
 & \text{ if } \Gamma' \models x = y \text{ then } \Gamma \models x = y \\
 & \text{ if } \Gamma' \models x \neq y \text{ then } \Gamma \models x \neq y
 \end{aligned} \tag{164}$$

The relation *implies* is stronger than *quiet*, since it demands that for every occurrence of a name-equation in  $\Gamma'$ ,  $\Gamma$  MUST imply the same thing, whereas in the case of quietness,



the condition only stipulates that *if* the name-equation  $\Gamma$  contains the names, then it must imply the same thing as  $\Gamma'$ , so it is possible for  $\Gamma$  to contain *less* information than  $\Gamma'$ . This is sometimes referred to as  
 "  $\Gamma$  is a *conservative extension* of  $\Gamma'$ ."

### 11.2.3 Consistence

The predicate *consistent*( $x, y, \Gamma$ ), written  $\mathcal{C}(x, y, \Gamma)$  yields true/false whether adding the singleton name-equation  $\langle x = y \rangle$  to  $\Gamma$  would yield a *consistent* set  $\Gamma'$ , i.e. that  $\Gamma$  does not already contain the information  $x \neq y$ .

## 11.3 Axioms

The axioms are the points of the algorithm where termination can be done.

### 11.3.1 STERM

Successful Termination:

$$STERM \frac{valid}{\Gamma \rightarrow A : True, \Theta, \Upsilon, \iota, \mathcal{S}, \emptyset} \quad (165)$$

A structure is valid when the current sequent yields "true" and the conjunctive set of structures is empty.

### 11.3.2 FTERM

Failing termination:

$$FTERM \frac{fail}{\Gamma \rightarrow A : false, \emptyset, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (166)$$

A structure is invalid when the current sequent is proven invalid and the list of alternatives is empty.

## 11.4 Structural Rules

The structural rules are applied when:

- 1) The current sequent is invalid and there is an alternative in the alternative list ( $\Theta$ ) to be proved instead.
- 2) The current sequent is valid and there is a conjunctive structure in the  $\Xi$  list to be proved.

#### 11.4.1 T-LIT

Literal true-rule, when current sequent is valid and there is a conjunctive structure to be proven as well. This is essentially the same rule as *THINNING* of [11].

$$TLIT \frac{\Gamma' \rightarrow A' : \Phi', \Theta', \Upsilon', \iota', \mathcal{S}', \Xi'}{\Gamma \rightarrow A : True, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (167)$$

where  $\Xi = (\Gamma', A', \Phi', \Theta', \Upsilon', \iota', \mathcal{S}') :: \Xi'$

#### 11.4.2 F-LIT

Literal false-rule; when the current sequent is proven invalid and there is an alternative sequent to be proven.

$$FLIT \frac{\Gamma \rightarrow A' : \Phi', \Theta', \Upsilon', \iota', \mathcal{S}', \Xi}{\Gamma \rightarrow A : False, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (168)$$

where  $\Theta = (Alt\langle A', \Phi', \Upsilon', \iota', \mathcal{S}' \rangle) :: \Theta'$

### 11.5 Equivalence Rules

#### 11.5.1 EQ1

$$EQ1 \frac{\Gamma \rightarrow A : False, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi}{\Gamma \rightarrow A : x = y, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (169)$$

if  $\Gamma \supset x \neq y$

#### 11.5.2 EQ2

$$EQ2 \frac{\Gamma \rightarrow A : True, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi}{\Gamma \rightarrow A : x = y, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (170)$$

if  $\Gamma \supset x = y$  or  $x \equiv y$  (syntactically).

#### 11.5.3 EQ3

When an equality between a variable and a name (or parameter) is encountered. Note that we have omitted the symmetrical case, i.e. when the formula of the conclusion has the form  $V_i = x$ .

$$EQ3 \frac{\Gamma \rightarrow A\{x/V_i\} : True, \Theta\{x/V_i\}, \Upsilon\{x/V_i\}, \iota, \mathcal{S}'\{x/V_i\}, \Xi'\{x/V_i\}}{\Gamma \rightarrow A : x = V_i, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (171)$$

If  $i > \text{index}(x)$  then  $V_i := x$ .  
 $\text{Activateds} := \{s \in \mathcal{S} \mid \mathcal{SV}(s) = V_i\}$   
 $\mathcal{S}' := \mathcal{S} \setminus \text{Activateds}$   
 $\Xi' := \Xi \cup \text{Activateds}$

Where  $\text{index}(x) = 0$  if  $x$  is a name, and  $\text{index}(x_j) = j$  if  $x$  is a parameter.

(Note that the substitution and altering of everything in the premise *but* the conjunctive set  $\Xi$  is really unnecessary from a logical point of view due to the fact that the whole *current* sequent will be abandoned in the next step.)

#### 11.5.4 EQ4

$$EQ4 \frac{\Gamma' \rightarrow A, : \text{False}, \Theta', \Upsilon, \iota, \mathcal{S}, \Xi}{\Gamma \rightarrow A : x = y, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (172)$$

if  $\Gamma \not\vdash x = y$  then:

$\Gamma' := \Gamma \bullet \{x \neq y\}$

$\Theta' := \{\theta \in \Theta \mid \{\text{Alt}\langle A', \Phi', \Upsilon', \iota', \mathcal{S}' \rangle / \text{Bar}\langle A', \Phi', \Upsilon', \iota', \mathcal{S}' \rangle\}\}$

(All *Bar*-marked alternatives of  $\Theta$  changed into *Alt*-marked alternatives, i.e. the "locking" of an alternative that the bar-marking effectuates is removed.)

#### 11.5.5 INEQ1

$$INEQ1 \frac{\Gamma' \rightarrow A, : \text{False}, \Theta', \Upsilon, \iota, \mathcal{S}, \Xi}{\Gamma \rightarrow A : x = y, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (173)$$

if  $\Gamma \not\vdash x \neq y$  then:

$\Gamma' := \Gamma \bullet \{x = y\}$

$\Theta' := \{\theta \in \Theta \mid \{\text{Alt}\langle A', \Phi', \Upsilon', \iota', \mathcal{S}' \rangle / \text{Bar}\langle A', \Phi', \Upsilon', \iota', \mathcal{S}' \rangle\}\}$

(All *Bar*-marked alternatives of  $\Theta$  changed into *Alt*-marked alternatives, i.e. the "locking" of an alternative that the bar-marking effectuates is removed.)

#### 11.5.6 INEQ2

*Suspension* of inequality sequent where at least one of the inequals is a variable. Note that this rule is also applicable when the formula spells  $V_i \neq V_j$  and also for the symmetrical cases.

$$INEQ2 \frac{\Gamma \rightarrow A, : \text{True}, \Theta, \Upsilon, \iota, \mathcal{S}', \Xi}{\Gamma \rightarrow A : V \neq x, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (174)$$

where  $\mathcal{S}' := \mathcal{S} \bullet \langle \Gamma \rightarrow A : V \neq x, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi \rangle$

## 11.6 Logical Connectives

The rules for logical connectives add alternative sequents and conjunctive structures to the current structure.

### 11.6.1 $\vee$ (OR)-intro

$$\vee - INTRO \frac{\Gamma \rightarrow A : \Phi_1, \Theta', \Upsilon, \iota, \mathcal{S}, \Xi}{\Gamma \rightarrow A : \Phi_1 \vee \Phi_2, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (175)$$

where  $\Theta' := \Theta \bullet \langle A, \Phi_2, \Upsilon, \iota, \mathcal{S} \rangle$

### 11.6.2 $\wedge$ (AND)-intro

$$\vee - INTRO \frac{\Gamma \rightarrow A : \Phi_1, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi'}{\Gamma \rightarrow A : \Phi_1 \wedge \Phi_2, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (176)$$

where  $\Xi' := \Xi \bullet \langle \Gamma, A, \Phi_2, \Theta, \Upsilon, \iota, \mathcal{S} \rangle$

## 11.7 Predicate Logic Quantifiers

The introduction of quantifiers impose introduction of variables and parameters.

### 11.7.1 $\forall$ -intro

$$\forall - INTRO \frac{\Gamma \rightarrow A' : \Phi', \Theta, \Upsilon, j, \mathcal{S}, \Xi}{\Gamma \rightarrow A : \forall x \Phi, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (177)$$

where  $j := \iota + 1$

$p_j \notin n(A) \cup n(\Phi) \cup n(\Theta) \cup n(\Upsilon) \cup n(\mathcal{S})$

$A' := A(p_j)$

$\Phi' := \Phi\{p_j/x\}$

Note that the new parameter is unique throughout the *current* sequent.

### 11.7.2 $\exists$ -intro

$$\exists - INTRO \frac{\Gamma \rightarrow A' : \Phi', \Theta', \Upsilon, \iota, \mathcal{S}, \Xi}{\Gamma \rightarrow A : \exists x \Phi, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (178)$$

where  $V_i \notin v(A) \cup v(\Phi) \cup v(\Theta) \cup v(\Upsilon) \cup v(\mathcal{S}) \cup v(\Xi)$

$A' := A(V_i)$

$\Phi' := \Phi\{V_i/x\}$

$\Theta' := \Theta \bullet \text{Bar}\langle A, \exists x\Phi, \Upsilon, \iota, \mathcal{S} \rangle$

The new variable is unique throughout *the entire* structure.

## 11.8 Agent Rules

The *COND*-rules have to be tested first before any other test on the structure can take place. This is due to the fact that the rules may divide the structure into several others, each having the same formula but different agents, depending on whether the *condition*, that is prefixing the agent expression, evaluates true or not.

### 11.8.1 COND1

$$COND1 \frac{\Gamma \rightarrow A_1 : \Phi, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi}{\Gamma \rightarrow \text{cond}(x = y, A_1, A_2) : \Phi, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (179)$$

if  $\Gamma \supset x = y$ .

### 11.8.2 COND2

$$COND2 \frac{\Gamma \rightarrow A_2 : \Phi, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi}{\Gamma \rightarrow \text{cond}(x = y, A_1, A_2) : \Phi, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (180)$$

if  $\Gamma \supset x \neq y$ .

