Acta Informatica 30, 1–59 (1993)

Acta Informatica

Plain CHOCS
A second generation calculus for higher order processes

Bent Thomsen

European Computer-Industry Research Centre, Arabellastrasse 17, W-8000 Munich 81, Federal Republic of Germany

Received July 1, 1991/August 11, 1992

Abstract. In this paper we present a calculus of communicating systems which allows one to express sending and receiving processes. We call this calculus Plain CHOCS. The calculus is a refinement of our earlier work on the calculus of higher order communicating systems (CHOCS).

Essential to the new calculus is the treatment of restriction as a static binding operator on port names. The new calculus is given an operational semantics using labelled transition systems which combines ideas from the applicative transition systems described by Abramsky and the transition systems used for CHOCS. The new calculus enjoys algebraic properties which are similar to those of CHOCS only needing obvious extra laws for the static nature of the restriction operator.

Processes as first class objects enable description of networks with changing interconnection structure, and there is a close connection between the Plain CHOCS calculus and the π-Calculus described by Milner, Parrow and Walker: the two calculi can simulate one another.

Recently object oriented programming has grown into a major discipline in computational practice as well as in computer science. From a theoretical point of view object oriented programming presents a challenge to any metalanguage since most object oriented languages have no formal semantics. We show how Plain CHOCS may be used to give a semantics to a prototype object oriented language called *O*.

1 Introduction

Several attempts to extend calculi for concurrent systems with the capability of describing processes as first class objects have recently been put forward [AstReg87; Bou89; Chr88; KenSle88; Nie89; Tho89; GiaMisPra90]. The justification for having process passing in a calculus of communicating systems may

be found in the powerful abstraction technique it yields, just as having (higher order) functions or procedures in traditional programming languages. Many systems can easily be described using process passing, some are even most naturally described in this way. As an excellent example take the system consisting of a satellite and an earth station originally described by Christensen in [Chr88]. One interesting property of this system is that the satellite is physically far away from the earth station. If the program controlling the satellite has to be changed, either because of a program error or because the job of the satellite is to be changed, then it would be preferable to be able to send a new program to the satellite, stop the old program and run the new program instead. Alternatively we would have to send a space shuttle to take the satellite out of orbit to bring it back to earth for reprogramming and then relaunch it, a rather expensive strategy. A reprogrammable system consisting of two components could be specified, in a CCS/CSP like syntax, as follows:

$$Sat = newprg?x.(x|(int?.Sat + error?.Sat + end?.Sat))$$
$$Earth = newprg!Job_1.newprg!Job_2 \ldots$$

The satellite is ready to receive a new job on the *newprg* channel. After reception it acts according to this job until it is "interrupted" either by a new job or because a program error has occurred or because the job has finished. In this example we are beyond CCS/CSP because of $newprg?x.(x|\ldots)$, what we receive on the *newprg* channel is a program (a process), we then run this program in parallel with the rest of the system.

In [Tho89] we showed how to extend CCS with processes as first class objects and we presented a calculus of higher order communicating systems (CHOCS) allowing processes to be sent and received in communication. Several examples showed the usefulness of this calculus. One result of the approach taken in [Tho89] was that almost all the algebraic laws for CCS carried over unchanged and only obvious new laws for process passing were introduced. A major result was the simulation of the (Lazy) λ-Calculus showing that rather important computational phenomena could be modelled.

But some peculiarities may arise due to the dynamic binding of port names in processes sent and received. Port names that intuitively would be considered restricted or bound can become unbound and vice versa as e.g.

$$(1) \qquad (b?x.(x|q))|((b!p'.p)\backslash a) \xrightarrow{\tau} p'|q|(p\backslash a)$$

$$(2) \qquad (b?x.((x|q)\backslash a))|(b!p'.p) \xrightarrow{\tau} ((p'|q)\backslash a)|p.$$

In (1) any occurrence of a in p' becomes unbound after the communication even though we would expect them to be bound if we analyse the system before the communication. In (2) we have the opposite situation. Now any occurrence of a in p' unbound before the communication would be bound after the communication. These examples show that sending the process p' amounts to passing the text of p'. This is closely related to the treatment of function parameters in LISP as originally defined by McCarthy and often referred to as dynamic binding. This parameter mechanism is complicated to work with when analysing the behaviour of programs from their text.

The approach in [Tho89] was chosen because the semantics of CHOCS could be given as a straightforward extension of the CCS semantics and because it yielded simple algebraic laws. However, some of the laws included reference to the sort of the process (i.e. the set of port names the process might use). The calculation of the sort is either a costly calculation needing to run the process (or even worse needing all possible runs of the process) or a very rough approximation to the actual sort. This approximation often yields infinite sort for processes intuitively having finite sort.

Inspired by the idea presented in [EngNie86; MilParWal89] of the restriction operator $p\backslash a$ being a scope binder, which intuitively should bind all occurrences of a in p, we now present a calculus of higher order communicating systems with static binding of port names by restriction. We call this calculus Plain CHOCS.

We are looking for a calculus which has the property that scope extrusion, as we call the technique to take care of the problem in (1) above, will automatically take care of a static binding mechanism for the restriction operator. For example (1) becomes:

$$(3) \qquad (b?x.(x|q))|((b!p'.p)\backslash a) \xrightarrow{\tau} (p'\{c/a\} \,|q|\, p\{c/a\})\backslash c,$$

where $\{c/a\}$ is a name substitution such that c does not belong to the set of free names in q and the restriction will therefore not bind any port in q, only in p and p'. Also scope intrusion, as we call the problem in (2), will be taken care of by a new definition of process substitution which takes the static nature of the restriction operator into account. Therefore (2) above becomes:

$$(4) \qquad (b?x.((x|q)\backslash a))|(b!p'.p) \xrightarrow{\tau} ((x|q)\backslash a)\,[p'/x]|p \equiv ((p'|q\{c/a\})\backslash c)|\,p,$$

where $\{c/a\}$ is a name substitution such that c does not belong to the set of free names in p' and the restriction will therefore not bind any port in p' only in q. To support the linking of processes received in communication with processes in the receiving environment it turns out that it is interesting to have the capability of describing a kind of dynamic binding of port names of processes received in communication. This is obtained by allowing free names to be renamed to bound names upon reception of a process[1]:

$$(5) \quad (b?x.((x[a\mapsto a']|\{a'/a\}\ q)\backslash a'))|(b!p'.p) \xrightarrow{\tau} ((p'[a\mapsto a']|\{a'/a\}\ q)\backslash a')|\,p,$$

where a' does not belong to the set of free names of p' and q. This construction simulates the behaviour described in (2). However, we can not program the behaviour described in (1) since in Plain CHOCS a bound name remains bound and can never become unbound again.

To illustrate these concepts, before presenting a formal syntax and semantics of the Plain CHOCS calculus, we first study a small example.

[1] Recently Milner [Mil91] and Sangiorgi [San92] have approached this by allowing λ-abstractions over port names

Example 1.1 *The example consists of a simple user/resource system similar to the system studied in* [EngNie86]. *The system is constructed from a number of users, a resource manager and a resource. In this example the resource is a process which takes in a number and multiplies it by 2. A resource is obtained on the c channel, then put into use in parallel with the user process. Note how free names of the resource are renamed and bound when received by the user process.*

$$U_1 = c?x.(x[b \mapsto a] \mid a!8.a?y.d_1!y.nil) \backslash a$$
$$U_2 = c?x.(x[b \mapsto a] \mid a!5.a?y.d_2!y.nil) \backslash a$$
$$RM = (c!(R).fin?.RM) \backslash fin$$
$$R = b?x.b![2*x].fin!.nil$$
$$SYS = (U_1 \mid U_2 \mid RM) \backslash c.$$

The fin! signal from the resource R tells the resource manager RM when the resource has finished its task for a user. The resource manager can then (recursively) restore itself and thus provide a resource for other users. The restriction of fin ensures that there is a private communication channel between resource and resource manager which can not be interfered by any user process.

It is interesting to observe how the system executes and how scope extrusion takes care of preserving private links with the sending process. We give an example of one execution sequence where U_2 gets the resource first.

$SYS = (U_1 \mid U_2 \mid RM) \backslash c$

$\downarrow \tau$ *Since* $U_2 \xrightarrow{c?x} U_2' = (x[b \mapsto a] \mid a!5.a?y.d_2!y.nil) \backslash a$ *and*
$\mathscr{RM} \xrightarrow{c!\{fin\}^R} RM' = fin?.RM$

$(U_1 \mid ((R[b \mapsto a] \mid a!5.a?y.d_2!y.nil) \backslash a \mid fin?.RM) \backslash fin) \backslash c$

$\downarrow \tau$ *Since* $R[b \mapsto a] \xrightarrow{a?x} (b![2*x].fin!.nil)[b \mapsto a]$ *and*
$a!5.a?y.d_2!y.nil \xrightarrow{a!5} a?y.d_2!y.nil$

$(U_1 \mid (((b![2*5].fin!.nil)[b \mapsto a] \mid a?y.d_2!y.nil) \backslash a \mid fin?.RM) \backslash fin) \backslash c$

$\downarrow \tau$ *Since* $(b![2*5].fin!.nil)[b \mapsto a] \xrightarrow{a!10} (fin!.nil)[b \mapsto a]$ *and*
$a?y.d_2!y.nil \xrightarrow{a?y} d_2!y.nil$

$(U_1 \mid (((fin!.nil)[b \mapsto a] \mid d_2!10.nil) \backslash a \mid fin?.RM) \backslash fin) \backslash c$

$\downarrow d_2!10$ *Since* $d_2!10.nil \xrightarrow{d_2!10} nil$

$(U_1 \mid (((fin!.nil)[b \mapsto a] \mid nil) \backslash a \mid fin?.RM) \backslash fin) \backslash c$

$\downarrow \tau$ *Since* $(fin!.nil)[b \mapsto a] \xrightarrow{fin!} nil[b \mapsto a]$ *and* $fin?.RM \xrightarrow{fin?} RM$

$(U_1 \mid ((nil[b \mapsto a] \mid nil) \backslash a \mid RM) \backslash fin) \backslash c$

$\downarrow \tau$

$\vdots$

This derivation of transitions illustrates how the system may evolve. However, the linear representation of the system in the Plain CHOCS *syntax does not show very well how the underlying process network dynamically reconfigures itself.*

As an attempt to illustrate this the following cartoon is intended to show how the system evolves spatially when going through the first of the above transitions:

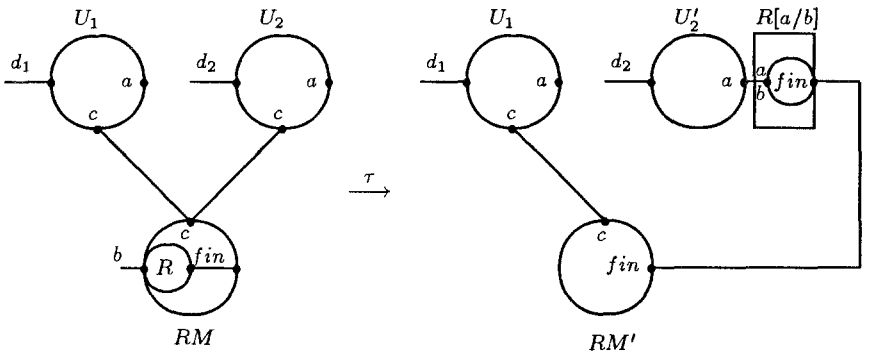

Fig. 1. Dynamic reconfiguration of user/resource system

We have adopted the convention from the process diagrams in [MilParWal89] *and displayed private links inside the circles representing processes and public links along the edges of the connections. The box around the resource R is symbolising the renaming of the public name b to the private name a.*

Note that the number of users and resources is not hard wired into the system. As for the system studied in [EngNie86] *we may add any number of users or resources without changing the structure of the overall system e.g.:*

$$SYS_1 = (U_1 | \ldots | U_n | RM_1 | \ldots RM_m) \backslash c.$$

The above system is very simple, but it easily generalises to systems with a queue system for resource requests from users, multiple resources or even systems where the resource is returned to the resource manager instead of just stopping and allowing a new copy to be used. Some quite elaborate examples of user/resource systems with the above facilities which use process passing have been studied by Cozens in [Coz90]. *This work presents a promising motivation for the use of process passing in system description.*

The paper is organised as follows: In Sect. 2 we present the calculus. The syntax is essentially that of CHOCS, but with the renaming construct restricted to the form $[b \mapsto a]$ meaning b is renamed to a and all other names are not affected. We introduce the notion of free names, free variables and substitution, and the operational semantics is given in terms of a labelled transition system.

In Sect. 3 we present an abstracting equivalence between Plain CHOCS processes. This equivalence resembles a merge between the applicative bisimulation of [Abr90], the higher order bisimulation of [Tho89] and the strong ground bisimulation of [MilParWal89]. We also present the algebraic properties of Plain CHOCS with respect to this equivalence.

The connection to Mobile Processes [MilParWal89] is made explicit in Sect. 4. Here we present two translations, one from Plain CHOCS to Mobile Processes and one in the reverse direction.

Object-oriented programming is drawing a lot of interest from practical programming. Object-oriented programming languages pose a lot of interesting questions, also from a theoretical point of view. One such question is finding appropriate formal semantics for such languages. In Sect. 5 we apply the Plain CHOCS formalism to the semantic study of a toy object-oriented programming language *O*.

Finally in Sect. 6 we round off the discussion of process passing in calculi for communicating systems. We present a brief view of alternatives to the theory presented in this paper, and we give some directions for further studies.

2 Syntax and semantics

The syntax of Plain CHOCS is essentially that of "dynamic" CHOCS, but with the renaming construct restricted to the form $[b \mapsto a]$ meaning b is renamed to a and all other names are not affected.

Processes are built from the inactive process *nil*, three types of action prefixing, often referred to as input, output and tau prefix, (nondeterministic) choice, parallel composition, restriction, renaming and variables to be bound by input prefix. We presuppose an infinite set *Names* (the set of port names) ranged over by $a, b, c, \ldots$ and an infinite set V of process variables ranged over by $x, y, z, \ldots$. We denote by *Pr* the set of expressions built according to the following syntax:

$$p ::= nil \mid a?\, x.p \mid a!\, p'.p \mid \tau.p \mid p + p' \mid p \mid p' \mid p \backslash a \mid p[a \mapsto b] \mid x.$$

To avoid heavy use of brackets we adopt the following precedence of operators: restriction or renaming > prefix > parallel composition > choice.

We shall write $p[S]$ for $p[a \mapsto b]$ where $S = a \mapsto b$ and let $Dom(S) = \{a\}$ and $Im(S) = \{b\}$. The operator $\backslash a$ acts as a kind of λ-binder for port names (elements of *Names*) in a sense to be formalised later, e.g. we have a notion of α-convertibility of restricted names. To formalise this we define the set of free names $fn(p)$ of a process p.