### 11.8.3 COND3

$$COND3 \frac{\Gamma' \rightarrow A_1 : \Phi, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi'}{\Gamma \rightarrow \text{cond}(x = y, A_1, A_2) : \Phi, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (181)$$

if  $\Gamma$  does not decide  $x = y$  nor  $x \neq y$ :

$\Gamma' := \Gamma \bullet \langle x = y \rangle$

$\Gamma'' := \Gamma \bullet \langle x \neq y \rangle$

$\Xi' := \Xi \bullet \langle \Gamma'', A_2, \Phi, \Theta, \Upsilon, \iota, \mathcal{S} \rangle$ .

### 11.8.4 $\Sigma$ -intro

$$\Sigma - INTRO \frac{\Gamma \rightarrow A : \Phi', \Theta, \Upsilon, \iota, \mathcal{S}, \Xi}{\Gamma \rightarrow [y]A : \Sigma x\Phi, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (182)$$

where  $\Phi' := \Phi\{y/x\}$ .

### 11.8.5 New-Rule

$$NEW - RULE \frac{\Gamma \rightarrow A' : \Phi', \Theta, \Upsilon, J, \mathcal{S}, \Xi}{\Gamma \rightarrow (\nu y)[y]A : \Sigma x \Phi, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (183)$$

where  $j := \iota + 1$   
 $p_j \notin n(A) \cup n(\Phi) \cup n(\Theta) \cup n(\Upsilon) \cup n(\mathcal{S})$   
 $A' := A\{p_j/y\}$   
 $\Phi' := \Phi\{p_j/x\}$

## 11.9 Modal Rules

### 11.9.1 DIAMOND1

$$DIAMOND1 \frac{\Gamma \rightarrow A : False, \Theta', \Upsilon, \iota, \mathcal{S}, \Xi}{\Gamma \rightarrow A : \langle \alpha \rangle \Phi, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (184)$$

where  $\alpha$  contains the name or parameter  $a$ :  
 $\Theta' := \Theta \cup \{Alt\langle B, \Phi, \Upsilon, \iota, \mathcal{S} \mid \Gamma \supset a = b \text{ and } A \succ b.B \rangle\}$ .

### 11.9.2 DIAMOND2

$$DIAMOND2 \frac{\Gamma \rightarrow A : False, \Theta', \Upsilon, \iota, \mathcal{S}, \Xi}{\Gamma \rightarrow A : \langle \alpha \rangle \Phi, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (185)$$

where  $\alpha$  contains the variable  $V_i$ :  
 $\Theta' := \Theta \cup \{Alt\langle B\{b/V_i\}, \Phi\{b/V_i\}, \Upsilon\{b/V_i\}, \iota, \mathcal{S}\{b/V_i\} \mid V_i := a, \Gamma \supset a = b \text{ and } A \succ b.B \rangle\}$ .

### 11.9.3 DIAMOND3

$$DIAMOND3 \frac{\Gamma \rightarrow A : False, \Theta', \Upsilon, \iota, \mathcal{S}, \Xi}{\Gamma \rightarrow A : \langle \tau \rangle \Phi, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (186)$$

where  $\Theta' := \Theta \cup \{Alt\langle B, \Phi, \Upsilon, \iota, \mathcal{S} \mid A \succ \tau.B \rangle\}$ .

#### 11.9.4 Box1

$$BOX1 \frac{\Gamma \rightarrow A : True, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi'}{\Gamma \rightarrow A : [\alpha]\Phi, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (187)$$

where  $\alpha$  contains the name or parameter  $a$ :  
 $\Xi' := \Xi \cup \{\langle \Gamma', B, \Phi, \Theta, \Upsilon, \iota, \mathcal{S} \rangle\}$   
 $A \succ b.B$  and  $\Gamma' := \Gamma \bullet \langle a = b \rangle$  and  $\mathcal{C}(a, b, (\Gamma))$

#### 11.9.5 BOX2

We suspend any box-formula containing a boxed variable:

$$BOX2 \frac{\Gamma \rightarrow A : True, \Theta, \Upsilon, \iota, \mathcal{S}', \Xi}{\Gamma \rightarrow A : [\alpha]\Phi, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (188)$$

where  $\alpha$  contains the variable  $V_i$ :  
 $\mathcal{S}' := \mathcal{S} \bullet \langle \Gamma, A, [\alpha]\Phi, \Theta, \Upsilon, \iota \rangle$

#### 11.9.6 BOX3

$$BOX3 \frac{\Gamma \rightarrow A : True, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi'}{\Gamma \rightarrow A : [\tau]\Phi, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (189)$$

where  $\Xi' := \Xi \cup \{\langle \Gamma', B, \Phi, \Theta, \Upsilon, \iota, \mathcal{S} \rangle\}$   
 $A \succ_{\{x,y\}} \tau.B$  and  $\Gamma' := \Gamma \bullet \langle x = y \rangle$  and  $\mathcal{C}(x, y, (\Gamma))$

### 11.10 Temporal Rules

The temporal rules concern the fixpoints. We use the letter  $\sigma$  as shorthand for one of the fixpoint operators. Thus,  $\sigma \in \{\nu, \mu\}$ .

#### 11.10.1 LOOPCHECK

$$LOOPCHECK \frac{\Gamma \rightarrow A : False, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi}{\Gamma \rightarrow A : U_i, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (190)$$

if  $\text{Min}\langle U_i, \Gamma', A', (\mu X)\Phi, \iota \rangle \in \Upsilon$  and  
 $A \equiv_{\alpha} A'$  and  $\Gamma \leq \Gamma'$ .

### 11.10.2 DISCHARGE

$$DISCHARGE \frac{\Gamma \rightarrow A : True, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi}{\Gamma \rightarrow A : U_i, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (191)$$

if  $\text{Max}\langle U_i, \Gamma', A', (\nu X)\Phi, \iota \rangle \in \Upsilon$  and  
 $A \equiv_\alpha A'$  and  $\Gamma \supset \Gamma'$ .

### 11.10.3 Fold

When the appearance of a fixpoint formula constant indicates that the configuration has been visited before, but other required conditions do not hold ( i.e. we have *not* returned to a previously visited fixpoint, so the folding has to continue):

$$FOLD \frac{\Gamma' \rightarrow A' : \Phi', \Theta, \Upsilon, \iota, \mathcal{S}, \Xi}{\Gamma' \rightarrow A' : U_i, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (192)$$

where  $\langle U_i, \Gamma, A, (\sigma X)\Phi, \iota \rangle \in \Upsilon$ , and  $\Phi' := \Phi\{U_i/X\}$  and  $A \not\equiv_\alpha A'$  or  $\Gamma \not\supset \Gamma'$ .

### 11.10.4 Unfold

When the fixpoint formula configuration has not been visited before:

$$UNFOLD \frac{\Gamma \rightarrow A : \Phi', \Theta, \Upsilon', \iota, \mathcal{S}, \Xi}{\Gamma \rightarrow A : (\sigma X)\Phi, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (193)$$

where  $U_i \notin \Upsilon$ , and  $\Phi' := \Phi\{U_i/X\}$  and  $\Upsilon' := \Upsilon \bullet \langle U_i, \Gamma, A, (\sigma X)\Phi, \iota \rangle$

## 11.11 Implicit Rules

The rule *FTERM* is an implicit rule, since the omission of it does not affect soundness of the system. The inclusion of *FTERM* is just for proper display of the point where the algorithm actually *fail* a sequent. Other rules, implicit to the algorithm are *BAR*, *EQUIVALENCE* and *NAMES*, where the latter is implemented as several other rules.

### 11.11.1 Bar

$$BAR - RULE \frac{\Gamma \rightarrow A : False, \Theta', \Upsilon, \iota, \mathcal{S}, \Xi}{\Gamma \rightarrow A : False, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (194)$$

where  $\Theta = \text{Bar}\langle A', \Phi', \Upsilon', \iota', \mathcal{S}' \rangle :: \Theta'$ ,

i.e. alternatives marked with *Bar* cannot be part of a proof, not until they have been "unbared" by the rules altering the name-equation  $\Gamma$ , i.e. *EQ4* or *INEQ1*. See the section 13.13 for argumentation on this.



### 11.11.2 Equivalence

$$EQUIVALENCE \frac{\Gamma \rightarrow A' : \Phi, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi}{\Gamma \rightarrow A : \Phi, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (195)$$

if  $A \equiv_{\alpha} A'$ . (A and A' are syntactically equivalent.) This rule is implicit in the implementation of the algorithm, since *de Bruijn-indexing* is used (as explained in section 15.1).

### 11.11.3 Names

$$NAMES \frac{\Gamma' \rightarrow A : \Phi, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi}{\Gamma \rightarrow A : \Phi, \Theta, \Upsilon, \iota, \mathcal{S}, \Xi} \quad (196)$$

where  $\Gamma' = \Gamma \cup \{z\}$  and  $\forall x \in n(\Gamma) : z \neq x$ , and  $z \neq n(A) \cup n(\Phi) \cup n(\Theta) \cup n(\mathcal{S}) \cup n(\Xi)$ . This means that we can always choose a new name from our infinite set, and put it in any name-equation, and it will be different from all previously existing names there. In other words, since we assume there exists an infinite space of names,  $\Gamma$  can always be extended with another (fresh) name that is *not* equal to all existing names in  $\Gamma$ , provided this new name does not already exist in some other place of the structure. Thus, in trying to prove the sequent  $\emptyset \rightarrow A : \exists u.(u \neq x \wedge u \neq y \wedge \Phi)$ , we can always introduce the fresh  $z$  and prove  $z \neq x \wedge z \neq y \rightarrow A : \exists z.\Phi\{z/u\}$ . This rule is interesting, since it functions as a "theoretical fundament" of the whole *suspension mechanism* (see below). The rules involved are *INEQ2*, *BOX2* (for suspension of sequents), and *EQ3* (for *re-activation* of suspended sequents).

## 11.12 Suspension

Suspension is an *optimization technique*. Instead of trying to prove certain sequents, if their formulas satisfy certain patterns, we simply put the sequents away, consider them as proved if nothing else happens. This is due to the fact of:

- 1) The existence of *variables* in the formula and
- 2) The assumption of an infinite name-space

These two aspects make it possible to make the suspension of a sequent where the formula is either

$$V_i \neq x \quad (197)$$

$$V_i \neq W_j \quad (198)$$

$$[V_i]\Phi \quad (199)$$

where  $i = j$  or  $i \neq j$ . Assumption 2) above makes it possible to state that:

**Theorem 2** *Due to the fact that the name-space is infinite, there will always exist an assignment of the variable  $V_i$  such that any of the formulas above can be transformed into validity.[11]*

This becomes clear when we reason intuitively; We can always choose a  $V_i$  from our name-space so that the inequality holds. We can always choose any name for the BOX-formula, and it will hold.

### 11.13 Contraction

Contraction is the repetition of a conclusion in the premise. This must take place at certain points in the algorithm, where information on the sequent otherwise would be lost. Especially important is this in the case of the  $\exists - INTRO$  as seen in section 13.13. Here, the current alternative  $\Gamma \rightarrow A : \exists x\Phi$  is repeated in the premise in the shape of an Bar-marked alternative in the alternative set  $\Theta$ . The alternative can thus be used again in a proof, but not before another rule has removed the Bar-status and turned it into a "Alt"-mark instead. This is done *whenever there is new information at hand, that could help in proving the  $\exists$ -formula*, i.e. whenever the name-equation set  $\Gamma$  is altered (extended). This technique guarantees that *infinite looping* is avoided, i.e. that the algorithm will terminate. Since the  $\exists$ -formula always is repeated in the premise above, without the Bar-marking, this repetition could go on forever, effectively hindering the algorithm from termination:

$$\begin{array}{c} \dots \\ \dots\exists x\Phi\dots \\ \underline{\dots\exists x\Phi\dots} \\ \dots\exists x\Phi\dots \end{array}$$

## 12 Semantics

In order to reason about soundness, completeness and behaviour of the tableau rules, we need to know their exact meaning. This is usually presented as a set of *semantic*[33] definitions, mostly in set algebra. This is a very precise way of describing the exact meaning of different operators and transition rules.

### 12.1 Definitions

**Definition 35** Let  $\rho$  be a mapping,  $Names \mapsto Names$  (formal names to actual names).

**Definition 36** Let  $\|\Gamma \rightarrow A : \Phi\|_\rho$  denote the set of agents, satisfying  $\Phi$  under the name-equation constrains of  $\Gamma$  and with respect to  $\rho$

**Definition 37** let  $\mathcal{A}$  denote the universe of all possible agents.

**Definition 38**  $\succ_\Gamma$  means "committing under  $\Gamma$ ", i.e. if  $A \succ \alpha.B$ ,  $n(\alpha) = a$ ,  $\Gamma \supset a = b$ , and  $n(\beta) = b$  then  $A \succ_\Gamma \beta.B$ .

### 12.2 Tableau Rules Semantic definitions

$$\|\Gamma \rightarrow A : True\|_\rho \triangleq \mathcal{A} \quad (200)$$

$$\|\Gamma \rightarrow A : False\|_\rho \triangleq \emptyset \quad (201)$$

$$\|\Gamma \rightarrow A : x=y\|_\rho \triangleq \begin{cases} \|\Gamma \rightarrow A : False\|_\rho & \text{if } \Gamma \supset x \neq y \\ \|\Gamma \rightarrow A : True\|_\rho & \text{if } \Gamma \supset x=y \\ \|\Gamma \cup \{x \neq y\} \rightarrow A : True\|_\rho & \text{if } (x, y) \notin \Gamma \end{cases} \quad (202)$$

$$\|\Gamma \rightarrow A : x \neq y\|_\rho \triangleq \begin{cases} \|\Gamma \rightarrow A : False\|_\rho & \text{if } \Gamma \supset x=y \\ \|\Gamma \rightarrow A : True\|_\rho & \text{if } \Gamma \supset x \neq y \\ \|\Gamma \cup \{x=y\} \rightarrow A : True\|_\rho & \text{if } (x, y) \notin \Gamma \end{cases} \quad (203)$$

$$\|\Gamma \rightarrow A : \Phi_1 \vee \Phi_2\|_\rho \triangleq \|\Gamma \rightarrow A : \Phi_1\|_\rho \cap \|\Gamma \rightarrow A : \Phi_2\|_\rho \quad (204)$$

$$\|\Gamma \rightarrow A : \Phi_1 \wedge \Phi_2\|_\rho \triangleq \|\Gamma \rightarrow A : \Phi_1\|_\rho \cup \|\Gamma \rightarrow A : \Phi_2\|_\rho \quad (205)$$

$$\|\Gamma \rightarrow A : \exists x \Phi\|_\rho \triangleq \|\Gamma \rightarrow A(V) : \Phi\{V/x\}\|_\rho \cap \|\Gamma \rightarrow A : \exists x \Phi\|_\rho,$$

$$V \notin v(\Gamma) \cup v(A) \cup v(\Phi) \quad (206)$$

$$\|\Gamma \rightarrow A : \forall x \Phi\|_\rho \triangleq \|\Gamma \rightarrow A(y) : \Phi\{y/x\}\|_\rho, \quad y \notin n(\Gamma) \cup n(A) \cup n(\Phi) \quad (207)$$

$$\|\Gamma \rightarrow [y]A : \Sigma x \Phi\|_\rho \triangleq \|\Gamma \rightarrow A\{y/x\} : \Phi\{y/x\}\|_\rho \quad (208)$$

$$\|\Gamma \rightarrow (\nu y)[y]A : \Sigma x \Phi\|_\rho \triangleq \|\Gamma \rightarrow A\{z/y\} : \Phi\{z/x\}\|_\rho, \quad z \notin n(\Gamma) \cup n(A) \cup n(\Phi) \quad (209)$$

$$\|\Gamma \rightarrow A : \langle \alpha \rangle \Phi\|_\rho \triangleq \{\Gamma \rightarrow \{P \mid \exists \beta, B : A \succ_\Gamma \beta.B, \Gamma \supset (\alpha = \beta), B \in \|\Gamma \rightarrow A : \Phi\|_\rho\} : \Phi\} \quad (210)$$

$$\|\Gamma \rightarrow A : [\alpha] \Phi\|_\rho \triangleq \{\Gamma \rightarrow \{P \mid \forall \beta, B : A \succ_\Gamma \beta.B, \Gamma \supset (\alpha = \beta), B \in \|\Gamma \rightarrow A : \Phi\|_\rho\} : \Phi\} \quad (211)$$

$$\|X\|_\rho \triangleq \rho(X) \quad (212)$$

$$\|\Gamma \rightarrow A : \mu X. \Phi\|_\rho \triangleq \{\Gamma \rightarrow \bigcap \{A' \subseteq \mathcal{A} \mid \|\Gamma \rightarrow A : \Phi\|_\rho \subseteq A'\} : \Phi\} \quad (213)$$

$$\|\Gamma \rightarrow A : \nu X. \Phi\|_\rho \triangleq \{\Gamma \rightarrow \bigcup \{A' \subseteq \mathcal{A} \mid A' \subseteq \|\Gamma \rightarrow A : \Phi\|_\rho\} : \Phi\} \quad (214)$$

$$\|\Gamma \rightarrow \text{cond}(x=y), A_1, A_2 : \Phi\|_\rho \triangleq \begin{cases} \|\Gamma \rightarrow A_1 : \Phi\|_\rho & \text{if } \Gamma \supset x=y \\ \|\Gamma \rightarrow A_2 : \Phi\|_\rho & \text{if } \Gamma \supset x \neq y \\ \|\Gamma \cup \{x=y\} \rightarrow A_1 : \Phi\|_\rho \cup \|\Gamma \cup \{x \neq y\} \rightarrow A_2 : \Phi\|_\rho & \text{if } (x, y) \notin \Gamma \end{cases} \quad (215)$$

## 13 Soundness

We have presented the algorithm, its rules and properties, and its semantics. The time has come to reason about whether the algorithm is sound or not. Soundness is a very important feature. A sound calculus means that every answer is correct.<sup>39</sup> First let us stipulate:

**Theorem 3** *The rules of the algorithm are sound. Every tableau rule turns a sound structure into another sound structure .*

And since the tableau is a *complete* description of the algorithm, it suffices to prove all proof rules sound to claim:

**Theorem 4** *If all tableau proof rules are sound, then the whole algorithm is sound.*

A key feature here is locality: *Everything we need to know at every point of time during the performance of the algorithm is stored inside the current structure.* Consider the fact that the sets  $\Theta$ ,  $\mathcal{S}$ ,  $\Upsilon$  and  $\Xi$  at every point contain the alternatives, suspended sequents, visited fixpoint configurations and conjunctive structures to be visited by the algorithm, depending on what happens at the current sequent of the structure. Whenever these sets are modified, the premise of the rule will keep the altered set along, this ensuring the theorem above.

We will thus have to argue that every rule of the tableau proof system is sound, bringing back the features of the original sequent calculus, which purpose was to make all formulas to the left valid and the formulas to the right invalid:

### 13.1 The axioms & structural rules

$\Gamma \rightarrow A : True$

It is clear that having the constant *True* as a succedent will yield no proof of the algorithm's *invalidity*, thus forcing us to search for another disproof in one of the conjunctive structures (if they exist). Else, we have a proof of the sequent.

$\Gamma \rightarrow A : False$

We are at the goal, the sequent is proved invalid. Since the set of alternatives *all* have to be proved invalid (i.e. the set must be interpreted as a *disjunctive* set of alternatives), we here must test another alternative (if any). Else, we have a disproof of the sequent and can terminate.

---

<sup>39</sup>Completeness, on the other hand, means that "every correct answer can be output by this calculus". Completeness is often a much heavier task to prove than soundness, and the topic will not be further investigated in this thesis.