Definition 2.1 We define free names $fn(p)$ structurally on p:

$$\begin{aligned}
fn(nil) &= \emptyset \\
fn(a?\, x.p) &= \{a\} \cup fn(p) \\
fn(a!\, p'.p) &= \{a\} \cup fn(p') \cup fn(p) \\
fn(\tau.p) &= fn(p) \\
fn(p + p') &= fn(p) \cup fn(p') \\
fn(p \mid p') &= fn(p) \cup fn(p') \\
fn(p \backslash a) &= fn(p) \backslash \{a\} \\
fn(p[S]) &= fn(p) \cup Dom(S) \cup Im(S) \\
fn(x) &= \emptyset.
\end{aligned}$$

Intuitively one might expect the clause for renaming to be $fn(p[S]) = (fn(p) \setminus Dom(S)) \cup Im(S)$ since process p can not interact with its environment over names in $Dom(S)$. However, the renaming construct can interact with processes having ports in $Dom(S)$, so in order to be on the safe side we let the set of free names of processes constructed using the renaming construct carry a potential overhead since it is not necessarily the case that the names in $Dom(S) \cup Im(S)$ are going to be used, but the overhead is necessary since we may receive processes in communication with free names which will be renamed by S. The free names of Plain CHOCS processes are going to play an important rôle in the definition of the semantics of the language and as we shall see in the next section, where we define a notion of equivalence, the free names are the windows through which we can observe the processes. As opposed to the static sort of "dynamic" CHOCS we point out that processes to be sent contribute to the free names of the overall system, whereas the empty set of free names is ascribed to process variables.

We may need to syntactically substitute one port name for another. Using the above definition we may now define a name substitution.

Definition 2.2 First for $a, b, c \in Names$ let

$$\{b/c\}\, a = \begin{cases} b & \text{if } c = a \\ a & \text{otherwise} \end{cases}$$

Then name substitution $\{b/c\}\, p$ is defined structurally on p:

$$\begin{aligned}
\{b/c\}\, nil &\equiv nil \\
\{b/c\}(a?\, x.p) &\equiv (\{b/c\}\, a)?\, x.(\{b/c\}\, p) \\
\{b/c\}(a!\, p'.p) &\equiv (\{b/c\}\, a)!(\{b/c\}\, p').(\{b/c\}\, p) \\
\{b/c\}(\tau.p) &\equiv \tau.(\{b/c\}\, p) \\
\{b/c\}(p + p') &\equiv (\{b/c\}\, p) + (\{b/c\}\, p') \\
\{b/c\}(p \mid p') &\equiv (\{b/c\}\, p) \mid (\{b/c\}\, p') \\
\{b/c\}(p \setminus a) &\equiv \begin{cases} p \setminus a & \text{if } a = c \\ (\{b/c\}(\{d/a\}\, p)) \setminus d & \text{otherwise for some } d \notin fn(p \setminus a) \cup \{b\} \end{cases} \\
\{b/c\}(p[a \mapsto a']) &\equiv (\{b/c\}\, p)[(\{b/c\}\, a) \mapsto (\{b/c\}\, a')] \\
\{b/c\}(x) &\equiv x.
\end{aligned}$$

Input prefix is a variable binder. This implies a notion of free and bound variables.

Definition 2.3 We define the set of free variables $FV(p)$ structurally on p:

$$\begin{aligned}
FV(nil) &= \emptyset \\
FV(a?\, x.p) &= FV(p) \setminus \{x\} \\
FV(a!\, p'.p) &= FV(p) \cup FV(p') \\
FV(\tau.p) &= FV(p) \\
FV(p + p') &= FV(p) \cup FV(p') \\
FV(p \mid p') &= FV(p) \cup FV(p') \\
FV(p \setminus a) &= FV(p) \\
FV(p[S]) &= FV(p) \\
FV(x) &= \{x\}.
\end{aligned}$$

A variable which is not free i.e. does not belong to $FV(p)$ is said to be bound in p. An expression p is closed if $FV(p)=\emptyset$. Closed expressions are referred to as processes. The set of closed expressions is denoted by CPr. We shall also talk about processes with at most one free variable. We denote this set by $Pr[x]$. Elements of $Pr[x]$ may be thought of as functions from processes to processes i.e. belonging to $CPr \to CPr$.

To allow processes received in communication to be used we need a way of substituting the received processes for bound variables. We shall use the definition of name substitution to avoid unintentional binding of free names when processes are substituted.

Definition 2.4 The substitution $p[q/x]$ is defined structurally on p:

$$
\begin{aligned}
nil[q/x] &\equiv nil \\
(a?\,y.p)[q/x] &\equiv \begin{cases} a?\,y.(p[q/x]) & \text{if } y \neq x \text{ and } y \notin FV(q) \\ a?\,z.((p[z/y])[q/x]) & \text{otherwise} \\ & z \notin FV(p) \cup FV(q) \cup \{x, y\} \end{cases} \\
(a!\,p'.p)[q/x] &\equiv a!(p'[q/x]).(p[q/x]) \\
(\tau.p)[q/x] &\equiv \tau.(p[q/x]) \\
(p+p')[q/x] &\equiv (p[q/x])+(p'[q/x]) \\
(p|p')[q/x] &\equiv (p[q/x])|(p'[q/x]) \\
(p\backslash a)[q/x] &\equiv (((\{d/a\}\,p)[q/x])\backslash d \quad \text{for some } d \notin (fn(p\backslash a) \cup fn(q)) \\
(p[S])[q/x] &\equiv (p[q/x])[S] \\
y[q/x] &\equiv \begin{cases} q & \text{if } x=y \\ y & \text{otherwise.} \end{cases}
\end{aligned}
$$

The only difference between the above definition of substitution and the one given for "dynamic" CHOCS in [Tho89] is in the clause for restriction. In the above definition we ensure that we do not restrict names in q. We shall consider $p \equiv q$ if p and q only differ up to change of bound names and/or bound variables.

Here are a few useful properties of substitution:

Proposition 2.5

1. *If* $x \neq y$ *then* $p[p'/x][p''/y] \equiv p[p''/y][p'[p''/y]/x]$.
2. $p[p'/x] \equiv p$ *if* $x \notin FV(p)$.

Proof: Easily established by structural induction on p. □

With the above machinery in hand we may now give the operational semantics for Plain CHOCS. The operational semantics is given in terms of a labelled transition system in the style of [Plo81].

Definition 2.6 The transition relation $\to$ is a family of binary labelled relations $\xrightarrow{\Gamma}$ between elements of CPr (processes) and $Pr[x]$ of the form $p \xrightarrow{\Gamma} p'$.

The action Γ may have one of the following forms: $a?\,x$, $a!_B p$, τ, where $a \in$ Names, $B \subseteq$ Names, $x \in V$ and $p \in CPr$. Let the bound names bn of an action be defined as:

$$
bn(\Gamma) = \begin{cases} B & \text{if } \Gamma = a!_B p' \\ \emptyset & \text{otherwise.} \end{cases}
$$

Table 1. Operational semantics for Plain CHOCS. The choice, par, com-close rules have symmetric counterparts

$$\text{input } a?x.p \xrightarrow{a?x} p$$

$$\text{output } a!p'.p \xrightarrow{a!_\emptyset p'} p$$

$$\text{tau } \tau.p \xrightarrow{\tau} p$$

$$\text{choice } \frac{p \xrightarrow{\Gamma} p''}{p+q \xrightarrow{\Gamma} p''}$$

$$\text{par } \frac{p \xrightarrow{\Gamma} p''}{p|q \xrightarrow{\Gamma} p''|q}, \; bn(\Gamma) \cap fn(q) = \emptyset$$

$$\text{ren } \frac{p \xrightarrow{a?x} p''}{p[S] \xrightarrow{S(a)?x} p''[S]}$$

$$\frac{p \xrightarrow{a!_B p'} p''}{p[S] \xrightarrow{S(a)!_B p'} p''[S]}, \; B \cap (Dom(S) \cup Im(S)) = \emptyset$$

$$\frac{p \xrightarrow{\tau} p''}{p[S] \xrightarrow{\tau} p''[S]}$$

$$\text{res } \frac{p \xrightarrow{a?x} p''}{p\backslash b \xrightarrow{a?x} p''\backslash b}, \; a \neq b$$

$$\frac{p \xrightarrow{a!_B p'} p''}{p\backslash b \xrightarrow{a!_B p'} p''\backslash b}, \; a \neq b, \; b \notin fn(p')$$

$$\frac{p \xrightarrow{\tau} p''}{p\backslash b \xrightarrow{\tau} p''\backslash b}$$

$$\text{open } \frac{p \xrightarrow{a!_B p'} p''}{p\backslash c \xrightarrow{a!_{B \cup \{d\}}([d/c]p')} \{d/c\}p''}, \; a \neq c, \; d \notin fn(p\backslash c), \; c \in fn(p')$$

$$\text{com-close } \frac{p \xrightarrow{a?x} p' \quad q \xrightarrow{a!_B q'} q''}{p|q \xrightarrow{\tau} (p'[q'/x]|q'')\backslash B}, \; B \cap fn(p') = \emptyset$$

$$\text{non-struct } \frac{p \xrightarrow{a!_B p'} p''}{p \xrightarrow{a!_{B'} p'} p''}, \; B \cap (fn(p') \cup fn(p'')) = B' \cap (fn(p') \cup fn(p''))$$

In the definition of the semantics of Plain CHOCS it is convenient to write $p\backslash B$ where $B \subseteq$ Names is a finite set: $p\backslash B$ is shorthand for $p\backslash b_1 \ldots \backslash b_n$ where $B = \{b_1, \ldots, b_n\}$ and p if $B = \emptyset$. We let $\rightarrow$ be the smallest transition relation closed under the rules of Table 1.

The structure of this transition system is tailored to cater for the behaviour we have in mind for systems like those described by (3) and (4) in the introduction to this paper, but it also carries some philosophy of its own. The three kinds of actions yield the following types of transitions or observations:

Input action $p \xrightarrow{a?x} p'$, this kind of transitions may be interpreted as, "the process p is capable of receiving on channel a". We only allow transitions of this kind where $p \in CPr$ and $p' \in Pr[x]$. We want to model input transitions in such a way that no further observations are possible until a value is supplied. The reason for this is both technical and philosophical. Technically it ensures that we do not "rewrite" to open terms which, without care, could lead to confusion of free variables e.g.: $a?x.x|b?x.x \xrightarrow{b?x} a?x.x|x \xrightarrow{a?x} x|x$. Philosophically it follows a point of view of only observing systems by atomic observations or combinations of atomic observations. The input observations consist of observing that input on channel a is possible and the systems readiness to accept a value. To make further observations about this process we have to supply a value say $q \in CPr$ and observe the system $p'[q/x]$ with this value. A more suggestive notation would perhaps be $p \xrightarrow{a?} \lambda x.p'$, but it is not essential in the present calculus since x only acts as a place holder. We have chosen the notation $p \xrightarrow{a?x} p'$ since p' is describable in the Plain CHOCS syntax. We could extend the above transition system to open expressions. To avoid confusion of variables introduced by the input-rule we would have to ascribe the par-rule by the additional constraint $FV(p'') \cap FV(q) = \emptyset$. We have not done this since the theory of equivalence will be defined in terms of closed expressions and extended to open expressions using the definition for closed expressions.

Output actions (with scope extrusion) $p \xrightarrow{a!_B p'} p''$. We refer to p' as the emitted process and p as the emitting process or rather p'' since this is the state of the system after emitting p'. If $B = \emptyset$ this kind of transitions may be interpreted as, "the process p can output the process p' on channel a and in doing so become p''". To observe this action we observe that output on channel a is possible, to make further observations we have to observe both the value p' and the resulting state p''. If p' and p'' share some private channels these will be in the set B and a scope extrusion is necessary. We observe this by the combined observation as for normal output actions together with the additional observation of the scope extrusion. A more suggestive notation for output transitions might be $p \xrightarrow{a!} (B, p', p'')$.

Silent actions $p \xrightarrow{\tau} p'$, this kind of transitions may be interpreted as, "the process p can do an internal or silent move and in doing so become the process p'". Silent actions arise from communications between two processes. Since communications are the only computations in our calculus these are in a sense the real computations of the system. τ-transitions may of course arise from processes of the form $\tau.p$ as well.

The input, output and com-close rules form the basis for inferring a communication between two agents. In the rules of Table 1 all transitions of the form $p \xrightarrow{a?x} p'$ have the property that $p \in CPr$ and $p' \in Pr[x]$ and all transitions of the form $p \xrightarrow{a!_B p'} p''$ have the property $p, p', p'' \in CPr$, therefore $p'[q'/x] \in CPr$ in the com-close rule. This set of rules gives an operational description where input is modelled as a function and communication acts as a generalised application. This is very different from the nature of inferring communication in "dy-

namic" CHOCS (or in CCS with value passing [Mil80]). In "dynamic" CHOCS we have the following three rules as the basis for inferring communication:

$$a?x.p \xrightarrow{a?p'} p[p'/x] \quad a!p'.p \xrightarrow{a!p'} p \quad \frac{p \xrightarrow{a?p'} p'' \; q \xrightarrow{a!p'} q''}{p|q \xrightarrow{\tau} p''|q''}.$$

Note that in these rules the transition relation is always between elements of *CPr*. One way of interpreting the above rules is to say that the process with input prefix knows all the possible values it can receive. What it does is to offer a (an infinite) choice between all the possible new states and when the communication takes place it is only a signal from the output process to the input process telling which value to use (choose). This viewpoint is further strengthened by the (elegant) way of encoding value passing in SCCS as described in [Mil83]. In [MilParWal89] a scheme similar to the above for inferring communication has been termed early instantiation, referring to the fact that the instantiation of the free variable takes place in the axiom for input prefix as opposed to the scheme used in Table 1. The scheme we are using has been termed late instantiation, though in their case there is a difference since processes are allowed to offer new transitions after an input transition. This calls for some machinery to ensure that free variables are not confused. We have chosen the late instantiation scheme with the restriction that $p' \in Pr[x]$ in $p \xrightarrow{a?x} p'$ for the reasons given above; late instantiation also seems necessary for the scope opening and closing rules for the restriction operator. The rules concerning the restriction operator have several alternatives, e.g. in "dynamic" CHOCS this operator does not bind names in the process emitted but only in the emitting process as the examples in the introduction show. Another possibility would be the following rule

$$\frac{p \xrightarrow{a!p'} p''}{p\backslash b \xrightarrow{a!(p'\backslash b)} p''\backslash b}, \qquad a \neq b.$$

This approach would ensure that bound names would be bound both in the emitted process and in the emitting process, but it is too restrictive since they can not use the local channel to communicate with one another since the $\backslash b$ encapsulates the process. To elaborate on this we follow the ideas of [EngNie86; MilParWal89] and adopt a restriction rule with the side condition that p' can escape the restriction only if $b \notin fn(p')$. In [Tho89a; Tho90] and previous versions of this paper the following rule was adopted:

$$\frac{p \xrightarrow{a!_B p'} p''}{p\backslash b \xrightarrow{a!_B p'\backslash b} p''\backslash b}, \qquad a \neq b,\ b \notin (fn(p') \cap fn(p'')).$$

It was pointed out to me by Crasemann that this rule will imply that Proposition 3.14 does not hold. The reason is that although the emitted process does not share the restricted name with the emitting process, the emitted process may be copied upon reception and each copy may share the restricted name. With the above rule each copy will be encapsulated by the restriction and the sharing of the name is therefore broken. Originally this rule was intended to allow

processes embraced by a restriction to emit processes with the restriction if they are not sharing the restricted name. This was intended to save doing a scope extrusion in this situation. However, the non-struct rule will suffice in this case since if the restricted name does not occur free in the emitted process it may be eliminated using the non-struct rule. In general we may use the non-struct rule for adding or deleting names in the set of bound names in scope extrusion if these names are not shared by the emitting and emitted processes.