## 13.2 Equivalence rules

$\Gamma \rightarrow A : x=y,$

where  $x$  and/or  $y$  can be names, parameters or variables. For the case of non-variables, consider that this is the only place in the calculus where a formula can be moved from the right side of the sequent to the left side. As in classical sequent calculus, this can be performed by *negating* the transferred formula, thus yielding the extended name-equation set for  $\Gamma$ , containing this new information, in order for the algorithm to progress. However, if the name-equation set is to be inconsistent adding this new tuple, we cannot progress the algorithm any further from here, and thus have to fail the current sequent. When a variable is involved, the equality is merely turned into an assignment statement, propagating throughout the entire structure, and re-awakening possible suspended sequents.

$\Gamma \rightarrow A : x \neq y$

In the case of *inequalities*, the same reasoning as previous is valid, except for the case of a variable being unequal to another variable or name/parameter; this is universally true (as seen in section 13.12), due to the infinite space of names, and we need (nor can) not progress this sequent any further, therefore it is suspended.

## 13.3 Logical rules

$\Gamma \rightarrow A : \Phi_1 \oplus \Phi_2$ , where  $\oplus \in \{\vee, \wedge\}$

It is clearly sound and according to the semantic definitions of the logical connectives to generate an alternative to the current sequent (to be put in the "disjunctive" set of alternatives,  $\Theta$ ) as a premise for the "OR"-introduction, as well creating a conjunctive structure to be put in the set  $\Xi$  for the case of "AND"-introduction in the premise.

## 13.4 Quantifiers

$\Gamma \rightarrow A : \flat x \Phi$ , where  $\flat \in \{\exists, \forall\}$

Whenever an  $\exists$ -formula is encountered at a right side of a sequent, to disprove it, we need to disprove the formula for *all* names. Thus the contraction, as described in 13.13, for ensuring that the formula will be applied to all possible future names. We also need to instantiate the formula at present level, for the currently activated names, therefore we introduce a *unique variable*, denoting the set of all activated names at present point in the calculus. If this variable in fact *can* be assigned a name, the  $\exists$ -formula is valid for that name, and the succedent containing this formula turns *invalid*, due to the fact that in the right hand side of a sequent, for an  $\exists$ -formula to be invalid, no name must exist that makes  $\Phi$  valid. Failing this unification of a name and the variable makes it possible for the algorithm to progress. The opposite reasoning is performed for the  $\forall$ -formulas: if  $\forall x \Phi$  to be invalid, there must exist an annotated name (a parameter)  $p_i$  that has the property  $\Phi$  and that is not "activated" in the current structure. Thus  $\Phi\{p_i/x\}$  must be in the premise.

The parameter index, denoted  $i$  in the proof rules, play a rôle for soundness: A variable cannot be unified with a parameter with an index less than its own; to see why this is sound, consider

**Example 15**  $\exists x\forall y.\Phi(x, y) \supset \forall y\exists x.\Phi(x, y)$  but  $\forall x\exists y.\Phi(x, y) \not\supset \exists y\forall x.\Phi(x, y)$ .

The correct introduction of variables unifying with parameters can be assured by the indexing technique; a variable (an  $\exists$ -formula name) has to be created *later* than a parameter (a  $\forall$ -formula name), i e, *inside the scope of the parameter* in order for the variable to be unified with the parameter.

### 13.5 Agent rules

$\Gamma \rightarrow \text{cond}(x=y), A_1, A_2 : \Phi$

The conditional agent  $\text{cond}(x=y)A_1, A_2$  is simply evolved due to the result of the condition, ( $x=y$ ). If this cannot be decided, both sequents containing  $A_1$  and  $A_2$  will have to be tested by the algorithm.

### 13.6 Summation & New rules

$\Gamma \rightarrow [y]A : \Sigma x\Phi$

$\Gamma \rightarrow (\nu y)[y]A : \Sigma x\Phi$

The difference between those is the fact that the agent in the *NEW*-case has not only the abstraction of  $y$ , but it is also a private (bound) variable in  $A$ . Therefore, in order for progressing, a entire new parameter has to be created, that does not exist in current structure, and substituted into the sequent.

### 13.7 Modal rules

$\Gamma \rightarrow A : \langle \alpha \rangle \Phi$

The alternative set  $\Theta$  has to be extended with every sequent that, according to the definition of the *commitment* relation, fulfills the conditions. We also have to calculate the conditions with respect to that the name of the action  $\alpha$  can be equal to other names, according to  $\Gamma$ . The commitments for this equivalence class of names must also be calculated for. Putting the resulting sequents in the alternative set ensures that at least one of them will have to yield a valid structure, in strong cohesion to the definition of modalities (as seen in section 12.2). If  $\alpha$  does not consist of a name but a variable, we will at first try to unify this with a name, calculate the commitments for this name and  $A$ , and create an alternative with the unification propagated throughout the alternative. In case of the action being the silent action  $\tau$ , any communication inside a parallel agent that can result in a  $\tau$ -action commitment will have to be calculated and put into the alternatives.

$\Gamma \rightarrow A : [\alpha]\Phi$

The same calculation of commitments is done here as for the diamond-rules. However, since the box rules demand that the property holds *for every occurring* commitment, we create (zero or more) conjunctive sequents, that all have to be proved along with each other. This way, a correct implementation of the box-operator is guaranteed. In the case of  $\alpha$  containing not a name/parameter but a variable, we consider the formula as being true according to the infinite set of names, and suspend the sequent. This is different from the diamond-operator, where we **MUST** find a way to make the algorithm progress, and then we are forced to try out different unifications in order to try to find a sequent alternative that can be dissolved further.

### 13.8 Fixpoint rules

$\Gamma \rightarrow A : \sigma X.\Phi$

Where  $\sigma \in \{\nu, \mu\}$ . The fixpoint rules are often considered the ones most difficult to implement and prove correct. For *DISCHARGE*, we simply state that: Having visited a fixpoint configuration of the form  $\Gamma \rightarrow A : \nu X.\Phi$  and turned this into  $\Gamma \rightarrow A : U_i$  for a fresh  $U_i$ , we know when the configuration  $\Gamma' \rightarrow A' : U_i$  comes up, that this is a previously visited point of a recursive formula. However, in order to discharge the new configuration, more conditions are needed:  $A \equiv_\alpha A'$  and  $\Gamma' \supset \Gamma$ , i.e. the new configuration has to be *at least* as strong as the old one, in order for it to be a true revisited fixpoint (and to be discharged as a success). If not so, (i.e. if  $\Gamma' \not\supset \Gamma$ ), there exists information in the old name equation that is not included in the new one, and the new configuration is more general than the old one, and thus is the maximum fixpoint not re-visited, and the unfolding can then continue without discharging.

For minimum fixpoint configurations, the same arguments can be applied as above, note however that we fail a minimum fixpoint configuration if it is revisited instead of terminating it successfully. A minimum fixpoint configuration must thus only be visited once and then nevermore. (Also see the argumentation for *LOOPCHECK* in section 14 concerning termination).



## 14 Termination

In order for the algorithm to terminate on every possible input, there has to be a number of features that have to be addressed:

1. *Every agent must have a finite behaviour.*

Every agent, being of type finite-control (as described in section 9.4.2), does indeed have this property. A recursive agent is guaranteed to come back to a previous state, since the avoidance of free names in recursive agents is at hand. This guarantees a finite-expressible appearance, and thus an agent expression cannot evolve forever.

2. *Every formula must eventually be reduced to an atomic formula.*

This is trivially true for every formula except fixpoint formulas. For those, we substitute the propositional variable of the formula into a representing unique constant,  $U_i$ , which trivially is bound to be unfolded. Thus, a fixed-point formula must eventually be returned to, and then the decision of discharge/loopcheck can take place.

3. *The algorithm must detect any kind of looping and act properly upon it.*

Looping can take place in two cases:

- 1) The contraction of an  $\exists$ -formula
- 2) In the search of a revisitation of a fixpoint formula.

In case 1) we have described the Bar-mechanism that ensures that a contracted  $\exists$ -formula cannot be repeatedly used in the search of a proof; it is only used again in the case of additional information being added to the name-equations and thus new possibilities of unifying the variable that is created at every use of  $\exists$ -formulae. Else, a Bar-formula just fails the current sequent. In case 2) We note that for maximum fixpoint, a loop can only occur in the case of the re-visited configuration being more and more general each time it is revisited. Otherwise, we will have a maximum fixpoint-hit and the algorithm succeeds the sequent. In the case of *minimum fixpoints*, this is not enough.

**Example 16** *Consider the case when we start with the sequent*

$\{x \neq y \wedge x \neq u\} \rightarrow A : U_i$ . (And  $U_i$  stands for a  $\text{Min}\langle \dots \rangle$ -point.) *We later arrive in the proof tree at the point*  $\{x \neq y \wedge x \neq u \wedge x \neq z \wedge y = z\} \rightarrow A' : U_i$ .

*And  $z \notin n(A) \cup n(U_i)$ .*

In this case, the later  $\Gamma'$  is merely *virtually* stronger than  $\Gamma$  and since  $z$  is not a name that is included in the sequent, we will not yield a proof by applying a virtually stronger name-equation. Therefore, we use the *LOOPCHECK* test rather than just plain implication in the minimum-fixpoint case; a loopcheck discovers that the later name-equation is not stronger than their previous one, with respect to the names that can be of any interest to the sequent (and  $z$  cannot). Therefore, we do have a minimum fixpoint hit and can thus terminate the sequent failingly here. The loopcheck predicate guarantees that no "false" stronger implications force the minimum fixpoint search to a looping state.

4. *Every tableau rule that creates new sequents, be it alternatives, suspended sequents or conjunctives, must only create an infinite number of those.*

This applies foremost to the modal operators; Since they create only as many new sequents that the commitment relation stipulates, and the commitment relation calculates those from the agent expression, and the agent expression is of *finite control*-type, only a finite number of new sequents can be created at every usage of a modal operator.

## 15 Implementation

A suggestion for an implementation, made in Standard ML [24], [31] is available in the Mobility Workbench from FTP-server `ftp.docs.uu.se`, directory `pub/mwb`.

See also the MWB web page at <http://www.docs.uu.se/~victor/mwb.html> Files concerning the Model Prover are in the `/mc/`-directory: `SubSequents.sml`, `Nameequation.sml`, `Prover.sml` and `PFormula.sml` and, of course, their respective `.sig` files, in the source code distribution. The implementation is done in Standard ML of New Jersey, version 0.93 and compiled under Sun OS version 4.

### 15.1 de Bruijn-Indexing

[10]

A special feature of the implementation of the Mobility Workbench and thus the Model Prover is *de Bruijn-indexing*. The idea is that of a *universal way of denoting bound names in expressions*, as described in section 3.13. A bound name is then replaced by an integer number, indicating the depth of the nestling at which the bound variable occurs. (The free names have to be treated in a rather different way; they are replaced by an index number, serving as a pointer to an entry of a "universal" table of free names. Every  $\pi$ -calculus agent or  $\mu$ -calculus formula expression can be expressed as a de Bruijn-indexed expression. Two  $\alpha$ -convertible agents (formulas) look syntactically the same when indexed by de Bruijn-indexing. The usage of integers instead of names (that have to be represented as strings somewhere in the system) clearly has another advantage; the comparison between names is now a simple integer test. Implementations have shown that de Bruijn-indexing indeed speeds program execution up enormously.

### 15.2 Back-Tracking

[21],[22]

The original prover was implemented in PROLOG, so the back-tracking technique had to be solved another way. The construction of the main loop as a series of IF-statements, each one either terminating or recursing, was a way to solve this; a context of variable bindings was created for every new recursive call, and the whole mechanism would roll back if no proof was at hand, re-installing an older context. A pseudo-code presentation of the Prover algorithm follows the form

```
prove(agent, formula)=
id (axiomatic truth) then success
if <rule1, formula> then
{ new_agent, new_formula := apply_rule<1, agent, formula> ;
  prove(new_agent, new_formula);
}
```

```
if <rule2, formula ..... >  
else fail
```

where the if-statement, if the recursive call fails, rolls back all contexts and assignments and the algorithm continues another way.

## 16 The Mobility Workbench with Model Prover

The Mobility Workbench (MWB) is a tool for manipulating and analyzing mobile concurrent systems described in the  $\pi$ -calculus. It features commands for searching for deadlocks, stepwise simulation, determining strong and weak bisimulation equivalences. It also features both the old "checker" and the Model Prover, as described in this thesis.

### 16.1 Where?

The Mobility Workbench [12] is available as a gzipped tar-archive via anonymous FTP from `ftp.docs.uu.se`, directory `pub/mwb/`. Also see the web page with pointers to a on-line manual at: <http://www.docs.uu.se/~victor/mwb.html>

### 16.2 How?

The MWB is started by the command

```
mwb.sun4u
```

at the UNIX-prompt. The prover is invoked by the statement

```
prove agent formula
```

where *agent*, *formula* are valid expressions therefore as defined by:

#### 16.2.1 MWB Prover Syntax

For *agents*: The syntax of agents is given by the following grammar:

<u>Syntax</u>	<u>Meaning</u>
$0$	The null process
$\alpha.P$	Action prefix
$\text{pfx}.P$	prefix
$[a = b]P$	Match
$P_1 P_2$	Parallel
$P_1 + P_2$	Summation
$\text{Id}\langle \text{nlist} \rangle$	Application
$(\hat{\text{nlist}})P$	Restriction
$(\backslash \text{nlist})P$	Abstraction
$[\text{nlist}]P$	Concretion
$(P)$	Parenthesis

(216)

where *nlist* is a (non-empty) comma-separated list of names;  
 $\alpha$  is an action:  $\tau$  (silent) or a name (input) or a co-name (output);

**plx** is an abbreviated prefix (see below); and **Id** is a name starting with an *upper-case* letter. Names must start with a *lowercase* letter but can after that include special characters, letters and digits. The following translations and shorthands are used:

<u>Input</u>	<u>Translation</u>	<u>Meaning</u>	
~	$\nu$	Restriction	
\	$\lambda$	Abstraction	
0	0	Null process	
'α	$\bar{\alpha}$	Output action	
t	$\tau$	Internal action	
a(nlist)	a.(nlist)	Input prefix	
'a <nlist>	'a.[nlist]	Output prefix	(217)

For *formulas*:

<u>Syntax</u>	<u>meaning</u>	
TT	Truth	
FF	Falsity	
name=name	Equality between names	
name#name	Inequality between names	
formula & formula	Conjunction	
formula   formula	Disjunction	
<action> formula	Possibility modality	
[action] formula	Necessity modality	
Sigma name . formula	Sigma – expression	
Bsigma name . formula	Bound sigma	
Pi name . formula	Universal quantification	
exists name . formula	Existential quantification	
op id. formula	Fixpoint expression	
(formula)	Parenthesis	(218)

op:

mu   min	Least fixpoint operator	
nu   max	Greatest fixpoint operator	(219)

action:

$$\begin{array}{l|l}
 name & \text{Name} \\
 'name & \text{Co-name} \\
 \tau & \text{Tau}
 \end{array} \tag{220}$$

id:

IdENtIFieR BEgiNning WiTH UpPEercaSE LetTEr : *Fixpoint* identifier name  
 iDentTIFIER bEGInNING wITH lOWeRCASE lETtER : Pi-calculus name

Remarks:

1) The restriction needs not be immediately adjacent to the box.

E.g.  $(\nu y)(\nu z)[y][z]A \models \text{Bsigma } x.P \Leftrightarrow (\nu z)[z]A\{a/y\} \models P\{a/x\}$ , where  $a$  is a new name.

2) Fixpoint formulae must be closed. E.g.  $\text{nu } D.<x>(x=b\&D)$  is invalid, but the equivalent formula  $(\text{nu } D(b).<x>(x=b\&D(b)))(b)$  is ok.

The prover will answer "YES" if *agent* is a model for *formula*, "NO" otherwise.

## 16.3 Examples

These are tests for the original model-checker, conducted by prof. Joachim Parrow in 1994.

### 16.3.1 Agent examples

**Example 17** *Bufn (Bagn) is a buffer (bag) of capacity n. Postfix "p" means constructed as a parallel composition, "e" means explicit, "l" means lossy. FBuf is a finite buffer. What follows can be sent directly to MWB for testing.*

```
agent Buf1(i,o) = i(x).'o<x>.Buf1<i,o>
```

```
agent Buf2e(i,o) = i(x).Bufa<i,o,x>
```

```
agent Bufa(i,o,x) = i(y).Bufb<i,o,x,y> + 'o<x>. Buf2e<i,o>
```

```
agent Bufb(i,o,x,y) = 'o<x>.Bufa<i,o,y>
```

```
agent Buf2p(i,o) = (^m)(Buf1<i,m>|Buf1<m,o>)
```

```

agent Buf3pe(i,o) = (~m)(Buf2e<i,m>|Buf1<m,o>)
agent Buf3pp(i,o) = (~m)(Buf2p<i,m>|Buf1<m,o>)

agent Buf4pee(i,o) = (~m)(Buf2e<i,m>|Buf2e<m,o>)
agent Buf4ppe(i,o) = (~m)(Buf2p<i,m>|Buf2e<m,o>)
agent Buf4ppp(i,o) = (~m)(Buf2p<i,m>|Buf2p<m,o>)

agent Bag2e(i,o) = i(x).Baga<i,o,x>
agent Baga(i,o,x) = i(y).Bagb<i,o,x,y> + 'o<x>. Bag2e<i,o>
agent Bagb(i,o,x,y) = 'o<x>.Baga<i,o,y> + 'o<y>.Baga<i,o,x>

agent Bag2p(i,o) = Buf1<i,o>|Buf1<i,o>

agent Bag3pe(i,o) = Bag2e<i,o> | Buf1<i,o>
agent Bag3pp(i,o) = Bag2p<i,o> | Buf1<i,o>

agent Bag4pee(i,o) = Bag2e<i,o> | Bag2e<i,o>
agent Bag4ppe(i,o) = Bag2p<i,o> | Bag2e<i,o>
agent Bag4ppp(i,o) = Bag2p<i,o> | Bag2p<i,o>

agent Mixed3(i,o) = (~m)(Bag2p<i,m>|Buf1<m,o>)
agent Mixed3b(i,o) = (~m)(Buf1<i,m>|Bag2p<m,o>)

agent Buf11(i,o) = i(x).('o<x>.Buf11<i,o>+Buf11<i,o>)
agent Buf21p(i,o) = (~m)(Buf11<i,m>|Buf1<m,o>)
agent Buf31pp(i,o) = (~m)(Buf21p<i,m>|Buf1<m,o>)

agent FBuf(i,o) = i(x).i(y).i(z). 'o<x>. 'o<y>. 'o<z>.0

```

**Example 18** *The following is a collection of finite recursion-free agents, good for purposes of testing the behaviour of the model prover.*

```

agent T0(i,o) = 0
agent T1(i,o) = i(x).0
agent T2(i,o) = 'o<o>.0
agent T3(i,o) = i(x).t. 'o<x>.0
agent T4(i,o) = i(x). 'o<o>.0
agent T5(i,o) = i(x).(t.0+'o<x>.0)
agent T6(i,o) = t.i(x).0
agent T7(i,o) = i(x).i(y). 'o<x>. 'o<y>.0
agent T8(i,o) = i(x).i(y). 'o<y>. 'o<x>.0
agent T9(i,o) = i(x).i(y). 'o<x>. 'o<x>.0
agent T10(i,o) = i(x).i(y). 'o<x>. [x=y] 'o<y>.0
agent T11(i,o) = i(x).i(y).i(z). 'o<x>. 'o<z>. 'o<y>.0

```



```

agent T12(i,o) = i(x).( 'o<x>.0 | i(y). 'o<y>.0)
agent T13(i,o) = T7<i,o>|T7<i,o>
agent T14(i,o) = T11<i,o>|T11<i,o>

```

### 16.3.2 Formulae examples

**Example 19** *OP* Order-preserving of first data:

```

nu L. (([t]L) & ([ 'o] Sigma u.L) & ([i]Pi u.(nu I(u).([t]I(u))
      & ([i]Pi z.I(u)) & ([ 'o] Sigma z.z=u))) (u)))

```

*L* ("nothing has been input yet") must hold after all transitions, except after an input  $i(u)$  when  $I(u)$  holds.  $I(u)$  ("u has been input but not output") holds after all transitions, except after an output  $\bar{o}(u)$  (then nothing more is required) or after an output  $\bar{o}(z)$  with  $z \neq u$  (then it is false). So  $I(u)$  means nothing can be emitted before  $u$ , and  $L$  means that nothing can be emitted before the first received item.