We also introduce two new rules; open and com-close. The opening rule signals that in the emitted process there are some bound names, names which are shared with the emitting process. The com-close rule ensures that exported restrictions are reintroduced upon reception. The condition on this rule ensures that we do not bind free names in the receiving process. When $B=\emptyset$ this rule is just a communication rule.

We conclude this section by listing a few useful properties of the transition system defined in Table 1.

Proposition 2.7

1. *If* $p \xrightarrow{a!_B p'} p''$ *and* $b \notin fn(p) \cup B$ *then* $p \xrightarrow{a!_{((B\setminus\{c\})\cup\{b\})}\{b/c\}p'} \{b/c\}p''$ *for any* $c \in B$.
2. *If* $p \xrightarrow{a!_B p'} p''$ *then* $p \xrightarrow{a!_{B'} p'} p''$ *for some* B' *with* $B \cap (fn(p') \cup fn(p'')) = B' \cap (fn(p') \cup fn(p''))$ *and* $B' \subseteq fn(p') \cap fn(p'')$ *and* $B' \cap fn(p) = \emptyset$.
3. *If* $p \xrightarrow{a?x} p'$ *then* $fn(p') \subseteq fn(p)$.
4. *If* $p \xrightarrow{a!_B p'} p''$ *then* $fn(p') \subseteq fn(p) \cup B$ *and* $fn(p'') \subseteq fn(p) \cup B$
5. *If* $p \xrightarrow{\tau} p'$ *then* $fn(p') \subseteq fn(p)$.

Proof. By induction on the length of the inference used to establish the transition and cases of the structure of p. □

3 Bisimulation equivalence and laws

In the previous section we presented the operational semantics for Plain CHOCS in terms of a labelled transition system. The structure of this transition system resembles a merge between the applicative transition systems of [Abr90] and the higher order communication trees used in the semantics for CHOCS in [Tho89]. The transition relation $\rightarrow$ forms the basis for the observations we can make about processes, but it is in itself too shallow to use as a distinguishing equivalence. Instead we use the notion of (bi)simulation [Par81, Mil83] redefined to the kind of observations the transition allows:

Definition 3.1 An applicative higher order simulation R is a binary relation on CPr such that whenever pRq and $a \in Names$ then:

(i) Whenever $p \xrightarrow{a?x} p'$, then $q \xrightarrow{a?y} q'$ for some q', y and $p'[r/x]\,Rq'[r/y]$ for all $r \in CPr$

(ii) Whenever $p \xrightarrow{a!_B p'} p''$ with $B \cap (fn(p) \cup fn(q)) = \emptyset$, then $q \xrightarrow{a!_B p'} q''$ for some q', q'' with $p' R q'$ and $p'' R q''$

(iii) Whenever $p \xrightarrow{\tau} p'$, then $q \xrightarrow{\tau} q'$ for some q' with $p' R q'$

A relation R is an applicative higher order bisimulation if both it and its inverse are applicative higher order simulations.

Two processes p and q are said to be bisimulation equivalent iff there exists an applicative higher order bisimulation R containing (p, q). In this case we write $p \dot{\sim} q$.

The first clause of this predicate is essentially the clause for applicative (bi)simulation in the Lazy-λ-Calculus as defined in [Abr90]. It can be interpreted as saying that if p can do an input on channel a and become the function p', then q must match this by being able to input on channel a and become the function q' and for all values (arguments) we can receive on this channel the resulting process together with this value should continue to simulate each other. The second clause with $B=\emptyset$ and the third clause are similar to the clauses of higher order bisimulation defined in [Tho89]. The second clause supports a kind of black box view of the processes being sent. If p can output a process p' on channel a and in doing so become p'', then q should be able to output some q' on channel a and in doing so become q'' and p' and q', as well as p'' and q'', should be equivalent. The second clause with $B \neq \emptyset$ is a generalisation of the clause for scope extrusion in the strong ground bisimulation defined in [MilParWal89]. B is a set of private channels between p' and p''. These channels are exported from their original scope and are intended to become restricted upon reception.

Proposition 3.2 *$\dot{\sim}$ is an equivalence*

Before relating the process constructions of Plain CHOCS to the underlying semantic equivalence $\dot{\sim}$ we present a technical construction called an applicative higher order bisimulation up to restriction. This construction resembles the bisimulation up to $\sim$ presented in [Mil89] and it is an adaptation to the Plain CHOCS setting of the notion of strong ground bisimulation up to restriction presented in [MilParWal89b].

Definition 3.3 An applicative higher order simulation up to restriction R is a binary relation on CPr such that whenever pRq and $a \in Names$ then:

(i) If $b \notin fn(p) \cup fn(q)$ then $\{b/a\}\, pR\, \{b/a\}\, q$

(ii) Whenever $p \xrightarrow{a?x} p'$, then $q \xrightarrow{a?y} q'$ for some q', y and $p'[r/x]\, Rq'[r/y]$ for all $r \in CPr$

(iii) Whenever $p \xrightarrow{a!_B p'} p''$ with $B \cap (fn(p) \cup fn(q)) = \emptyset$, then $q \xrightarrow{a!_B q'} q''$ for some q', q'' with $p' Rq'$ and $p'' Rq''$

(iv) Whenever $p \xrightarrow{\tau} p'$, then $q \xrightarrow{\tau} q'$ for some q' and either $p' Rq'$ or for some p'', q'' and b: $p' \equiv p'' \backslash b$, $q' \equiv q'' \backslash b$ and $p'' Rq''$

A relation R is an applicative higher order bisimulation up to restriction if both it and its inverse are applicative higher order simulations up to restriction.

Lemma 3.4 *If R is an applicative higher order bisimulation up to restriction then $R \subseteq \dot{\sim}$.*

Proof. We show that the relation $R^{\backslash} = \bigcup_{n \in \omega} R_n$ where

$$R_0 = R$$

$$R_{n+1} = \{(p \backslash b, q \backslash b) : (p, q) \in R_n,\ b \in \text{Names}\}$$

is an applicative higher order bisimulation.

First we show by induction on n that if pR_nq and $c \notin fn(p) \cup fn(q)$ then $\{c/a\}\, pR_n \{c/a\}\, q$. For $n=0$ this is immediate from the definition of applicative higher order bisimulation up to restriction. Suppose $n>0$ and $p\backslash bR_nq\backslash b$ where $pR_{n-1}q$ and $c \notin fn(p\backslash b) \cup fn(q\backslash b)$. If $a=b$ then $\{c/a\}(p\backslash b) \equiv p\backslash bR\backslash q\backslash b \equiv \{c/a\}(q\backslash b)$. If $a \neq b$ then $\{c/a\}(p\backslash b) \equiv (\{c/a\}(\{b_1/b\}\, p))\backslash b_1 R\backslash (\{c/a\}(b_1/b\}\, q))\backslash b_1 \equiv \{c/a\}(q\backslash b)$. Next we show by induction on n that if pR_nq then

(i) Whenever $p \xrightarrow{a?x} p'$, then $q \xrightarrow{a?y} q'$ for some q', y and $p'[r/x]\, R\backslash q'[r/y]$ for all $r \in CPr$

(ii) Whenever $p \xrightarrow{a!_B p'} p''$ with $B \cap (fn(p) \cup fn(q)) = \emptyset$, then $q \xrightarrow{a!_B q'} q''$ for some q', q'' with $p' R\backslash q'$ and $p'' R\backslash q''$

(iii) Whenever $p \xrightarrow{\tau} p'$, then $q \xrightarrow{\tau} q'$ for some q' and $p' R\backslash q'$

The details can be found in Appendix A. □

Lemma 3.5 *If $p \dot{\sim} q$ and $b \notin fn(p) \cup fn(q)$ then $\{b/a\}\, p \dot{\sim} \{b/a\}\, q$.*

Proof. An easy corollary of Lemma 3.4 and Definition 3.3. □

Let $\bar{x}=(x_1, \ldots, x_n)$ be a vector of variables of length n and $x_i \neq x_j$ if $i \neq j$. We also consider $\bar{x}$ as a set of variables $\{x_1, \ldots, x_n\}$ and we write $\bar{x} \subseteq FV(p)$ which means that the set $\bar{x}$ is a subset of $FV(p)$. Let $p[\bar{q}/\bar{x}]$ mean $(\ldots(p[q_1/x_1])\ldots)[q_n/x_n]$. We only consider substitutions of compatible vectors, i.e. of vectors of the same length. Let $\bar{q}_1 \dot{\sim} \bar{q}_2$ mean $q_{1_j} \dot{\sim} q_{2_j}$ for all $q_{i_j} \in \bar{q}_i$, $i \in 1, 2$ and let $\bar{q}_i \in CPr$ mean $q_{i_j} \in CPr$ for all $q_{i_j} \in \bar{q}_i$.

Proposition 3.6 *$\dot{\sim}$ is a congruence relation on processes (closed expressions).*

1. $p[\bar{q}_1/\bar{x}] \dot{\sim} p[\bar{q}_2/\bar{x}]$ *if* $\bar{q}_1 \dot{\sim} \bar{q}_2$ *and* $\bar{x} \subseteq FV(p)$
2. $a?x.p \dot{\sim} a?x.q$ *if* $p[r/x] \dot{\sim} q[r/x]$ *for all* r
3. $a!p'.p \dot{\sim} a!q'.q$ *if* $p \dot{\sim} q$ *and* $p' \dot{\sim} q'$
4. $\tau.p \dot{\sim} \tau.q$ *if* $p \dot{\sim} q$
5. $p+p' \dot{\sim} q+q'$ *if* $p \dot{\sim} q$ *and* $p' \dot{\sim} q'$
6. $p|p' \dot{\sim} q|q'$ *if* $p \dot{\sim} q$ *and* $p' \dot{\sim} q'$
7. $p\backslash a \dot{\sim} q\backslash a$ *if* $p \dot{\sim} q$
8. $p[S] \dot{\sim} q[S]$ *if* $p \dot{\sim} q$.

The proof of this proposition is quite involved. The reason for this is that we can not prove the congruence properties for Plain CHOCS using the "standard" process calculus technique; i.e. prove that for each operator op in the process language the relation $R_{op}=\{op(\bar{p}_1), op(\bar{p}_2): \bar{p}_1 \sim \bar{p}_2\}$ is a bisimulation and then prove the substitution property (i.e. that if $\bar{q}_1 \sim \bar{q}_2$ then $p[\bar{q}_1/\bar{x}] \sim p[\bar{q}_2/\bar{x}]$) by structural induction on p. This approach fails for Plain CHOCS in the case of parallel composition since we need to know the substitution property to prove that the relation $R_|$ is a higher order bisimulation and we thus end up with a circular argument. This may at first seem surprising, but the "functional" nature of Plain CHOCS may indicate that this property should be hard to prove: e.g. Abramsky has to give quite an argument to prove congruence properties of the Lazy-λ-Calculus in [Abr90].

Proof. 1. We prove this by showing that the relation ACR^*, the reflexive and transitive closure of ACR, where

$$ACR = \{(p[\bar{q}_1/\bar{x}], p[\bar{q}_2/\bar{x}]) : p \in Pr \;\&\; \bar{x} \subseteq FV(p) \;\&\; \bar{q}_1 \dot{\sim} \bar{q}_2 \;\&\; \bar{q}_i \in CPr\},$$

is an applicative higher order bisimulation up to restriction.

Note if $q_1 \dot{\sim} q_2$ then $(x[q_1/x], x[q_2/x]) \in ACR^*$ and we write $(q_1, q_2) \in ACR^*$.

We only show that ACR^* is an applicative higher order simulation up to restriction, symmetry of ACR^* then yields the results. To see that ACR^* is an applicative higher order simulation up to restriction we show that if $(p_1, p_2) \in ACR$ then $p_i \equiv p[\bar{q}_i/\bar{x}]$ and:

(i) If $b \notin fn(p[\bar{q}_1/\bar{x}]) \cup fn(p[\bar{q}_2/\bar{x}])$ then $\{b/a\}(p[\bar{q}_1/\bar{x}])\, ACR^*\, \{b/a\}(p[\bar{q}_2/\bar{x}])$

(ii) Whenever $p[\bar{q}_1/\bar{x}] \xrightarrow{a?x} p'$, then $p[\bar{q}_2/\bar{x}] \xrightarrow{a?y} q'$ for some q', y and $p'[r/x]\, ACR^*\, q'[r/y]$ for all $r \in CPr$

(iii) Whenever $p[\bar{q}_1/\bar{x}] \xrightarrow{a!_B p'} p''$ with $B \cap (fn(p) \cup fn(q)) = \emptyset$, then $p[\bar{q}_2/\bar{x}] \xrightarrow{a!_B q'} q''$ for some q', q'' with $p'\, ACR^*\, q'$ and $p''\, ACR^*\, q''$

(iv) Whenever $p[\bar{q}_1/\bar{x}] \xrightarrow{\tau} p'$, then $p[\bar{q}_2/\bar{x}] \xrightarrow{\tau} q'$ for some q' and either $p'\, ACR^*\, q'$ or for some p'', q'' and b: $p' \equiv p'' \backslash b$, $q' \equiv q'' \backslash b$ and $p''\, ACR^*\, q''$

If $(p, q) \in ACR^*$ then there is a sequence $p_1 \ldots p_n$ such that $(p, p_1) \in ACR$, $(p_i, p_{i+1}) \in ACR$ for $1 \leqq i < n$ and $(p_n, q) \in ACR$. The result then follows by induction on the length of the transitive sequence $p_1 \ldots p_n$ of ACR^*.

First (i) is easily proved by structural induction on p using Lemma 3.5 in the case $p \equiv y$.

Next we show (ii)–(iv) simultaneously. We proceed by induction on the length of the inference used to establish the transitions of $p[\bar{q}_1/\bar{x}]$ and cases of the structure of p. We only need to consider transitions inferred by use of the structural rules since we may transform any derivation of a transitions into an equivalent one where we use the non-struct-rule exactly once after each application of a structural rule. (The full details are given in Appendix A).

2. This is proved by showing that the relation $R_1 = R \cup \dot{\sim}$, where:

$$R = \{(a?x.p, a?x.q) : FV(p) = FV(q) \subseteq \{x\} \;\&\; \forall r \in CPr.\, p[r/x] \dot{\sim} q[r/x]\}$$

is an applicative higher order bisimulation. Note that the relation R_1 consists of two parts; one part covers the structure we are interested in and the second component is a kind of closure to cover the processes sent and received. The second component is necessary since the processes sent and received do not necessarily have the structure of the first part.

That the above relation is indeed an applicative higher order bisimulation is easily established. (The full details are given in Appendix A).

3. follows from $((a!x.y)[(p, p')/(x, y)], (a!x.y)[(q, q')/(x, y)]) \in ACR$ if $p \dot{\sim} q$ and $p' \dot{\sim} q'$ and $x \neq y$.
4. follows from $((\tau.x)[p/x], (\tau.x)[q/x]) \in ACR$ if $p \dot{\sim} q$.
5. follows from $((x+y)[(p, p')/(x, y)], (x+y)[(q, q')/(x, y)]) \in ACR$ if $p \dot{\sim} q$ and $p' \dot{\sim} q'$ and $x \neq y$.

6. follows from $((x|y)[(p, p')/(x, y)], (x|y)[q, q')/(x, y)]) \in ACR$ if $p \dot{\sim} q$ and $p' \dot{\sim} q'$ and $x \neq y$.
7. follows from $(x[p/x], x[q/x]) \in ACR$ if $p \dot{\sim} q$ and the fact that ACR^* is an applicative bisimulation up to restriction.
8. follows from $((x[S])[p/x], (x[S])[q/x]) \in ACR$ if $p \dot{\sim} q$. □

The congruence result easily generalises to open terms by standard techniques by defining $p \dot{\sim} q$ iff $\forall r_1 \ldots r_n . p[r_1 \ldots r_n/x_1 \ldots x_n] \dot{\sim} q[r_1 \ldots r_n/x_1 \ldots x_n]$ where $x_1 \ldots x_n$ are the free variables of p and q and $r_1 \ldots r_n$ are closed terms. This is equivalent to the following definition: $p \dot{\sim} q$ iff $a?x_1 \ldots a?x_n . p \dot{\sim} a?x_1 \ldots a?x_n . q$.

From establishing bisimulations between Plain CHOCS processes we may show that two processes are equivalent, but this technique often involves quite an amount of ingenuity in the construction of a bisimulation relation. Instead we may prefer the more well known techniques of algebraic reasoning. A lot of interesting properties of Plain CHOCS may be inferred from equational reasoning. This kind of reasoning may of course be combined with establishing bisimulations directly.

The first set of laws concerns the choice operator and shows that *nil* is a zero for + and that + is idempotent, commutative and associative.

Proposition 3.7

$$p + nil \dot{\sim} p$$
$$p + p \dot{\sim} p$$
$$p + p' \dot{\sim} p' + p$$
$$p + (p' + p'') \dot{\sim} (p + p') + p''.$$

Proof. This follows from showing that the following relations are higher order applicative bisimulations:

$$R_1 = \{(p + nil, p)\} \cup Id$$
$$R_2 = \{(p + p, p)\} \cup Id$$
$$R_3 = \{(p + p', p' + p)\} \cup Id$$
$$R_4 = \{(p + (p' + p''), (p + p') + p'')\} \cup Id.$$

To see this observe that for $(r, q) \in R_i$, $i \in \{1, 2, 3, 4\}$ we have either $(r, q) \in Id$ and if $r \xrightarrow{\Gamma} r'$ then $r = q \xrightarrow{\Gamma} q' = r'$ and we have a matching move or (r, q) belongs to the first part of R_i and if $r \xrightarrow{\Gamma} r'$ then this must have been inferred by the rules for choice. Then also $q \xrightarrow{\Gamma} r'$ which is a matching move. □

We now proceed with some properties of the restriction operator and its interplay with the other operators. To smooth the presentation of equations we introduce a fourth (derived) prefix; an output prefix with scope extrusion: $a!_B p'$. Thus $a!_B p' . p$ is shorthand notation for $(a! p' . p) \backslash B$ with the obvious operational semantics: $a!_B p' . p \xrightarrow{a!_B p'} p$. We shall always assume that $B \subseteq fn(p') \cap fn(p)$.

Proposition 3.8

$$
\begin{aligned}
p\backslash a &\dot{\sim} p \quad \textit{if } a\notin fn(p)\\
p\backslash a\backslash b &\dot{\sim} p\backslash b\backslash a\\
(p+p')\backslash a &\dot{\sim} p\backslash a+p'\backslash a\\
(a?x.p)\backslash b &\dot{\sim} a?x.(p\backslash b) \quad \textit{if } a\neq b\\
(a?x.p)\backslash b &\dot{\sim} nil \quad \textit{if } a=b\\
(\tau.p)\backslash b &\dot{\sim} \tau.(p\backslash b)\\
(a!_B p'.p)\backslash b &\dot{\sim} a!_B p'.(p\backslash b) \quad \textit{if } a\neq b \textit{ and } b\notin fn(p')\\
(a!_B p'.p)\backslash b &\dot{\sim} a!_{B\cup\{b\}} p'.p \quad \textit{if } a\neq b \textit{ and } b\in fn(p')\\
(a!_B p'.p)\backslash b &\dot{\sim} nil \quad \textit{if } a=b.
\end{aligned}
$$

Proof. The proposition follows from showing that the following relations are applicative higher order bisimulations:

$$
\begin{aligned}
R_1 &= \{(p\backslash a, p)\colon p\in CPr, a\notin fn(p)\}\\
R_2 &= \{(p\backslash a\backslash b, p\backslash b\backslash a)\colon p\in CPr\}\cup Id\\
R_3 &= \{((p+p')\backslash a, p\backslash a+p'\backslash a)\colon p_i\in CPr\}\cup Id\\
R_4 &= \{((a?x.p)\backslash b, a?x.(p\backslash b))\colon a?x.p\in CPr, a\neq b\}\cup Id\\
R_5 &= \{((a?x.p)\backslash b, nil)\colon a?x.p\in CPr, a=b\}\\
R_6 &= \{((\tau.p)\backslash b, \tau.(p\backslash b))\colon p\in CPr\}\cup Id\\
R_7 &= \{((a!_B p'.p)\backslash b, a!_B p'.(p\backslash b))\colon p, p'\in CPr, a\neq b, b\notin fn(p')\}\cup Id\\
R_8 &= \{((a!_B p'.p)\backslash b, a!_{B\cup\{b\}} p'.p)\colon p, p'\in CPr, a\neq b, b\in fn(p')\}\cup Id\\
R_9 &= \{((a!_B p'.p)\backslash b, nil)\colon p, p'\in CPr, a=b\}.
\end{aligned}
$$

We must include Id in relation R_2 to R_4 and R_6 to R_8. For relation R_3, R_4 and R_6 to R_8 this is clear since if $(p, q)\in R_i$, $i\in\{3, 4, 6, 7, 8\}$ then after the first transition $p\xrightarrow{\Gamma} p'$ and a first matching transition $q\xrightarrow{\Gamma'} q'$ we will have $(p', q')\in Id$. For R_2 it is necessary to include Id since the restrictions may disappear due to applications of the open-rule. □

The following theorem states an expected property of restriction, namely that the restricted name may be α-converted without affecting the behaviour of the process involved.

Theorem 3.9 $p\backslash a\dot{\sim}(\{b/a\}\, p)\backslash b$ *if* $b\notin fn(p)$

Proof. This theorem follows by showing that the relation

$$R = \{(p\backslash a, (\{b/a\}\, p)\backslash b)\colon p\in CPr, b\notin fn(p)\}\cup Id$$

is an applicative higher order bisimulation.

The *Id* component of this relation is necessary in case of scope extrusion due to an application of the open-rule in which case the restrictions will disappear and a respectively b will be substituted with a new name $c \notin fn(p) \cup \{b\}$. The matching moves are easily established by appealing to Proposition 2.7. □

Before presenting any additional laws we need to introduce a concept related to the concept of an applicative higher order bisimulation up to restriction. The new concept is called an applicative higher order bisimulation up to $\lesssim$ and allows a relaxation of applicative higher order bisimulation in the sense that the relation only has to satisfy the applicative higher order bisimulation properties up to the closure property of $\lesssim$. This is an adaptation of the notion of bisimulation up to $\sim$ [Mil89] to the Plain CHOCS setting:

Definition 3.10 An applicative higher order simulation up to $\lesssim$ is a binary relation R on CPr such that whenever pRq and $a \in$ Names then:

(i) Whenever $p \xrightarrow{a?x} p'$, then $q \xrightarrow{a?y} q'$ for some q', y and $p'[r/x] \lesssim R \lesssim q'[r/y]$ for all $r \in CPr$

(ii) Whenever $p \xrightarrow{a!_B p'} p''$ with $B \cap (fn(p) \cup fn(q)) = \emptyset$, then $q \xrightarrow{a!_B q'} q''$ for some q', q'' with $p' \lesssim R \lesssim q'$ and $p'' \lesssim R \lesssim q''$

(iii) Whenever $p \xrightarrow{\tau} p'$, then $q \xrightarrow{\tau} q'$ for some q' with $p' \lesssim R \lesssim q'$.

A relation R is an applicative higher order bisimulation up to $\lesssim$ if both it and its inverse are applicative higher order simulations up to $\lesssim$.

Lemma 3.11 *If R is an applicative higher order bisimulation up to $\lesssim$ then $R \subseteq \lesssim$.*

Proof. Follows by arguments very similar to the arguments given for Lemma 3.4. □

The following definition is an adaptation to the Plain CHOCS setting of the notion of strong ground bisimulation up to $\lesssim$ and restriction from [MilPar-Wal89b]:

Definition 3.12 An applicative higher order simulation up to $\lesssim$ and restriction R is a binary relation on CPr such that whenever pRq and $a \in$ Names then:

(i) If $b \notin fn(p) \cup fn(q)$ then $\{b/a\}\, p \lesssim R \lesssim \{b/a\}\, q$

(ii) Whenever $p \xrightarrow{a?x} p'$, then $q \xrightarrow{a?y} q'$ for some q', y and $p'[r/x] \lesssim R \lesssim q'[r/y]$ for all $r \in CPr$

(iii) Whenever $p \xrightarrow{a!_B p'} p''$ with $B \cap (fn(p) \cup fn(q)) = \emptyset$, then $q \xrightarrow{a!_B q'} q''$ for some q', q'' with $p' \lesssim R \lesssim q'$ and $p'' \lesssim R \lesssim q''$

(iv) Whenever $p \xrightarrow{\tau} p'$, then $q \xrightarrow{\tau} q'$ for some q' and either $p' \lesssim R \lesssim q'$ or for some p'', q'' and b: $p' \lesssim p'' \backslash b$, $q' \lesssim q'' \backslash b$ and $p'' R q''$.

A relation R is an application higher order bisimulation up to $\lesssim$ and restriction if both it and its inverse are applicative higher order simulations up to $\lesssim$ and restriction.

Lemma 3.13 *If R is an applicative higher order bisimulation up to $\dot{\sim}$ and restriction then $R \subseteq \dot{\sim}$.*

Proof. Let $R^{\backslash \dot{\sim}} = \bigcup_{n \in \omega} R_n$ where

$$R_0 = \dot{\sim} R \dot{\sim}$$
$$R_{n+1} = \dot{\sim}\{(p\backslash a, q\backslash a) : (p, q) \in R_n, a \in \text{Names}\}\dot{\sim}$$

The argument that $R^{\backslash \dot{\sim}}$ is an applicative higher order bisimulation follows the same pattern as the proof of Lemma 3.4. □

With this machinery in hand we may now prove the following interplay between the restriction operator and parallel composition:

Proposition 3.14 $p_1\backslash a \mid p_2 \dot{\sim} (p_1 \mid p_2)\backslash a$ *if* $a \notin fn(p_2)$

Proof. This proposition is proved by showing that the relation

$$R = \{(p_1\backslash a \mid p_2, (p_1 \mid p_2)\backslash a) : p_i \in CPr, a \notin fn(p_2)\} \cup Id$$

is an applicative higher order bisimulation up to $\dot{\sim}$ and restriction. (The full proof is presented in Appendix A). □

The next set of laws shows some expected properties of the parallel operator. It would perhaps have been more natural to present these laws before the laws of restriction and its interplay with other operators, but to prove the law of associativity for the parallel operator we need some of the above properties.

Proposition 3.15

$$p \mid nil \dot{\sim} p$$
$$p_1 \mid p_2 \dot{\sim} p_2 \mid p_1$$
$$p_1 \mid (p_2 \mid p_3) \dot{\sim} (p_1 \mid p_2) \mid p_3 .$$

Proof. This proposition is proved by showing that the first two of the following relations are applicative higher order bisimulations and that the last relation is an applicative higher order bisimulation up to $\dot{\sim}$ and restriction:

$$R_1 = \{(p \mid nil, p) : p \in CPr\} \cup Id$$
$$R_2 = \{(p_1 \mid p_2, p_2 \mid p_1) : p_i \in CPr\} \cup Id$$
$$R_3 = \{(p_1 \mid (p_2 \mid p_3), (p_1 \mid p_2) \mid p_3) : p_i \in CPr\} \cup Id .$$

The Id component in each of the above relations is necessary to cover the cases when processes are communicated since these processes might not have the structure of the first part of the relation. (The full proof is presented in Appendix A). □

Using the above properties we may now present a law of interplay between parallel composition and restriction which will look more familiar to readers with knowledge of CCS.

Theorem 3.16 $(p_1|p_2)\backslash a \dot{\sim} p_1\backslash a|p_2\backslash a$ *if* $a\notin fn(p_1)\cap fn(p_2)$

Proof. If $a\notin fn(p_1)\cap fn(p_2)$ then a can not be a free name in both p_1 and p_2. Suppose $a\notin fn(p_2)$. Then by Proposition 3.14 and Proposition 3.8 we have $(p_1|p_2)\backslash a \dot{\sim} p_1\backslash a|p_2 \dot{\sim} p_1\backslash a|p_2\backslash a$. The other case where $a\notin fn(p_1)$ follows by a similar argument after commuting p_1 and p_2 using Proposition 3.15. □

We now present some expected properties of renaming:

Proposition 3.17

$$
\begin{aligned}
nil[S] &\dot{\sim} nil\\
p[S] &\dot{\sim} p[S][S]\\
p[S]\backslash b &\dot{\sim} p\backslash b[S] \quad \text{if } b\notin Dom(S)\cup Im(S)\\
(p_1+p_2)[S] &\dot{\sim} p_1[S]+p_2[S]\\
(p_1|p_2)[S] &\dot{\sim} p_1[S]|p_2[S] \quad \text{if } Dom(S)\cap(fn(p_1)\cup fn(p_2))=\emptyset\\
(a?x.p)[S] &\dot{\sim} S(a)?x.(p[S])\\
(\tau.p)[S] &\dot{\sim} \tau.(p[S])\\
(a!_B p'.p)[S] &\dot{\sim} S(a)!_B p'.(p[S]) \quad \text{if } B\cap(Dom(S)\cup Im(S))=\emptyset
\end{aligned}
$$

Proof. The proposition follows from showing that the following relations are applicative higher order bisimulations:

$$
\begin{aligned}
R_1 &= \{(nil[S], nil)\}\\
R_2 &= \{(p[S], p[S][S]): p\in CPr\}\cup Id\\
R_3 &= \{(p[S]\backslash b, p\backslash b[S]): p\in CPr, b\notin Dom(S)\cup Im(S)\}\cup Id\\
R_4 &= \{((p_1+p_2)[S], p_1[S]+p_2[S]): p_i\in CPr\}\cup Id\\
R_5 &= \{((p_1|p_2)[S], p_1[S]|p_2[S]): Dom(S)\cap(fn(p_1)\cup fn(p_2))=\emptyset\}\cup Id\\
R_6 &= \{((a?x.p)[S], S(a)?x.(p[S])): a?x.p\in CPr\}\cup Id\\
R_7 &= \{((\tau.p)[S], \tau.(p[S])): p_i\in CPr\}\cup Id\\
R_8 &= \{((a!_B p'.p)[S], S(a)!_B p'.(p[S])):\\
&\qquad p, p'\in CPr, B\cap(Dom(S)\cup Im(S))=\emptyset\}\cup Id.
\end{aligned}
$$

The Id component in the above relations serves to cover processes being sent. In addition the Id component of relation R_3 covers the case when the restriction disappears due to an application of the open-rule. It is relatively straightforward to find matching moves for each relation and we omit the details. (The proof for relation R_5 relies on the fact that $p[S] \dot{\sim} p[S][S]$ and this is easily established since $S=[a\mapsto b]$ and either $a=b$ in which case $p[S] \dot{\sim} p$ or $a\neq b$ in which case the second renaming will have no effect.) □

We have not listed any immediate interplay between (nondeterministic) choice and parallel composition. This is due to the fact that the two operators in general do not commute, but there is a restricted interplay between them:

Proposition 3.18 *Let* $\bar{x}=\{x_1 \ldots x_n\}$, $\bar{y}=\{y_1 \ldots y_n\}$ *and* $\bar{x}\cap\bar{y}\neq\emptyset$ *and* $A_j \cap fn(q)=\emptyset$ *and* $B_l \cap fn(p)=\emptyset$ *then*

$$\begin{aligned}
&\textit{if} \quad p=\Sigma_i a_i?x_i.p_i+\Sigma_j a_j!_{A_j}p'_j.p_j\\
&\textit{and} \quad q=\Sigma_k b_k?y_k.q_k+\Sigma_l b_l!_{B_l}q'_l.q_l\\
&\textit{then} \quad p|q \dot{\sim} \Sigma_i a_i?x_i.(p_i|q)+\Sigma_j a_j!_{A_j}p'_j.(p_j|q)\\
&\qquad\qquad +\Sigma_k b_k?y_k.(p|q_k)+\Sigma_l b_l!_{B_l}q'_l.(p|q_l)\\
&\qquad\qquad +\Sigma_{(i,l)\in\{(i,l):\,a_i=b_l\}}\tau.(p_i[q'_l/x_i]|q_l)\backslash B_l\\
&\qquad\qquad +\Sigma_{(j,k)\in\{(j,k):\,a_j=b_k\}}\tau.(p_j|q_k[p'_j/y_k])\backslash A_j,
\end{aligned}$$

where $\Sigma_i \Gamma_i.p_i$ describes the sum $\Gamma_1.p_1+\ldots+\Gamma_n.p_n$ *when* $n>0$ *and nil if* $n=0$, *knowing this notation is unambiguous because of Proposition* 3.7.