**Example 20** *NO* No spurious output:

```

(Pi p .
  (nu NO(x). ([t] NO(x))
    & ([i] Pi w . (w=x | NO(x)))
    & ([ 'o] Sigma w . (w#x & NO(x)))
  ) (p))

```

$NO(p)$  must hold invariantly after  $\tau$ :s, after non- $p$  inputs and any outputs, thus, it must hold whenever  $p$  has not been input. Nothing is required of the states following an input of  $p$ . But an output of something different from  $p$  falsifies it. So, it holds unless  $p$  can be emitted before it is received. all  $p$ .  $NO(p)$  thus says that any value must be received before it can be emitted. To check it on an agent  $A$ , do the check command on  $(\backslash z)A$  where  $z \notin \text{fn}(A)$ .

**Example 21** *NL* No lost input

```

(nu L . ( ([ 'o] Sigma z . L)
  & [t] L
  & [i] Pi z .
    (L &
      (mu O(x). ((<i>TT | <'o>TT | <t>TT) &
        ([i] Pi w . O(x)) &
        ([ 'o] Sigma w . (w=x | O(x))) &
        [t] O(x))
      )
    )
  )

```



after an output of  $p$  instead  $\text{Twice}(p)$  must hold. Again this is an invariant, falsified by an output of  $p$ , and satisfied by an input of  $p$ . So the only way to falsify  $L$  is to emit  $p$  twice although it has been received only once. An agent needs a dummy abstraction to be checked with this.

**Example 23** *OP2* Order-preserving (second version)

```
(Pi p. Pi q.
(p=q |
(nu L(p,q).
  ([t] L(p,q)) &
  ([’o] Sigma u. L(p,q)) &
  ([i] Pi u. ((u=q) | (u#p & u#q & L(p,q)) |
    (u=p) &
    (nu I(p,q).
      ([t] I(p,q)) &
      ([’o] Sigma u. ((u#p & I(p,q)) | u=p)) &
      ([i] Pi u. ((u#q & I(p,q)) |
        (u=q) &
        (nu O(p,q).
          ([t] O(p,q)) &
          ([i] Pi u. O(p,q)) &
          ([’o] Sigma u. ((u#p & u#q & O(p,q)) | u=p))
        ))
      ))
    ))
  ))
))
```

We express that if  $p$  is input before  $q$ , then  $q$  cannot be output before  $p$ . Consider two data  $p, q$ . The invariant  $L(p, q)$  must hold for all different  $p, q$ . If  $q$  is input there is nothing more to check. If  $p$  is input then  $I(p, q)$ , meaning “ $p$  has been input”, must hold.  $I(p, q)$  holds invariantly, but if  $p$  is output there is nothing more to check, and if  $q$  is input then  $O(p, q)$  holds, meaning “ $p$  has been input before  $q$ ”.  $O(p, q)$  must hold invariantly, but an output of  $p$  satisfies it and an output of  $q$  falsifies it. An agent needs a double dummy abstraction to be checked with this.

**Example 24** *NLW* No lost input: Weak version

```
(nu L .( ([ 'o] Sigma z . L)
          & ([t] L )
          & ([i] Pi z .
            (L &
              (nu O(x).
                ((mu M(x).
                  ((<'o> Sigma w. (w=x | M(x))) |
                   (<t> M(x)) |
                   (<i> exists w. (w#x & M(x)))
                  )
                )(x) &
                ([i] Pi w . O(x)) &
                ([ 'o] Sigma w . (w=x | O(x))) &
                ([t] O(x))
              ))(z)
            )
          )
        )
      )
```

*M(x)* holds if it is possible to reach a state where *x* can be emitted. *O(x)* means that *M(x)* holds invariantly until after *x* has been emitted. So *L* says that when a datum *z* is input *O(z)* must hold afterwards, i.e. , the possibility to emit *z* can never be removed until *z* is emitted. This is weaker than requiring that *z* is eventually emitted!

**Example 25** *DE* Deadlock freedom

```
(nu D.
  (((<t>TT) | (<i> exists w.TT) | (<'o> Sigma w.TT))
   &
  ((([t] D) & ([i] Pi w.D) & ([ 'o] Sigma w.D))))
```

There must be at least one transition, and that should be an invariant.

**Example 26** *TI* Trivial invariance

```
(nu D. ((([t] D) & ([i] Pi w.D) & ([ 'o] Sigma w.D))))
```

Invariants don't get any simpler. But this is only to test the efficiency of the modelchecker - it will have to visit each state.

**Example 27** *NB* No blocking of pending messages

```
(nu L.
  (([t] L) &
   (['o] Sigma w.L) &
   ([i] Pi w.
     (L &
      (mu I(w).
        ((<t> I(w)) |
         (<'o> Sigma u.u=w)
        )
      )(w)
     )
   )
  )
)
```

*L holds invariantly, and after an input  $i(x)$  additionally  $I(x)$  holds. This is a minimal fixedpoint and can only be satisfied if it is possible to reach, through a sequence of zero or more  $\tau$ 's, a state where this  $w$  can be emitted. So all inputs must be available immediately for output (discounting  $\tau$ 's) throughout the execution of the agent.*

## 16.4 Some Model Prover run-through examples

The usage of structural rules, as bringing the next alternative out of the alternative list, and testing conjunctives, are made *implicit*. Also, the presented structures are sometimes shortened, when the information in i.e. the parameter index is not interesting for the particular example. The examples should, of course, be read bottom-up (since we go from conclusion to premise).

**Example 28** *An example featuring suspension, variables:*

$$\begin{array}{l}
 \underline{\text{Valid}} \\
 \text{True}, \mathcal{S} = \{\langle \emptyset \rightarrow A : a \neq W_0 \rangle\} \\
 \frac{\emptyset \rightarrow A : V_0 = a \text{ (Unify, } V_0 := a)}{\mathcal{S} = \{\langle \emptyset \rightarrow A : V_0 \neq W_0 \rangle\}, \Xi = \{\langle \emptyset \rightarrow A : V_0 = a \rangle\}} \text{ (Suspend)} \\
 \frac{\emptyset \rightarrow A : V_0 \neq W_0, \Xi = \{\langle \emptyset \rightarrow A : V_0 = a \rangle\}}{\emptyset \rightarrow A : V_0 \neq W_0 \wedge V_0 = a} \\
 \frac{\emptyset \rightarrow A : \exists y.(V_0 \neq y \wedge V_0 = a)}{\emptyset \rightarrow A : \exists x.\exists y.(x \neq y \wedge x = a)} \\
 \text{where } A \triangleq a.b.0.
 \end{array}$$

**Example 29** *Another example, including possibility modality and a maximum fixpoint:*

$$\begin{array}{l}
 \underline{\text{Valid}} \\
 a \neq b \rightarrow A : \text{True}, \Upsilon \\
 \frac{a \neq b \rightarrow A : U_1, \Upsilon \text{ DISCHARGE}}{a \neq b \rightarrow b.A : \text{False}, \Theta^4 = \{\text{Alt}\langle A, U_1, \Upsilon \rangle\}} \\
 \frac{a \neq b \rightarrow b.A : \langle b \rangle U_1, \Upsilon}{a \neq b \rightarrow b.A : \text{False}, \Theta^3, \Upsilon} \\
 \frac{a \neq b \rightarrow b.A \langle a \rangle U_1, \Theta^3 = \{\text{Alt}\langle b.A, \langle b \rangle U_1, \Upsilon \rangle\}}{a \neq b \rightarrow b.A : \langle a \rangle U_1 \vee \langle b \rangle U_2, \Upsilon \text{ FOLD since } b.A \not\equiv_\alpha A} \\
 \frac{a \neq b \rightarrow b.A : U_1, \Upsilon}{a \neq b \rightarrow A : \text{False}, \Upsilon, \Theta''} \\
 \frac{a \neq b \rightarrow A : \langle b \rangle U_1, \Upsilon, \Theta'' = \{\text{Alt}\langle b.A, U_1, \Upsilon \rangle\}}{a \neq b \rightarrow A : \text{False}, \Theta' = \{\text{Alt}\langle A, \langle b \rangle U_1, \Upsilon \rangle, \text{Alt}\langle b.A, U_1, \Upsilon \rangle\}} \\
 \frac{a \neq b \rightarrow A : \langle a \rangle U_1, \Upsilon, \Theta = \{\text{Alt}\langle A, \langle b \rangle U_1, \Upsilon \rangle\}}{a \neq b \rightarrow A : \langle a \rangle U_1 \vee \langle b \rangle U_1, \Upsilon} \\
 \frac{a \neq b \rightarrow A : \nu X.(\langle a \rangle X \vee \langle b \rangle X)}{\text{where } A \triangleq a.b.P, A \succ b.A, b.A \succ A, \text{ and } U_1 \triangleq \langle a \rangle X \vee \langle b \rangle X, \text{ and } \Upsilon = \{\text{Max}\langle a \neq b, A, \nu X(\langle a \rangle X \vee \langle b \rangle X), U_i \rangle\}}.
 \end{array}$$

## 17 Summary & discussion

We have described a model checking algorithm – "the Model Prover".

**Assumed correct, is this algorithm really efficient compared to, say, the previous Model Checker?**

Yes, it has some efficiency properties;

First, the suspension mechanism, that, proved to be sound, "cuts off" some branches in a perhaps very large proof tree. Second, the usage of annotated parameters and variables clearly has some efficiency benefits over elder techniques; for a  $\forall$ -formula, one had to prove that the formula really was valid for *all activated names*, which of course could be thousands or millions at worst. The same reasoning for the usage of variables; instead of looking for that particular name that would make the  $\exists$ -formula valid, we simply "postpone" the decision of selecting that name by the introduction of a variable as a placeholder for it. The unification process is then an attempt to find that name, but only under certain conditions. This is a more time/effort-saving approach than simply traversing the whole list of names again, trying to find a name that would make the formula valid. The whole approach of the model prover can be addressed as "postponing", i.e. trying to delay as much of the proof search execution as possible, hoping that the search will have been completed before we ever will have to conduct the extended search. The symbolic way of treating names indeed has computational benefits.

**Does the modal  $\mu$ -calculus offer enough expressive power to be of interest?**

As we have seen, it is expressive enough to let us reason about liveness, deadlock freedom, no lost input in buffers, no blocking, order preserving, etc. It cannot express real-time properties, like maximum allowed execution time and similar.

**Can further improvements be made to the model prover?**

Yes, the *caching* of commitments, would be a speed improvement. The fact that the commitment relation has to be re-calculated every time a modal rule is to be used by the calculus is obviously an inefficient solution. Letting the commitment calculator cache its most previous work instead of re-calculating every time would save efforts. Another, more radical way was suggested by Torkel Franzén - A *System of agents*:

**Theorem 5** *The idea here is to eliminate all syntax in  $\pi$ -calculus agents and transform the whole "agent space" to these new system of symbols. This way, the whole graph of agents would be fully developed at first, before any model checking can take place. This eliminates all calculation of commitment relations, but can perhaps demand a lot of memory usage when fully developed for a large system of agents. We have an infinite set of names,  $a, b, \dots$ , and a set of agent symbols,  $F, G, \dots$ . The silent action symbol  $\tau$  is NOT among these names. Each agent symbol has an arity and belongs to one of three disjoint classes: Process Symbols, Emitter Symbols and Absorber Symbols.*

*An agent is an expression of the form  $F(a_1 \dots, a_n)$ , where  $F$  is an  $n$ -ary agent symbol and  $a_1 \dots, a_n$  are names. An agent is a process if  $F$  is a process symbol.*

A schematic agent is an expression of the form  $F(x_1, \dots, x_n)$  where  $x_1, \dots, x_n$  are different variables. We write schematic agents as  $f(\mathcal{X})$  where  $\mathcal{X}$  stands for such a sequence. We say that  $x$  occurs in  $\mathcal{X}, \mathcal{U}$  if it is one of the variables in the sequence, and it occurs in  $\mathcal{X}, \mathcal{U}$  if it is  $u$  or occurs in  $\mathcal{X}$ . Similarly for " $\mathcal{Y}$  is included in  $\mathcal{X}$ " and " $\mathcal{Y}$  is included in  $\mathcal{X}, \mathcal{U}$ ". A System Of Agents is given by a set of process symbols and a set of rules for the symbols of following form:

$$F(\overset{u}{\mathcal{X}}) \rightarrow G(\mathcal{Y}) \quad (221)$$

where  $G$  is a symbol in the system,  $u$  occurs in  $\mathcal{X}$  and  $\mathcal{Y}$  is included in  $\mathcal{X}$ .

$$F(\overset{\tau}{\mathcal{X}}) \rightarrow G(\mathcal{Y}) \quad (222)$$

where  $\mathcal{Y}$  is included in  $\mathcal{X}$ .

$$F(\overset{\tau}{\mathcal{X}}_{\{u=v\}}) \rightarrow G(\mathcal{Y}) \quad (223)$$

where  $\mathcal{Y}$  is included in  $\mathcal{X}$  and  $u, v$  occur in  $\mathcal{X}$ .

An emitter symbol  $F$  has exactly one rule, of the form

$$F(\overset{u}{\mathcal{X}}) \overset{**}{\rightarrow} G(\mathcal{Y}) \quad (224)$$

where  $\mathcal{Y}$  is included in  $\mathcal{X}, \mathcal{U}$  and  $u$  may or may not occur in  $\mathcal{X}$ . Here  $G$  is a process symbol or an emitter symbol.

An Absorber symbol, finally, has exactly one rule, of the form

$$F(\overset{u}{\mathcal{X}}) \overset{**}{\leftarrow} G(\mathcal{Y}) \quad (225)$$

where  $u$  does not occur in  $\mathcal{X}$ , and  $\mathcal{Y}$  is included in  $\mathcal{X}, \mathcal{U}$ . here  $G$  is a process symbol or an absorber symbol.

An instance of a rule is obtained by substituting names for the variables in a rule. This substitution is subject to one restriction: if the  $u$  in the emitter symbol does not occur in  $\mathcal{X}$  (this is called a new emission, the name substituted for  $u$  must not be substituted for any variable in  $\mathcal{X}$ ). A finite system of agents is one that has finitely many agent symbols.

Another efficiency aspect is the avoidance of duplication in the molecular components of the model prover. There is no logical difference if the Alternatives, suspended sequents and conjunctive sequents sets at every point of updating are removed from duplicates, but it saves some memory and future calculation.

As proof trees grow large, they perhaps re-visit a configuration that has been developed before. A caching technique for whole branches of a search tree would render the execution more effective, the need for re-calculation is then reduced further. Of course, all effectiveness improving methods will have to be taken into account from a cost/benefit point of view – large caches may take long time to look through, longer than the actual re-calculation would



have taken...

The development of sequent calculus based model checkers for newer calculi; for example the Update or the Fusion Calculus with their notion of variable scope are still un-explored topics, as well as logic for the open bisimulation equivalence.

## Part IV

# Appendix

## A Implementation results

The implementation of The Model Prover was done as a totally integrated part of the MWB command set, i.e., the original model checker was NOT omitted from MWB, but co-exists with the Model Prover. As a consequence, the user is free to select any algorithm for performing model checking, simply change `check` to the word `prove` instead. To measure the efficiency of both of the algorithms, we use the notion of *inference*, i.e. the number of times a recursive call is made to the algorithm. If each recursive call takes approximately the same amount of time in both algorithms, (and a simple average calculation suggests they are alike in the same order of magnitude), we simply can compare the answers and inference number for both algorithms to decide which one does the faster job!

**Table 1** *The agents and formulas are the same as described in chapter 17. "P" and "C" stands for "The Model Prover" and "checker", respectively. A "Yxxx" or "Nyyy" entry means that the respective algorithm answered "YES" after xxx inferences, or "No" after yyy inferences, respectively. A "\*" means that the entry was not tested, or required too long time (> 6 min) to be completed.*

Formula→	TI		OP		NB		DE	
Agent ↓	P	C	P	C	P	C	P	C
<i>Buf1</i>	Y22	Y27	Y22	Y27	Y32	Y45	Y44	Y54
<i>Buf2p</i>	Y41	Y6739	Y44	Y115	N67	N48	Y84	Y14110
<i>Buf2e</i>	Y32	Y5784	Y35	Y100	N52	N42	Y65	Y12200
<i>Buf3pe</i>	Y51	*	Y76	Y776	Y76	*	Y104	*
<i>Buf3pp</i>	Y60	*	Y97	Y931	N94	N111	Y122	*
<i>Buf4pee</i>	Y51	*	Y136	Y6348	N79	N251	Y103	*
<i>Buf4ppe</i>	Y60	*	Y156	Y7539	Y156	*	Y122	*
<i>Buf4ppp</i>	Y88	*	Y212	Y22888	N121	N212	Y180	*
<i>Buf2lp</i>	Y51	*	N96	N778	N93	N51	*	*
<i>Buf3lpp</i>	Y89	*	N89	*	N26	*	Y181	*
<i>FBuf</i>	Y69	Y820	Y42	Y308	N26	N49	N136	N84
<i>Bag2p</i>	Y42	Y12003	N35	N20	Y52	*	Y86	Y22732
<i>Bag2e</i>	Y32	Y7502	N35	N18	Y32	*	Y86	Y13468
<i>Bag4pee</i>	Y62	*	N134	*	N134	N30	Y128	*
<i>Bag4ppe</i>	Y82	*	N207	*	N207	N30	Y170	*
<i>Bag4ppp</i>	Y82	*	N207	*	Y92	*	Y170	*
<i>Mixed3</i>	Y81	*	N188	N337	N160	N122	Y165	*
<i>T13</i>	Y1349	Y164769	N174	N30	N26	N60	N2761	N110

Formula→	<i>NLW</i>		<i>NL</i>	
Agent ↓	<i>P</i>	<i>C</i>	<i>P</i>	<i>C</i>
<i>Buf1</i>	<i>Y50</i>	<i>Y75</i>	<i>Y46</i>	<i>Y72</i>
<i>Buf2p</i>	<i>Y119</i>	<i>Y58538</i>	<i>Y98</i>	*
<i>Buf2e</i>	<i>Y78</i>	<i>Y48675</i>	<i>Y70</i>	*
<i>Buf3pe</i>	<i>Y718</i>	*	<i>Y386</i>	*
<i>Buf3pp</i>	<i>Y1106</i>	*	<i>Y521</i>	*
<i>Buf4pee</i>	<i>Y1345</i>	*	<i>Y713</i>	*
<i>Buf4ppe</i>	<i>Y1908</i>	*	<i>Y936</i>	*
<i>Buf4ppp</i>	<i>Y7817</i>	*	<i>Y2048</i>	*
<i>Buf2lp</i>	*	*	*	*
<i>Buf3lpp</i>	*	*		
<i>FBuf</i>	*	*	*	*
<i>Bag2p</i>	<i>Y159</i>	<i>Y98177</i>	<i>N130</i>	<i>N56</i>
<i>Bag2e</i>	<i>Y93</i>	<i>Y61365</i>	<i>N72</i>	<i>N79</i>
<i>Bag4pee</i>	<i>Y266</i>	*	<i>N186</i>	<i>N231</i>
<i>Bag4ppe</i>	<i>Y380</i>	*	<i>N284</i>	<i>N74</i>
<i>Bag4ppp</i>	<i>Y588</i>	*	<i>N438</i>	<i>N74</i>
<i>Mixed3</i>	<i>Y1258</i>	*	<i>N552</i>	<i>N154</i>
<i>T13</i>	*	*	*	*

Some conclusions: We see that symbolic name-manipulation as in the model prover indeed has advantages over the explicit approach. However, on smaller testings, such as T13 and OP, we see that the checker finds the answer out quicker than the prover! This can probably be explained by the prover needing a bit of computational "overhead", which cannot be recaptured on smaller tests. On larger tests, although, this overhead pays off well. (Compare for instance, Buf2p:NB, an agent with 8 states, where the checker beats the prover by 48 to 67 inferences, but when the same test is conducted for Buf4ppp, a size 213 agent, the opposite holds with 121 to 212 inferences.