Proof. (Given in Appendix A). □

We can not hope that these equations form a basis for a sound and complete proof system for Plain CHOCS. One reason for this is hinted in the translation given in the next section from Plain CHOCS into Mobile Processes [MilPar-Wal89]. This translation needs parallel composition under the scope of recursion to work. In [Mil83] Milner shows how this combination could be used to simulate a Turing machine. Another reason is that we may encode recursion using the constructs of Plain CHOCS.

Definition 3.19 Let $W_x[\,]$ be the context:

$$a?x.([\,][(x|a!x.nil)[a\mapsto b]\backslash b/x])$$

and let $Y_x[\,]$ be the context:

$$(W_x[\,]|a!(W_x[\,]).nil)[a\mapsto b]\backslash b.$$

To a certain extent this construct resembles the Curry paradoxical combinator $Y[\,]=(\lambda x.[\,](xx))(\lambda x.[\,](xx))$ which is often referred to as the Y combinator in the λ-Calculus.

Note that if $FV(p)\subseteq\{x\}$ then

$$\begin{aligned}
Y_x[p] &\xrightarrow{\tau} (p[(x|a!x.nil)[a\mapsto b]\backslash b/x][W_x[p]/x]|nil)[a\mapsto b]\backslash b\\
&\equiv (p[Y_x[p]/x]|nil)[a\mapsto b]\backslash b \dot{\sim} (p[Y_x[p]/x])[a\mapsto b]\backslash b\\
&\equiv (p[a\mapsto b]\backslash b)[Y_x[p]/x].
\end{aligned}$$

By Proposition 3.8 and Proposition 3.17 we have $(p[a\mapsto b]\backslash b)[Y_x[p]/x] \dot{\sim} p[Y_x[p]/x]$ if $a, b\notin fn(p)$.

Note how $Y_x[\,]$ needs a τ-transition to unwind the "recursion". This resembles the unwinding of recursion in the inference rule of recursion in TCCS [HenNic87]: $\texttt{rec}\,x.p \rightsquigarrow p[\texttt{rec}\,x.p/x]$, where $\rightsquigarrow$ may be read as $\xrightarrow{\tau}$.

As in CCS we may introduce a recursion operator $\mathtt{rec}\,x.p$ with the following operational semantics:

$$\frac{p[\mathtt{rec}\,x.p/x] \xrightarrow{\Gamma} p'}{\mathtt{rec}\,x.p \xrightarrow{\Gamma} p'}.$$

The inference rule basically says that a recursive process has the same derivations as its unfoldings. $\mathtt{rec}\,x.$ is a variable binder and fn, $\{/\}$, FV and $[/]$ have to be extended to cater for the new operator.

Theorem 3.20 $Y_x[p] \mathrel{\dot{\sim}} \mathtt{rec}\,x.\tau.(p[a{\mapsto}b]\backslash b)$.

Proof. For this proof we need the following property of substitution:

$$\text{if } x \neq y \text{ then } p[p'/x][p''/y] \equiv p[p''/y][p'[p''/y]/x]$$

and a simple corollary:

$$\text{if } x \neq y \text{ and } p', p'' \text{ are closed then } p[p'/x][p''/y] \equiv p[p''/y][p'/x]$$

which is easily established by structural induction on p. (They are not corollaries of Proposition 2.5 since we have to take recursive processes into account.)

Then the relation:

$$R = \{(q[\mathtt{rec}\,x.\tau.(p[a{\mapsto}b]\backslash b)/x],\ q[Y_x[p]/x]) \colon FV(q) \subseteq \{x\}\}$$

is an applicative higher order simulation up to $\dot{\sim}$ and restriction. To prove this we show that:

If $c \notin fn(q[\mathtt{rec}\,x.\tau.(p[a{\mapsto}b]\backslash b)/x]) \cup fn(q[Y_x[p]/x])$ then
$\{c/d\}(q[\mathtt{rec}\,x.\tau.(p[a{\mapsto}b]\backslash b)/x]) \mathrel{\dot{\sim}} R \mathrel{\dot{\sim}} \{c/d\}(q[Y_x[p]/x])$.

Whenever $q[\mathtt{rec}\,x.\tau.(p[a{\mapsto}b]\backslash b)/x] \xrightarrow{a?x} p'$, then $q[Y_x[p]/x] \xrightarrow{a?y} q'$ for some q', y and $p'[r/x] \mathrel{\dot{\sim}} R \mathrel{\dot{\sim}} q'[r/y]$ for all $r \in CPr$.

Whenever $q[\mathtt{rec}\,x.\tau.(p[a{\mapsto}b]\backslash b)/x] \xrightarrow{a!_B p'} p''$, then $q[Y_x[p]/x] \xrightarrow{a_B!q'} q''$ for some q', q'' with $B \cap (fn(q[\mathtt{rec}\,x.\tau.(p[a{\mapsto}b]\backslash b)/x]) \cup fn(q[Y_x[p]/x])) = \emptyset$, $p' \mathrel{\dot{\sim}} R \mathrel{\dot{\sim}} q'$ and $p'' \mathrel{\dot{\sim}} R \mathrel{\dot{\sim}} q''$.

Whenever $q[\mathtt{rec}\,x.\tau.(p[a{\mapsto}b]\backslash b)/x] \xrightarrow{\tau} p'$, then $q[Y_x[p]/x] \xrightarrow{\tau} q'$ for some q' and either $p' \mathrel{\dot{\sim}} R \mathrel{\dot{\sim}} q'$ or for some p'', q'' and b: $p' \mathrel{\dot{\sim}} p''\backslash b$, $q' \mathrel{\dot{\sim}} q''\backslash b$ and $p'' R q''$.

We prove this by induction on the length of the inference used to establish the transition $q[\mathtt{rec}\,x.\tau.(p\backslash a)/x] \xrightarrow{\Gamma'} q'$ and cases of the structure of q. In the case where q has the form $a?y.q'$ or $\mathtt{rec}\,y.q'$ we need the above properties of substitution. The theorem then follows by choosing $q \equiv x$. (The proof follows the pattern of the proof of Proposition 4.6 of [Mil83]). □

This proof is limited to the case where at most x is free in q. The extension to the case where there are other free variables is now routine.

With the Y-context of Definition 3.19 we may program systems which recursively send out copies of themselves.

Example 3.21 Let $p \equiv c!x.x$ then according to the inference rules of Definition 2.6 $Y_x[p]$ has the following derivations:

$$
\begin{array}{c}
Y_x[p] \\
\downarrow{\scriptstyle \tau} \\
(c!x \cdot x[Y_x[p]/x] \mid nil)[a \mapsto b] \backslash b \\
\downarrow{\scriptstyle c!\,Y_x[p]} \\
(Y_x[p] \mid nil)[a \mapsto b] \backslash b \\
\downarrow{\scriptstyle \tau} \\
((c!x \cdot x[Y_x[p]/x] \mid nil)[a \mapsto b] \backslash b \mid nil)[a \mapsto b] \backslash b \\
\downarrow{\scriptstyle c!\,Y_x[p]} \\
\vdots
\end{array}
$$

This is almost a specification of a computer virus. Think of the behaviour of $Y_x[p]$ where $p = ethernet!x.(x \mid delete\_all\_files!.nil)$ and the consequences such a program could have in a network of computers connected via an ethernet.

In [Tho89] the following alternative Y-context was presented:

$$Y'_x[\,] = (a?x.([\,] \mid a!x.nil) \mid a!(a?x.([\,] \mid a!x.nil)).nil) \backslash a.$$

For "dynamic" CHOCS [Tho89] this context is limited to processes where x does not occur free in a sending position (i.e. does not occur free in any subsubexpression p' of the form $q = c!p'.p''$ where q is a subexpression of p).

However, for Plain CHOCS it also simulates recursion for processes where x occurs in a sending position due to the static nature of the restriction operator.

Example 3.22 Let $p \equiv c!x.x$ then according to the inference rules of Definition 2.6 $Y'_x[p]$ has the following derivations:

$$
\begin{array}{c}
Y'_x[p] \equiv (a?x.(c!x.x \mid a!x.nil) \mid a!(a?x.(c!x.x \mid a!x.nil)).nil) \backslash a \\
\downarrow{\scriptstyle \tau} \\
(c!(a?x.(c!x.x \mid a!x.nil)).(a?x.(c!x.x \mid a!x.nil)) \mid \\
\quad a!(a?x.(c!x.x \mid a!x.nil)).nil \mid nil) \backslash a \\
\downarrow{\scriptstyle c!_{\{a\}}(a?x.(c!x.x \mid a!x.nil))} \\
(a?x.(c!.x \mid a!x.nil) \mid a!(a?x.(c!.x \mid a!x.nil)).nil \mid nil)
\end{array}
$$

After this transition we have a scope extrusion on a, but when the "copy" $(a?x.(c!x.x \mid a!x.nil))$ is received the com-close-rule will ensure that this "copy" can communicate with $a!(a?x.(c!.x \mid a!x.nil)).nil$ and thus continue the "recursive unfolding" of p.

However, we can not prove a simulation of recursion theorem for this context as directly as Theorem 3.20. This is because when we send out copies of the recursive process we have to do a scope extrusion for a in the Y' construct to keep a connection to the remaining part and keep the "recursion" going, and the two terms are incomparable until they are received and we have closed the scope in the Y' construct. It would be interesting to formulate an equivalence theory where the kind of distributed property of a system linked by internal channels such as the above Y' construct is taken into account. I imagine that such a theory could be based on the ideas of context dependent bisimulation described by Larsen in [Lar86]. Recently Milner and Sangiorgi have proposed a notion of barbed bisimulation [MilSan92], which also seems to be a promising avenue to explore.

4 Plain CHOCS and mobile processes

In this section we compare the approach taken in this paper of sending processes to that of sending names as described in [EngNie86; MilParWal89]. We shall not embark on a discussion of which is the best or the correct way of expressing mobility in concurrent systems, since we feel that both approaches have their justifications. This is further strengthened by showing that the calculi may simulate each other.

The description of Plain CHOCS in Mobile Processes uses the capability of changing the interconnection structure of processes describable in Mobile Processes in a very disciplined way. Whenever a process is sent in Plain CHOCS a link to a trigger construct (which provides copies of the process to be sent) is sent in the Mobile Processes translation. To a certain extent this resembles invocations of procedures in conventional programming languages. The triggering of a copy of the process to be sent and the instantiation of its names could correspond to a new activation record for a procedure and instantiations of its parameters.

The description of Mobile Processes in Plain CHOCS is done by passing very small processes around. These small processes are essentially one element buffers which simulate the behaviour of channels.

This section is not self-contained. We shall not give a review of the calculus of Mobile Processes (π-Calculus) in this section, but a short review is included in Appendix B. For motivation of the constructs and exposure of the expressive power we refer to the excellent presentation given in [MilParWal89]. In the following we shall use upper case letters such as P and Q (possibly primed or indexed) to stand for Mobile Processes and lower case letters such as p and q to stand for Plain CHOCS processes. To compare terms in the π-Calculus we use a generalisation of the notion of bisimulation called strong ground bisimulation $\sim$.

Before turning to translations between the π-Calculus and Plain CHOCS we present a useful construct and show a few facts about this. We shall need

communications in the π-Calculus which carry no parameters. This could be modelled by presupposing a special name ε which is never bound and we write $\bar{x}.p$ in place of $\bar{x}\varepsilon.p$ and $x.p$ in place of $x(y).p$, where y is not free in p.

Definition 4.1 Let

$$b \Rightarrow P = \mathrm{rec}\, X.b.(P \mid X)$$

where $b \notin n(P)$ and $X \notin FV(P)$.

This construction is intended to provide copies of P when triggered by $\bar{b}$ actions e.g.:

$$(b)(\bar{b}.nil \mid \bar{b}.nil \mid b \Rightarrow P) \xrightarrow{\tau} \xrightarrow{\tau} (b)(nil \mid nil \mid P \mid P \mid b \Rightarrow P) \sim P \mid P.$$

This construct satisfies several interesting properties:

Lemma 4.2 *If $b \notin n(Q)$ then*

$$(b)(P_1 \mid b \Rightarrow Q) + (b)(P_2 \mid b \Rightarrow Q) \sim (b)((P_1 + P_2) \mid b \Rightarrow Q).$$

Proof. (Given in Appendix B). □

Lemma 4.3 *If $P_i' \not\xrightarrow{b}$ for all derivatives P_i' of P_i, $i \in \{1, 2\}$ and $b \notin fn(Q)$ then*

$$(b)(P_1 \mid b \Rightarrow Q) \mid (b)(P_2 \mid b \Rightarrow Q) \sim (b)((P_1 \mid P_2) \mid b \Rightarrow Q).$$

Proof. (Given in Appendix B). □

Lemma 4.4 *If $P_i' \not\xrightarrow{b}$ for all derivatives P_i' of P_i, $i \in \{1, 2\}$ and $b \notin fn(Q)$ and $c \notin fn(P_1) \cup fn(P_2) \cup fn(Q)$ then*

$$(c \Rightarrow (b)(P_1 \mid b \Rightarrow Q)) \mid (b)(P_2 \mid b \Rightarrow Q) \sim (b)(c \Rightarrow P_1 \mid P_2 \mid b \Rightarrow Q).$$

Proof. (Given in Appendix B). □

We now turn to the question of translations between Plain CHOCS and Mobile Processes. First we give a translation of Plain CHOCS without the renaming construct into Mobile Processes. This subset of Plain CHOCS corresponds very closely to the idea of encoding process passing in Mobile Processes described in [MilParWal89]. This translation carries no additional parameters which shows that Plain CHOCS programs can be viewed as a set of derived operators in Mobile Processes.