**Table 2** By using the MWB time command, we can get the real-time results for the different calculations. In no case does the real-time result contradict the result as indicated by the reference number of table 1, i.e. we can regard the time needed for a computation as linearly related to the reference number. Below, the number entry means the computational time in seconds required for deciding the entry.

The tests were performed on a Sun Ultra-1 station with 128 MB of RAM memory and a 167 MHz UltraSparc processor running Solaris 2.5.1. A "xxx min"-entry means that the checking was not completed after xxx minutes.

Formula→	TI		OP		NB		DE	
Agent ↓	P	C	P	C	P	C	P	C
Buf1	0.007	0.006	0.004	0.006		0.007	0.011	
Buf2p	0.025	9.39	0.028	0.094			0.046	17.4
Buf2e	0.007	4.91	0.028	0.094			0.015	8.19
Buf3pe	2.4	60min	0.07	1.05			0.4	10min
Buf4pee			0.147	12.32				
Buf4ppe			0.21	23.1			0.17	
Buf4ppp			0.39	64.7	0.2	0.5		
Buf2lp			0.16	1.09			0.07	30min
Buf3lpp	0.14				0.18		0.25	
FBuf	0.008	0.18	0.026	0.012			0.028	20.74
Bag2p	0.007	11.52	0.009	0.007			0.013	8.04
Mixed3			0.339	0.55				
T13	8.63	34.3	0.12	0.02	0.01	0.02	19.6	0.03

Formula→	NLW		NL	
Agent ↓	P	C	P	C
Buf1	0.011	0.017		
Buf2p	0.08	67.3		
Buf2e	0.023	28.2		
FBuf			0.05	0.34
Bag2p	0.087	91.93	0.062	0.029
Bag2e	0.026	36.44	0.017	0.03
Bag4pee	0.3		0.83	0.07
Bag4ppp	9.6		3.7	0.07
Mixed3			2.11	0.2

## B References

- [1] **Trollkarlen Blir Ingenjör - Formella Metoder: En Översikt**  
JOACHIM PARROW  
KTH Stockholm, Sweden May 1997  
<http://www.sics.se/~joachim/hatttext.ps.Z>  
Shorter version also published as "Programmeraren som Shaman",  
Forskning & Framsteg #2, february 1998  
[http://www.fof.se/artiklar/98\\_2\\_14.htm](http://www.fof.se/artiklar/98_2_14.htm)
  
- [2] **Foundational Calculi For Programming Languages**  
(To Appear In The CRC Handbook of Computer Science and Engineering)  
BENJAMIN C. PIERCE  
Computer Laboratory, University of Cambridge, UK, december 1995.
  
- [3] **Beräkningsbarhet för dataloger: Från  $\lambda$  till P**  
KENT PETERSSON  
Programmeringsmetodikgruppen,  
Inst. f. Informationsbehandling, Göteborgs Universitet/Chalmers  
Bokförlaget Aquila 1987/1988.
  
- [4] **Mathematical Foundations of Programming**  
FRANK S. BECKMAN  
Brooklyn College of the City University of New York  
The System Programming Series, Addison-Wesley 1980
  
- [5] **Introduction to Combinators and  $\lambda$ -calculus**  
J. ROGER HINDLEY  
Dept. of mathematics and Computer Science, University College, Swansea  
JONATHAN P. SELLDIN  
Dept. of Mathematics, Concordia University, Montreal  
London Mathematical Society Student Texts 1  
Cambridge University Press 1986
  
- [6] **An Introduction to Modal Logic**  
G. E. HUGHES & M. J. CRESSWELL  
Victoria University of Wellington  
Methuen & Co, London 1968
  
- [7] **The Polyadic  $\pi$ -calculus: a Tutorial**  
ROBIN MILNER  
Dept. of Computer Science, University of Edinburgh  
ECS-LFS-91-180 Technical Report

October 1991

- [8] **Communication and Concurrency**  
ROBIN MILNER  
Dept. of Computer Science, University of Edinburgh  
Prentice-Hall International Series In Computer Science 1989
- [9] **An Intuitionistic Predicate Logic Theorem Prover**  
TORDEL FRANZEN, DAN SAHLIN & SEIF HARIDI  
Swedish Institute Of Computer Science  
Journal Of Logic Computation Vol 2, #5, pp 619-656, 1992
- [10] **A de Bruijn notation for the  $\pi$ -calculus**  
SIMON J. AMBLER  
University of London  
Queen Mary & Westfield College Technical Report # 569 1991
- [11] **A theorem-proving approach to deciding properties of finite-control agents**  
(Extended Abstract)  
TORDEL FRANZEN  
SICS Technical Report T96:02  
Swedish Institute of Computer Science 1996
- [12] **A Verification Tool for the Polyadic  $\pi$ -calculus**  
BJÖRN VICTOR  
Department of Computer Science, Uppsala University  
DoCS Licentiate Thesis 94/50 May 1994  
<http://www.docs.uu.se/~victor/tr/docs-tr-94-50.ps.gz>
- [13] **Algebraic Theories for Name-Passing Calculi**  
JOACHIM PARROW  
Swedish Institute of Computer Science  
DAVID SANGIORGI  
Dept. of Computer Science, University of Edinburgh  
Journal of Information & Computation #120(2),pp 174-197  
February 1996
- [14] **Model Checking Mobile Processes**  
MADS F. DAM  
Swedish Institute of Computer Science  
Information and Computation vol. 129, #1, pp 35-51 august 1996  
SICS Research Report R94:01
- [15] **A Note on model checking in the modal  $\nu$ -calculus**

- GLYNN WINSKEL  
Dept. of Computer Science, Aarhus University, Denmark  
Theoretical Computer Science #83, pp 157-167 1991
- [16] **Local model checking in the modal mu-calculus**  
COLIN STIRLING & DAVID WALKER  
Dept. of Computer Science, University of Edinburgh  
Theoretical Computer Science #89, 1991
- [17] **Modal and Temporal Logics for Processes**  
COLIN STIRLING  
LFCS, Dept. of Computer Science, University of Edinburgh 1992  
Technical Report ECS-LFCS -92-221,
- [18] **Algebraic Laws for Nondeterminism and Concurrency**  
MATTHEW HENNESSY & ROBIN MILNER  
Dept. of Computer Science, University of Edinburgh  
Journal of the Association for Computing Machinery (ACM), vol 32, #1, 1985
- [19] **Modal Logics for Mobile Processes**  
ROBIN MILNER  
Dept. of Computer Science, University of Edinburgh  
JOACHIM PARROW  
Swedish Institute of Computer Science  
DAVID WALKER  
University of Technology, Sydney, Australia  
Theoretical Computer Science vol 114, pp 149-171 1993.
- [20] **A Calculus of Mobile Processes, Part I & II**  
ROBIN MILNER  
Dept. of Computer Science, University of Edinburgh  
JOACHIM PARROW  
Swedish Institute of Computer Science  
DAVID WALKER  
University of Technology, Sydney, Australia  
Journal of Information & Computation, vol 100 pp 1-77, september 1992
- [21] **The Art Of Prolog - Advanced Programming Techniques**  
LEON STERLING & EHUD SHAPIRO  
2nd Edition, MIT Press,  
Massachusetts Institute of Technology (MIT) 1986/1994
- [22] **SICStus Prolog User's Manual**  
The Programming Systems Group,

Swedish Institute of Computer Science  
Rel. 3#0, June 1995

- [23] **Logic Programming & The Intuitionistic Sequent Calculus**  
TORHEL FRANZEN  
Swedish Institute Of Computer Science  
SICS Research Report R88002, 1988
  
- [24] **Elements of ML Programming**  
JEFFREY D. ULLMAN  
Stanford University  
Prentice-Hall 1994
  
- [25] **Grundläggande logik**  
KAJ B. HANSEN  
Computer Science Dept, CSD, Uppsala University  
3:e upplagan, Studentlitteratur 1994/1997
  
- [26] **Logic and Structure**  
DIRK VAN DALEN  
Mathematical Institute, Utrecht University  
3:rd Edition, Springer Verlag Berlin, Heidelberg 1994
  
- [27] **Algebraic Theory of Processes**  
MATTHEW HENNESSY  
Dept. of Computer Science, University of Edinburgh  
MIT Press, Massachusetts Institute of Technology 1988
  
- [28] **L<sup>A</sup>T<sub>E</sub>X - A Document Preparation System**  
LESLIE LAMPORT  
Digital Equipment Corporation  
Addison-Wesley Publishing Company 1986
  
- [29] **Lambda Calculi - A Guide for Computer Scientists**  
CHRIS HANKIN  
Dept. of Computing, Imperial College, London  
Clarendon Press 1994
  
- [30] **Matematiklexikon**  
JAN THOMPSON (ed.)  
Wahlström & Widstrand 1991
  
- [31] **ML for the Working Programmer**  
L. C. PAULSON



University of Cambridge  
2nd edition, Cambridge University Press 1996

- [32] **Discrete Mathematics**  
KENNETH A. ROSS & CHARLES R.B. WRIGHT  
Dept. Of Mathematics, University Of Oregon  
3rd Edition, Prentice Hall 1992
  
- [33] **The Formal Semantics Of Programming Languages - An Introduction**  
GLYNN WINSKELL  
MIT Press  
Foundations Of Computing Series, 1993
  
- [34] **Fullständighetsatsen för Predikatlogiken via Sekventkalkyl**  
INGER SIGSTAM  
Dept. of mathematics, Uppsala University  
March 1995
  
- [35] **Lambda Calculus Notation with Nameless Dummies**  
A tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem  
N.G DE BRUIJN  
Indagationes Mathematicae, Vol 34, Fasciculus 5  
Koninklijke Nederlandse Akademie van Wetenschappen 1972