Definition 4.5 $[\![\ ]\!]$: Plain CHOCS $\rightarrow$ MP

$$\begin{aligned}
[\![nil]\!] &= 0 \\
[\![a?x.p]\!] &= a(x).[\![p]\!] \\
[\![a!p'.p]\!] &= (b)(\bar{a}b.([\![p]\!] \mid b \Rightarrow [\![p']\!])),\ b \notin fn(p) \cup fn(p') \cup \{a\} \\
[\![\tau.p]\!] &= \tau.[\![p]\!] \\
[\![p + p']\!] &= [\![p]\!] + [\![p']\!] \\
[\![p \mid p']\!] &= [\![p]\!] \mid [\![p']\!] \\
[\![p \backslash a]\!] &= (a)[\![p]\!] \\
[\![x]\!] &= \bar{x}.0.
\end{aligned}$$

Note how a process variable in Plain CHOCS is translated into a process which is only capable of synchronising on the x channel and then stop. This is exactly the idea described in [MilParWal89] of an executor to trigger the start of the process.

An interesting point to note about the above translation is that only a rather special kind of recursion is needed. We only need a construction which provides "copies" of the process to be sent. This construction resembles a Kleene-star operator. Combining this with Theorem 3.20 (which shows that general recursion may be simulated in Plain CHOCS) we see that using this Kleene-star operator and the dynamic interconnection mechanism provided by Mobile Processes we may simulate recursion in e.g. CCS. In fact we do not need to appeal to Theorem 3.20 to show this; The lemmas above suffice to prove $(z)(\bar{z}.0 \mid z \Rightarrow P[\bar{z}.0/X]) \sim \mathrm{rec}\, X.\tau.P$ if $z \notin fn(P)$.

Note that this translation ensures static scope for the restriction operator since the process p' being "sent" stays in the "sending" environment e.g.:

$$[\![a?x.(x|x) \mid (a!p'.p)\backslash c]\!] = a(x).(\bar{x}.0 \mid \bar{x}.0) \mid (c)((b)(\bar{a}b.((b \Rightarrow [\![p']\!]) \mid [\![p]\!])))$$

$$\downarrow \tau$$

$$(b)(\bar{b}.0 \mid \bar{b}.0 \mid (c)((b \Rightarrow [\![p']\!]) \mid [\![p]\!]))$$

$$\downarrow \tau$$

$$(b)(\bar{b}.0 \mid 0 \mid (c)([\![p']\!] \mid (b \Rightarrow [\![p']\!]) \mid [\![p]\!]))$$

$$\downarrow \tau$$

$$(b)(0 \mid 0 \mid (c)([\![p']\!] \mid [\![p']\!] \mid (b \Rightarrow [\![p']\!]) \mid [\![p]\!]))$$

$$\sim$$

$$(c)([\![p']\!] \mid [\![p']\!] \mid [\![p]\!]).$$

In this example we see how the recursion in the translation of the output prefix ensures that a sufficient number of copies of the process to be passed is provided.

As we can see from the above example the translated terms need an additional τ-move to simulate the substitution. Let us specify this on the Plain CHOCS level by introducing a notion we call τ-substitution $[/]_\tau$. This substitution is defined as $[p/x]_\tau = [\tau.p/x]$. In the following two propositions let $\rightarrow$ be a transition relation defined as the transition relation of Definition 2.6, but with $[/]_\tau$ instead of $[/]$ in the com-close-rule. Let $\dot{\sim}_\tau$ be the applicative higher order bisimulation equivalence defined as in Definition 3.1 relative to the new transition system with τ-substitution instead of the usual substitution in clause (*i*). Using these definitions we can now formally relate the two calculi. In the following $\sim$ is the strong ground bisimulation defined in [MilParWal89].

Proposition 4.6 $[\![p[q/x]_\tau]\!] \sim (b)([\![p]\!]\{b/x\} \mid b \Rightarrow [\![q]\!])$ *where* $b \notin fn(p) \cup fn(q)$.

Proof. By structural induction on p using Lemma 4.2 to Lemma 4.4. □

Proposition 4.7

1. *if* $p \xrightarrow{a?x} p'$ *then* $[\![p]\!] \xrightarrow{a(x)} [\![p']\!]$

2. *if* $p \xrightarrow{a!_B p'} p''$ *then* $[\![p]\!] \xrightarrow{\bar{a}(b)} Q \sim (b_1)\ldots(b_n)(b \Rightarrow [\![p']\!] \mid [\![p'']\!])$ *where* $B = \{b_1, \ldots, b_n\}$ *for some* Q.

3. *if* $p \xrightarrow{\tau} p'$ *then* $[\![p]\!] \xrightarrow{\tau} [\![p']\!]$

4. *if* $Q \sim [\![p]\!]$ *and* $Q \xrightarrow{a(x)} Q'$ *then* $p \xrightarrow{a?x} p'$ *for some* p' *with* $Q'\{b/x\} \sim [\![p']\!]\{b/x\}$ *for all* $b \in Names$.

5. *if* $Q \sim [\![p]\!]$ *then* $Q \not\xrightarrow{\bar{a}b}$.

6. *if* $Q \sim [\![p]\!]$ *and* $Q \xrightarrow{\bar{a}(b)} Q'$ *then* $p \xrightarrow{a!_B p'} p''$ *with* $Q' \sim (b_1)\ldots(b_n)(b \Rightarrow [\![p']\!] \mid [\![p'']\!])$ *for some* B, p', p'' *where* $B = \{b_1, \ldots, b_n\}$.

7. *if* $Q \sim [\![p]\!]$ *and* $Q \xrightarrow{\tau} Q'$ *then* $p \xrightarrow{\tau} p'$ *with* $Q' \sim [\![p']\!]$ *for some* p'.

Proof. By induction on the length of the inference used to establish the transitions observing the structure of the process p. □

The above proposition shows a strong connection between transitions of processes in Plain CHOCS and their translations into Mobile Processes. We have so far been unsuccessful in proving that the translation preserves equivalence but we conjecture that this holds under certain restrictions on the observations we allow ourselves i.e.:

Conjecture 4.8 $p_1 \dot{\sim}_\tau p_2 \Leftrightarrow [\![p_1]\!] \sim [\![p_2]\!]$.

An immediate attempt to prove the above conjecture is to show that the relation:

$$R_1 = \{(Q_1, Q_2) : \exists p_1, p_2 . Q_1 \sim [\![p_1]\!], Q_2 \sim [\![p_2]\!], p_1 \dot{\sim}_\tau p_2\}$$

is a strong ground bisimulation and that the relation

$$R_2 = \{(p_1, p_2) : [\![p_1]\!] \sim [\![p_2]\!]\}$$

is an applicative higher order bisimulation w.r.t. $[/]_\tau$. Unfortunately this attempt has so far been unsuccessful. The reason for this is that for relation R_1 I have been unable to prove that if $Q_1 \xrightarrow{a(x)} Q_1'$ then $Q_2 \xrightarrow{a(x)} Q_2'$ and $Q_1'\{b/x\} \sim Q_2'\{b/x\}$ for all $b \in Names$ from $p_1 \xrightarrow{a?x} p_1'$ and $p_2 \xrightarrow{a?x} p_2'$ and $p_1'[r/x]_\tau \dot{\sim}_\tau p_2'[r/x]_\tau$ for all r. $[\![p_1'[r/x]_\tau]\!] \sim [\![p_2'[r/x]_\tau]\!]$ does not seem to imply $[\![p_1']\!]\{b/x\} \sim [\![p_2']\!]\{b/x\}$ for all $b \in Names$. For relation R_2 I have been unable to prove that if $p_1 \xrightarrow{a!_B p_1'} p_1''$ then $p_2 \xrightarrow{a!_B p_2'} p_2''$ and $p_1' \dot{\sim}_\tau p_2'$ and $p_1'' \dot{\sim}_\tau p_2''$ from $Q_1 \xrightarrow{\bar{a}(b)} Q_1' \sim (b_1)\ldots(b_n)(b \Rightarrow [\![p_1']\!] \mid [\![p_1'']\!])$ and $Q_2 \xrightarrow{\bar{a}(b)} Q_2' \sim (b_1)\ldots(b_n)(b \Rightarrow [\![p_2']\!] \mid [\![p_2'']\!])$ and $Q_1' \sim Q_2'$. It does not seem to be possible to infer from $(b \Rightarrow [\![p_1']\!] \mid [\![p_1'']\!]) \sim (b \Rightarrow [\![p_2']\!] \mid [\![p_2'']\!])$ that $[\![p_1']\!] \sim [\![p_2']\!]$ and $[\![p_1'']\!] \sim [\![p_2'']\!]$.

The above only applies to the sublanguage of Plain CHOCS where the renaming operator has been omitted. The type of systems we can describe in this language is limited in the sense that there is no real need for passing the process in the communication since the receiving process can do no more than

copy it and start each copy at different times. This is reflected in the above translation in the sense that the process to be "sent" stays in the sending environment and the "receiving" process only receives a link which can be used to trigger copies of the "received" process. The renaming construct allows us to change the way we communicate with each copy by renaming some of the free names to locally bound names. This may be incorporated into the translation by extending the translation function by a list of names L i.e.:

Definition 4.9 $[\![\ ]\!]$: Plain CHOCS $\to$ *Names** $\to$ MP

$$
\begin{aligned}
[\![nil]\!]L &= 0\\
[\![a?x.p]\!]L &= a(x).[\![p]\!]L\\
[\![a!p'.p]\!]L &= (b)(\bar{a}b.([\![p]\!]L \mid b(L) \Rightarrow [\![p']\!]L)),\ b \notin fn(p) \cup fn(p') \cup \{a\} \cup L\\
[\![\tau.p]\!]L &= \tau.[\![p]\!]L\\
[\![p+p']\!]L &= [\![p]\!]L + [\![p']\!]L\\
[\![p \mid p']\!]L &= [\![p]\!]L \mid [\![p']\!]L\\
[\![p\backslash a]\!]L &= (d)[\![\{d/a\}\,p]\!]L \quad \text{where } d \notin fn(p\backslash a) \cup L\\
[\![p[a \mapsto b]]\!]L &= \{b/a\}([\![p]\!]L)\\
[\![x]\!]L &= \bar{x}L.0,
\end{aligned}
$$

where $b(L).p$ means $b(l_1)\ldots b(l_n).p$ and $\bar{b}L.p$ means $\bar{b}l_1\ldots\bar{b}l_n.p$ for $L = \{l_1, \ldots, l_n\}$.

When translating a Plain CHOCS expression p we then instantiate L to a list consisting of the elements of $fn(p)$ to obtain the desired effect.

Let us consider the following small example to give an idea about how the above translation works: Assume $\{a, b\} = fn(p) \cup fn(p') \cup fn(p'')$ and $b' \notin \{a, b\}$.

$$
\begin{aligned}
&[\![a?x.(x[b \mapsto b'] \mid b'?x.p)\backslash b' \mid a!p'.p'']\!]_{[a,b]}\\
&\quad =\\
&a(x).(b')(\bar{x}ab'.0 \mid b'(x).[\![p]\!]_{[a,b]}) \mid (c)(\bar{a}c.((c(a)(b) \Rightarrow [\![p']\!]_{[a,b]}) \mid [\![p'']\!]_{[a,b]}))\\
&\quad \downarrow \tau\\
&(c)((b')(\bar{c}ab'.0 \mid b'(x).[\![p]\!]_{[a,b]}) \mid (c(a)(b) \Rightarrow [\![p']\!]_{[a,b]}) \mid [\![p'']\!]_{[a,b]})\\
&\quad \downarrow \tau\\
&\quad \downarrow \tau\\
&(c)((b')(0 \mid b'(x).[\![p]\!]_{[a,b]} \mid (\{a/a\}\{b'/b\}([\![p']\!]_{[a,b]}))) \mid (c(a)(b) \Rightarrow [\![p']\!]_{[a,b]}) \mid [\![p'']\!]_{[a,b]})\\
&\quad \sim\\
&(b')(b'(x).[\![p]\!]_{[a,b]} \mid (\{a/a\}\{b'/b\}([\![p']\!]_{[a,b]}))) \mid [\![p'']\!]_{[a,b]}.
\end{aligned}
$$

Note that if p' has any b-ports they will be renamed to b' and thus be private between $b'(x).[\![p]\!]_{[a,b]}$ and $[\![p']\!]_{[a,b]}$. The translated terms need a sequence of τ-

transitions to establish the connection between the "receiving" process and the "copy" it is "receiving". This sequence has the same length as the parameter L. In the above example we needed two τ-transitions and in general we will need as many τ-transitions as the cardinality of the set $fn(p)$.

We now turn to the question of encoding label passing using process passing. This may seem as an artificial question, but as a theoretical result it is of interest since it will provide a basis for discussion of the expressive power of the two approaches.

The idea in the translation below is that instead of sending a channel a we send a wire $(a-chan)$ defined as $i?.a?x.c!x.nil+o?.c?x.a!x.nil$. This wire has a multi-purpose plug c and a switch to indicate in which direction the wire is to be used. We assume c, i, o are distinct names not used in the Mobile Processes expression being translated. When this wire is received it is plugged into the receiving process by the localising constructions: $(\ldots[c\mapsto c'][i\mapsto i'][o\mapsto o']\ldots)\backslash c'\backslash i'\backslash o'$. The receiving process will choose in which direction to use this wire by sending an o' signal for output or an i' signal for input. The wire will be private to the sending and receiving processes in the case of a bound name in the Mobile Processes expression. This is ensured by a scope extrusion caused by the static restriction operator.

Mobile Processes [MilParWal89] was developed from ECCS [EngNie86] by simplifying the notions of values, labels and variables into one concept called names. This, however, presents a problem when translating Mobile Processes into Plain CHOCS since a name in a process P may act as a name of a link (as e.g. y in $y(x).P$) or it may act as a variable (as e.g. x in $y(x).P$) or it may act as a local link name (as e.g. x in $(x)P$). To overcome this difficulty we first translate all free names and all names bound by input prefix into process variables. Then we instantiate the process variables corresponding to free names in the Mobile Processes expression to names in Plain CHOCS. Names bound by restriction will be allocated names in Plain CHOCS in the first translation step.

Definition 4.10 $[\![\ ]\!]_1$: MP $\to$ Plain CHOCS is defined structurally:

$$
\begin{aligned}
[\![0]\!]_1 &= nil \\
[\![x(y).P]\!]_1 &= (x[c\mapsto c'][i\mapsto i'][o\mapsto o'] \mid i'!.c'?y.[\![P]\!]_1)\backslash c'\backslash i'\backslash o' \\
[\![\bar{x}y.P]\!]_1 &= (x[c\mapsto c'][i\mapsto i'][o\mapsto o'] \mid o'!.c'!y.[\![P]\!]_1)\backslash c'\backslash i'\backslash o' \\
[\![\tau.P]\!]_1 &= \tau.[\![P]\!]_1 \\
[\![P+P']\!]_1 &= [\![P]\!]_1 + [\![P']\!]_1 \\
[\![P\mid P']\!]_1 &= [\![P]\!]_1 \mid [\![P']\!]_1 \\
[\![(x)(P)]\!]_1 &= ([\![P]\!]_1[(a-chan)/x])\backslash a, \text{ where } a\notin fn(P).
\end{aligned}
$$

$[\![\ ]\!]_2$: MP $\to$ Plain CHOCS is defined as:

$$[\![P]\!]_2 = (\ldots([\![P]\!]_1[(a_1-chan)/x_1])\ldots)[(a_n-chan)/x_n],$$

where $FV([\![P]\!]_1)=\{x_1, \ldots, x_n\}$ and $a_1 \ldots a_n$ are allocated by some $1-1$ mapping between V and $Names$ (usually established by the $1-1$ mapping between $fn(P)$ and $FV([\![P]\!]_1)$).

We have omitted the match construct of Mobile Processes. This can be eliminated in the Mobile Processes expression according to Example 9 in Sect. 4 of [MilParWal89]. Recursion could be translated using the $Y_x[\,]$ construction from the previous section.

It is easy to see from the above definition that name passing in the Mobile Processes is mimicked by the translation only requiring two additional communications for each use of the wire, i.e.:

$$[\![a(x).P]\!]_2 \dot{\sim} \tau.a?x.\tau.[\![P]\!]_2$$
$$[\![\bar{a}b.P]\!]_2 \dot{\sim} \tau.\tau.a!(b-chan).[\![P]\!]_2 .$$

We may state this more precisely:

Proposition 4.11

1 *if* $P \xrightarrow{a(x)} P'$ *then* $[\![P]\!]_2 \xrightarrow{\tau} \xrightarrow{a?x} \xrightarrow{\tau} [\![P']\!]_2$

2. *if* $P \xrightarrow{\bar{a}b} P'$ *then* $[\![P]\!]_2 \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{a!_\emptyset(b-chan)} [\![P']\!]_2$

3. *if* $P \xrightarrow{\bar{a}(b)} P'$ *then* $[\![P]\!]_2 \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{a!_{\{b\}}(b-chan)} [\![P']\!]_2$

4. *if* $P \xrightarrow{\tau} P'$ *then* $[\![P]\!]_2 \xrightarrow{\tau} [\![P']\!]_2$ *or* $[\![P]\!]_2 \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{\tau} [\![P']\!]_2$.

Proof. By induction on the length of the inference used to establish the transition of P observing the structure of P. □

We conjecture the following relationship between Mobile Processes and their translations into Plain CHOCS:

Conjecture 4.12 *If* $P \sim Q$ *then* $[\![P]\!]_2 \dot{\approx} [\![Q]\!]_2$, where $\dot{\approx}$ *is a suitable formulation of weak higher order applicative bisimulation.*

We can not hope for the implication to hold in the opposite direction since the translation may introduce non-determinism not present in the original term e.g.: Consider the following term $P=(a)(b)(a(x).c(x).0+b(x).0 \mid \bar{a}c.0)$ then $P \sim \tau.c(x).0 \not\sim P+\tau.0$ whereas $[\![P]\!]_2 \dot{\sim} \tau.\tau.\tau.\tau.\tau.c?x.\tau.nil+\tau.0 \dot{\sim} [\![P+\tau.0]\!]_2$.

To see how the translation works we study the following small system consisting of two components. Initially the first component is ready to receive a channel on a and the second component is ready to send the b-channel on a. Upon receiving a channel the first component is ready to send a bound channel d on the newly received channel. The second component is ready to receive this channel. The end result is that the second component receives a private d-channel from the first component.

$$[\![a(x).(d)(\bar{x}d.P) \mid \bar{a}b.b(x).Q]\!]_2$$
$$\dot{\sim}$$
$$\tau.a?x.\tau.[\![(d)(\bar{x}d.P)]\!]_2 \mid \tau.\tau.a!(b-chan).[\![b(x).Q]\!]_2$$
$$\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau}$$
$$([\![(d)(\bar{x}d.P)]\!]_2[(b-chan)/x] \mid [\![b(x).Q]\!]_2)$$
$$\dot{\sim}$$
$$((\tau.\tau.b!(d-chan).[\![P]\!]_2[(b-chan)/x])\backslash d \mid \tau.b?x.\tau.[\![Q]\!]_2)$$
$$\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau}$$
$$(([\![P]\!]_2[(b-chan)/x] \mid [\![Q]\!]_2[(d-chan)/x])\backslash d).$$

Comparing the two translations presented in this section we see that the two caluli Mobile Processes and Plain CHOCS are equally expressive in the sense that they may simulate one another. However, the translations are rather ad hoc. It would be of interest if this comparison could be formulated in a more general framework for comparison. One such study has recently been undertaken by Sangiorgi. Based on the notion of barbed bisimulation [MilSan92] he shows in his forthcoming thesis [San92] how higher order processes can be simulated in the π-Calculus.

5 Plain CHOCS and object-oriented programming

Over the past two decades object-oriented programming has grown into a strong discipline in the world of industrial programming. One reason for the success of this programming notion is the link with ideas of structured programming. Object oriented programming allows problems to be broken down into "objects" of manageable size. There is to date no unifying definition of what exactly an object is and what an object does, although over the years much effort has been devoted to finding such definitions. It seems as if each object-oriented programming language (and even each object oriented programmer) has its (his/her) own definition of an object.

This having been said, there seems to be a consensus that an object is regarded as an encapsulating entity and there are strong analogies to the ideas of abstract data types. Thus objects encapsulate "things" and users access these "things" via "methods". The idea behind the method paradigm is to present the user with an interface through which objects can be accessed and at the same time hide the way the objects are implemented. Most present day object-oriented programming languages have roots in ideas presented in the SIMULA language [DahMyhNyg68] designed in the late sixties, and ideas presented in the Smalltalk language [GolRob83] have had substantial influence.

The object-oriented approach has mostly grown out of an imperative sequential programming discipline as a structuring device for large scale programs, but recently it has been recognised as a useful tool in the description and construction of distributed and concurrent systems [Atk89]. As we shall see in this section there seems to be a strong analogy between the idea of objects and processes, encapsulation and restriction, method call and communication via named channels. We shall also see that it is possible to make connections between concurrency theory and inheritance, which for many object-oriented programmers seems to be a vital part of the definition of what can be characterised as object-oriented programming.

Many object oriented programming languages do not have a formal semantics but rely on (thorough) verbal descriptions of the semantics. Recently some more thorough studies of semantics foundations of object-oriented programming languages have emerged, POOL [Ame87] and Dragoon [Atk89] are very good examples of how far the current state of affairs for real life programming languages has reached.

In this section we study the connection between concurrency and object oriented programming in more detail. We do this via a small toy language O. We may consider O as a prototype core of most imperative concurrent object

oriented programming languages. In O we may define a class of objects and instantiate objects to be of a defined class. In each class we may define a number of methods and a thread of control. This thread of control is the primary means for concurrency since objects may be started and executed in parallel. The parallelism is asynchronous, and synchronisation is obtained by method calls. O was inspired by the toy language P studied in [Mil80] and in [Tho89], and the thread of control in each object is similar to the sequential part of P. Expressions in the language O are untyped, but for the cause of simplicity we only consider type meaningful programs. We assume that objects are declared before they are created, that all objects are created before started and that all objects are started only once.

The semantics of O is described in Plain CHOCS in a phrase-by-phrase style resembling a denotational semantics. However, we do not give any semantic domains. Instead we may view the O semantics as a set of derived operators in Plain CHOCS since the translation carries no parameters. Plain CHOCS only caters for process values in communication. To allow for other values in Plain CHOCS than process values we use the technique of [Mil83] and introduce a $\mathscr{D}$-indexed family of actions $a?_d$, $a!_d$, $d \in \mathscr{D}$ for each value domain $\mathscr{D}$. Due to the fact that only finite sums of processes can be handled in Plain CHOCS we restrict our attention to finite value domains as e.g. the set of booleans and finite subsets of the integers. We let $\alpha?_x.p$ abbreviate $\Sigma_{d \in D} \alpha?_d.p\{d/x\}$ where $\{d/x\}$ means exchanging all occurrences of $_x$ in p by $_d$ as e.g. $\alpha?_x.\beta!_x.nil\{d/x\} \equiv \Sigma_{d \in D} \alpha?_d.\beta!_d.nil$. We shall use the following construct from [Mil83]: If b is a boolean valued expression in $_x$ then let $\alpha?_x.(if\ b\ then\ p\ else\ p')$ be encoded by $\Sigma_{d \in D \& b} \alpha?_d.p + \Sigma_{d \in D \& \neg b} \alpha?_d.p'$. We should not confuse $\alpha?_x.p$ with $\alpha?x.p$ since the first is a convenient shorthand notation and the latter is part of the Plain CHOCS syntax.

The language O

Programs in O are built from declarations D, expressions E and commands C. In declarations we may declare program variables ranged over by X, object variables Y, methods P with parameter X to be instantiated by reference in the command body C, and we may declare a class Z with local declarations D and command body C. Some set of functions F is assumed and for the cause of simplicity we do not consider types of expressions. Commands are assignments to program variables, sequencing, conditionals, while loops, skip statements, blocks with local declarations and three commands for creating an object of class Z bound to the object variable Y, a command for triggering the start of the command body of object Y and a command for calling method P of object Y with X as parameter. O has the following abstract syntax:

Table 2. Syntax of O

Declarations:	$D ::= \texttt{var}\ X \mid \texttt{obj}\ Y \mid D; D \mid$ $\texttt{method}\ P(\texttt{ref}\ X)\ \texttt{is}\ C \mid \texttt{class}\ Z\ \texttt{is}\ D\ \texttt{body}\ C$
Expressions:	$E ::= X \mid F(E_1, \ldots, E_n)$
Commands:	$C ::= X := E \mid C; C \mid \texttt{if}\ E\ \texttt{then}\ C\ \texttt{else}\ C' \mid$ $\texttt{while}\ E\ \texttt{do}\ C \mid \texttt{skip} \mid \texttt{begin}\ D; C\ \texttt{end} \mid$ $Y.\texttt{create}\ Z \mid Y.\texttt{start} \mid Y.\texttt{call}\ P(X)$

We do not put restrictions on how often a variable, object, method or class is declared in the same scope. To ensure a deterministic semantics one could require that variables, objects, methods or classes are only defined once in the same scope. The semantics presented below will yield a nondeterministic choice between two declarations of the same name.

To give a smooth definition of the semantics of O we need some auxiliary definitions.

To each variable X we associate a register Reg_X. Generally it follows the pattern:

$$Loc = \alpha?_x.Reg(x)$$
$$Reg(y) = \alpha?_x.Reg(x) + \gamma!_y.Reg(y)$$

and thus for X we will have $Loc_X = Loc[\alpha \mapsto \alpha_X][\gamma \mapsto \gamma_X]$. Initially we write in a value, thereupon we can read this value on γ or overwrite the contents of Loc via α. We have written the above definition in an equation style to make it more readable. The proper Plain CHOCS definition is:

$$Loc = (\alpha?_x.h!_x.nil \mid Reg) \backslash h$$

where

$$Reg = Y_{Reg}[h?_x.(\alpha?_x.h!_x.Reg + \gamma!_x.h!_x.Reg)] \mid Y_{Keep}[h?_x.h!_x.Keep].$$

The second component of this process takes care of the parameters in the recursion of the above equations. (This is in fact a general technique for simulating the parameterised recursion of [Mil83]). We also associate a register to each class Z, each object Y and each method P. It may be defined in the same way as above with $_x$ substituted with x.

To each n-ary function symbol F we associate a function f which is represented by:

$$b_f = \rho_1?_{x_1} \ldots \rho_n?_{x_n}.\rho!_{f(x_1 \ldots x_n)}.nil.$$

Constants will thus be represented as e.g. $b_{true} = \rho!_{true}.nil$. The result of evaluating an expression is always communicated via ρ. It is therefore useful do define:

$$p \; result \; p' = (p \mid p') \backslash \rho.$$

We adopt the protocol of signaling successful termination of commands via δ and it is therefore convenient to define:

$$done = \delta!.nil$$
$$p \; before \; p' = (p[\delta \mapsto \beta] \mid \beta?.p') \backslash \beta, \beta \notin fn(p) \cup fn(p').$$

We now give the semantics of O by the translation into Plain CHOCS shown in Table 3.

Table 3. Semantics of O

Declarations:

$$[\![\texttt{var } X]\!] = Loc_X$$
$$[\![\texttt{obj } Y]\!] = Loc_Y$$
$$[\![D; D']\!] = [\![D]\!] \mid [\![D']\!]$$
$$[\![\texttt{method } P(\texttt{ref } X) \texttt{ is } C]\!] = ((Loc_P \mid \alpha_P!(\textit{method process}).nil) \backslash \alpha_P)$$
$$[\![\texttt{class } Z \texttt{ is } D \texttt{ body } C]\!] = ((Loc_Z \mid \alpha_Z!(\textit{class process}).nil) \backslash \alpha_Z)$$

where $\textit{method process} = [\![C]\!][\alpha_X \mapsto \alpha_{P_v}][\gamma_X \mapsto \gamma_{P_v}]$
and $\textit{class process} = ([\![D]\!][\alpha_{P_v^1} \mapsto \alpha^Z_{P_v^1}][\gamma_{P_v^1} \mapsto \gamma^Z_{P_v^1}] \ldots [\alpha_{P_v^n} \mapsto \alpha^Z_{P_v^n}][\gamma_{P_v^n} \mapsto \gamma^Z_{P_v^n}] \mid [\![C]\!]) \backslash V_D$.

Expressions:

$$[\![X]\!] = \gamma_X?_x.\rho!_x.nil$$
$$[\![F(E_1, \ldots, E_n)]\!] = ([\![E_1]\!][\rho_1/\rho] \mid \ldots \mid [\![E_n]\!][\rho_n/\rho] \mid b_f) \backslash \rho_1 \ldots \backslash \rho_n$$

Commands:

$$[\![X := E]\!] = [\![E]\!] \textit{ result } (\rho?_x.\alpha_X!_x.\textit{done})$$
$$[\![C; C']\!] = [\![C]\!] \textit{ before } [\![C']\!]$$
$$[\![\texttt{if } E \texttt{ then } C \texttt{ else } C']\!] = [\![E]\!] \textit{ result } \rho?_x.(\textit{if}_x \textit{ then } [\![C]\!] \textit{ else } [\![C']\!])$$
$$[\![\texttt{while } E \texttt{ do } C]\!] = Y_w[\![E]\!] \textit{ result } \rho?_x.(\textit{if}_x \textit{ then } ([\![C]\!] \textit{ before } w) \textit{ else done})]$$
$$[\![\texttt{skip}]\!] = \textit{done}$$
$$[\![\texttt{begin } D; C \texttt{ end}]\!] = ([\![D]\!] \mid [\![C]\!]) \backslash L_D$$
$$[\![Y.\texttt{create } Z]\!] = \gamma_Z?x.\alpha_Y!(x[\alpha^Z_{P_v} \mapsto \alpha^Y_{P_v}][\gamma^Z_{P_v} \mapsto \gamma^Y_{P_v}]).\textit{done}$$
$$[\![Y.\texttt{start}]\!] = \gamma_Y?x.(x[\delta \mapsto \beta] \mid \beta?.nil \mid \textit{done}) \backslash \beta$$
$$[\![Y.\texttt{call } P(X)]\!] = \gamma^Y_P?x.(x[\alpha^Y_{P_v} \mapsto \alpha_X][\gamma^Y_{P_v} \mapsto \gamma_X])$$

In the definition of *class process* we let $\backslash V_D$ abbreviate restrictions with respect to all variables and objects declared in D, and $[\alpha_{P_v^1} \mapsto \alpha^Z_{P_v^1}][\gamma_{P_v^1} \mapsto \gamma^Z_{P_v^1}] \ldots [\alpha_{P_v^n} \mapsto \alpha^Z_{P_v^n}][\gamma_{P_v^n} \mapsto \gamma^Z_{P_v^n}]$ is a renaming for each method (assumed to be named $P^1, \ldots, P^n$) defined in D. In the equation for $[\![\texttt{begin } D; C \texttt{ end}]\!]$ we let $\backslash L_D$ abbreviate restriction with respect to α and γ channels for all variables, objects, classes and methods declared in D. The method and class definitions each create a location to store the method process respectively the class process. The restrictions $\backslash \alpha_P$ respectively $\backslash \alpha_Z$ ensure that these processes can not be overwritten after their definitions.

Note that if we disregard the object oriented part of O we have essentially a language definition similar to the definition of P from [Tho89]. However, if we compare the semantic definition of procedures in P with the semantic definition of methods in O we note that the *Transform* process needed to ensure static binding of variables in the P semantics is no longer present in the O semantics. This is not because we advocate dynamic binding for variables in the object oriented paradigm. It is because the static nature of the restriction operator in Plain CHOCS will ensure that static binding is obtained. The static nature will ensure (by a scope extrusion) that any variable reference is kept with the defining environment. Assignments to variables may be nondeterministic since two or more methods may refer to the same variable, and we can have situations where one method reads the value currently stored then another method writes a new value before the first method overwrites the current value. As for the semantic description of P we can avoid this problem by surrounding each variable with a semaphore construct.

A class is defined as a process stored in a register. The class process behaves like a block except that we can invoke the methods defined in the declaration part. These will execute concurrently with the thread defined by the command part of the class process. A class is a passive entity in the sense that is stored in a register. An object Y of class Z is just a copy of the class process stored in another register. It becomes active when started by the $Y.\texttt{start}$ command which reads the register and activates the process by the $\gamma_Y?x.(x[\delta \mapsto \beta] \mid \beta?.nil \mid done)\backslash\beta$ construct. The finish signal δ from the activated process is renamed into β which is going to synchronise with $\beta?.nil$ when the process terminates. The object which started Y continues its execution since as soon as the register has been read the *done* part may issue a δ signal. The object which starts Y could be forced to wait until Y terminates by defining $[\![Y.\texttt{start}]\!] = \gamma_Y?x.x$ since then the δ signal to indicate the end of the $Y.\texttt{start}$ command would have to come from Y. Each method is also just a process stored in a register. When a method is called the register is read and a copy of the method process is activated. The renaming surrounding the variable x ensures a call-by-reference parameter mechanism in the method call. This parameter mechanism seems to be most in line with current trends in object oriented programming, but we can also define call-by-value, call-by-name and lazy parameter mechanisms for method calls in O using the same approach as in the definition of parameter mechanisms in P discussed in [Tho89].

The semantic definition of O has not taken the object oriented paradigm to its extreme where everything is an object. We have kept a distinction between objects, values and methods. We can go a bit further and describe how objects can be passed in method call. To some object oriented programmers this is the true spirit of the object oriented paradigm. Let us see how object passing in method call can be described semantically:

$$[\![\texttt{method } P(\text{obj } Y) \texttt{ is } C]\!] = ((Loc_P \mid \alpha_P!(\textit{method process}).nil)\backslash\alpha_P),$$

where $\textit{method process} = [\![C]\!][\alpha^Y_{P_v} \mapsto \alpha_{P_v}][\gamma^Y_{P_v} \mapsto \gamma_{P_v}]$

$$[\![Y.\texttt{call } P(Y')]\!] = \gamma^Y_P?x.(x[\alpha_{P_v} \mapsto \alpha^{Y'}_{P_v}][\gamma_{P_v} \mapsto \gamma^{Y'}_{P_v}])\ \textit{before}\ \textit{done}.$$

Passing an object in a method call works very similar to the call-by-reference parameter mechanism for normal method calls. We simply rename the method calls of the formal parameter to method calls of the actual parameter. It should be mentioned that this mechanism does not allow for arbitrary assignment of objects among variables (which is also often considered to be part of the true spirit of the object oriented paradigm). This is left for future studies, but it is envisioned that an easy adaptation of assignment to program variables may facilitate this.

Another phenomenon often connected with object oriented programming languages is the concept of inheritance. This is often considered the main structuring mechanism. We may describe this semantically as follows:

$$[\![\texttt{class } Z \texttt{ inherits } Z' \texttt{ is } D \texttt{ body } C]\!] = ((Loc_Z \mid \alpha_Z!(\textit{class process}).nil)\backslash\alpha_P)$$

and

$$\textit{class process} = (\gamma_{Z'}?x.(x[\alpha^{Z'}_{P_v} \mapsto \alpha^Z_{P_v}][\gamma^{Z'}_{P_v} \mapsto \gamma^Z_{P_v}]$$
$$\textit{before}\ ([\![D]\!][\alpha_{P_v} \mapsto \alpha^Z_{P_v}][\gamma_{P_v} \mapsto \gamma^Z_{P_v}] \mid [\![C]\!])\backslash V_D).$$

This describes that the class Z inherits the methods and the thread of control of class Z'. All methods of Z' are renamed to methods of Z and the thread of control of Z' is sequentially composed with that of Z. It is easy to generalise this to multiple inheritance simply by sequentially composing each inheritance class. In some object oriented programming languages programmers are allowed to redefine inherited methods. This is easily obtained by restricting the α and γ channels of the redefined method from the inheritance class and redefining it in the declaration part of the class.

This section represents a small step towards a semantic description of object-oriented programming in Plain CHOCS. Syntactically O is very similar to the core of POOL [Ame87]. The main difference is that O contains one construct for creating an object and another one for starting it, whereas in POOL objects are started as soon as they are created. Furthermore method calls in POOL are only answered through an explicit command, whereas in O they are processed concurrently with the sequential part of the object. However, the two languages are rather different semantically since O has a copy semantics, where the code of the object is copied when assigned to an object variable. POOL has a reference semantics, where a reference to the object is copied when assigned to an object variable. This is very well illustrated by the semantics cast in the π-Calculus for core POOL given in [Wal91]. Through the translation of the π-Calculus we may claim that we can use Walker's translation and obtain a Plain CHOCS semantics for core POOL. However, this would be rather unnatural. Studying the (very elegant) presentation of the POOL semantics in [Wal91] there are a few points, such as class definitions and method declarations, where the semantic description could benefit from using higher order processes. This seems to call for a notion where name passing can be mixed with process passing. This is provided by the recently developed Polyadic π-Calculus [Mil91]. These prospects are left for future studies.

6 Concluding remarks

The presentation in this paper has focused on the theoretical aspects of introducing process passing in a CCS like language. Higher order constructs arise in almost any branch of theoretical computer science, since they yield elegant and powerful abstraction techniques. In this paper and in [Tho89] we have studied how to extend CCS with processes as first class objects. We have seen that the operations of prefix, (nondeterministic) choice and parallel composition are equally fundamental and do not allow much variation. But there seems to be room for various constructs of the restriction\renaming nature. In this paper we have followed the ideas of [EngNie86; MilParWal89]. We have described how a restriction operator with static scope could be introduced and how this calls for an interplay with the parallel operator. In [Tho89] we showed the usefulness of such an operator, together with a renaming operator, where both had a dynamic nature.

Section 4 shows that the distinction between process passing and name passing is more on the level of abstraction than on the level of expressive power. The translation from Plain CHOCS into Mobile Processes resembles an implementation of a procedural imperative programming language in a more primitive language using Goto's. In the given translation the sending of the name plays

the rôle of the Goto and the recursion construction resembles an activation stack (with all entries active). It is an interesting path to pursue for further research how CHOCS could act as a high level specification language and Mobile Processes as an implementation language. The equational theories on both levels open possibilities for program transformations on both the specification and on the implementation level.

The study of Plain CHOCS as a metalanguage for the specification of programming languages has only just begun by the application of Plain CHOCS in giving a semantics to the object-oriented programming language *O* presented in Sect. 5. However, in this section we have seen that quite complex notions such as concurrent method invocations, object passing in method calls and inheritance can easily be described and investigated using Plain CHOCS. It is a major challenge to the theory to apply it to larger examples.

In this paper and in [Tho89] we have taken the stand that the processes sent are inactive until they are received and put into use by the receiving process. The underlying idea behind this is that it is a process description, either the text as in [Tho89] or a semantic description as in this paper. A different viewpoint was taken by Kennaway and Sleep in [KenSle83]. They chose to send a running process (script) to describe an SKI-reduction algorithm in an Actor language. Unfortunately it is hard to see how to formalise this idea from the informal operational semantics they gave. An interesting idea may be found in [Bou89]. Here internal activity is allowed for the process to be sent. This could be described by the rule:

$$\frac{p' \xrightarrow{\tau} p''}{a!p'.p \xrightarrow{\tau} a!p''.p}.$$

In [Bou89] Boudol presents a translation of the λ-Calculus similar to the one given in [Tho89]. But the evaluation strategy becomes more "eager" due to the above rule. This brings the reduction strategy closer to full β-reduction, whereas the strategy in [Tho89] coincides with the Lazy-λ-Calculus as described in [Abr90]. In [Nie89] Nielson has a similar rule to the above in addition with the following rule:

$$\frac{p \xrightarrow{\tau} p''}{a!p'.p \xrightarrow{\tau} a!p'.p''}.$$

Both these rules seem to violate the idea that the prefix operators are primitives for sequentiality. If one accepts to abandon this principle then the processes should be allowed external activity as well, but then it is hard to see what effect this will have on the equational theory.

We have not embarked on a discussion of a theory where τ-transitions are interpreted as unobservable. But we conjecture that an easy generalisation of the bisimulation predicate may yield a notion of observational equivalence. For technical reasons we would have to define $\xrightarrow{a?x}_0 \equiv \xrightarrow{\tau} * \xrightarrow{a?x}$ since $\dashrightarrow \subseteq CPr \times Pr[x]$ (intuitively $CPr \times [CPr \to CPr]$) and it would not make sense

to write $\stackrel{a?x}{\Longrightarrow} \equiv \stackrel{\tau}{\longrightarrow}^* \stackrel{a?x}{\longrightarrow} \stackrel{\tau}{\longrightarrow}^*$. We could define the output transitions as $\stackrel{a!p'}{\Longrightarrow} \equiv \stackrel{\tau}{\longrightarrow}^* \stackrel{a!p'}{\longrightarrow} \stackrel{\tau}{\longrightarrow}^*$ but this would introduce an unnecessary asymmetry. It is interesting to note that in [Wal88], where bisimulation and divergence is studied in the context of CCS, the equivalence relation generated using $\Rightarrow$ is stronger than the one using $\Rightarrow_0$. In "dynamic" CHOCS [Tho89] the observational equivalence $\approx$ does not enjoy the property of being a congruence with respect to the operators of CHOCS. As for CCS, it is the (nondeterministic) choice operator which gives the problem and this may be seen from the following counter example first presented in [Mil80]: $\tau.nil \approx nil$ but $a!p.nil + \tau.nil \not\approx a!p.nil + nil$. We may obtain a congruence using techniques presented in [Mil80] by defining $p \approx^c q$ iff $\forall C.C[p] \approx C[q]$, where C is a context. It is an interesting problem to study how to extend the theory presented here to a theory of observational equivalence.

Another major challenge is to establish a denotational theory. The operational modelling of input suggests that input should be modelled by function space $D \to D$, but the behaviour is also dependent on the set of bound names exported in scope extrusion, thus one suggestion for a denotation domain worth investigating is:

$$D \cong P^0[\Sigma_{a \in Names}[Names \to D] \to D + \Sigma_{a \in Names} Names \times D \times D + D],$$

where P^0 is the Plotkin Power Domain with the empty set adjoined as defined in [Abr91].

In the context of Plain CHOCS there is a very interesting variant of the applicative higher order bisimulation which deserves to be explored:

Definition 6.1 A variant applicative higher order simulation R is a binary relation on CPr such that whenever pRq and $a \in Names$ then:

(i) Whenever $p \stackrel{a?x}{\longrightarrow} p'$, then for all $r \in CPr$ there is some q', y such that $q \stackrel{a?y}{\longrightarrow} q'$ and $p'[r/x] \, Rq'[r/y]$

(ii) Whenever $p \stackrel{a!_B p'}{\longrightarrow} p''$ with $B \cap (fn(p) \cup fn(q)) = \emptyset$, then $q \stackrel{a!_B q'}{\longrightarrow} q''$ for some q', q'' with $p' Rq'$ and $p'' Rq''$

(iii) Whenever $p \stackrel{\tau}{\longrightarrow} p'$, then $q \stackrel{\tau}{\longrightarrow} q'$ for some q' with $p' Rq'$

A relation R is a variant applicative higher order bisimulation if both it and its inverse are applicative higher order simulations.

If there exists a variant applicative higher order bisimulation R containing (p, q) we write $p \dot{\sim}' q$.

This relation is obtained by commuting the quantifiers in the first clause of Definition 3.1. The relation $\dot{\sim}'$ is interesting since it is stronger than the applicative higher order bisimulation relation. A similar variation of strong ground bisimulation was suggested in [MilParWal89] and it was shown that in the context of Mobile Processes the variant relation is strictly stronger. It is an open question if the inclusion is strict in the context of Plain CHOCS.

Acknowledgement. The presentation in this paper owes much to very valuable comments given by the anonymous referees. Special thanks for comments go to the following people: S. Abramsky, J. Cozens, C. Crasemann, A. Giacalone, M. Hennessy, L. Leth, R. Milner, P. Mishra, F. Nielson, L. Ong, I. Phillips and S. Prasad.

Most of this work has been carried out while I was employed as a research assistant on the Foundational Models for Software Engineering project (SERC GR-F 72475) within the Department of Computing, Imperial College. I have also been supported by The Danish Natural Science Research Council and The Danish Research Academy.

References

[Abr90] Abramsky, S.: The lazy lambda calculus. In: Turner, D. (ed.) Research Topics in Functional Programming, chap. 4, pp. 65–116. Reading: Addison-Wesley 1990

[Abr91] Abramsky, S.: A domain equation for bisimulation. Inf. Comput. **92**(2), 161–218 (1991)

[Ame87] America, P.: POOL-T: A parallel object-oriented language. Proceedings of object-oriented concurrent programming, pp. 199–220. Cambridge, MA: MIT Press 1987

[AstReg87] Astesiano, E., Reggio, G.: SMoLS-driven concurrent calculi. Proceedings of TAPSOFT 87 (Lect. Notes Comput Sci., vol. 249, pp. 169–201) Berlin Heidelberg New York: Springer 1987

[Atk89] Atkinson, C.: An object-oriented language for software reuse and distribution. Ph. D. Thesis, Department of Computing, Imperial College, London University 1989

[Bou89] Boudol, G.: Towards a lambda-calculus for concurrent and communicating systems. Proceedings of TAPSOFT 89 (Lect. Notes Comput. Sci., vol. 351, pp. 149–161) Berlin Heidelberg New York: Springer 1989. Preliminary version in Research Report no. 885, INRIA Sophia Antipolis, autumn 1988

[Chr88] Christensen, P.: The domain of CSP processes (incomplete draft). The Technical University of Denmark 1988

[Coz90] Cozens, J.: Adaptable computer systems (incomplete draft). University of Surrey 1990

[DahMyhNyg68] Dahl, O.J., Myhrhaug, B., Nygaard, K.: SIMULA 67 Common base language. Norwegian Computing Center 1968

[EngNie86] Engberg, U., Nielsen, M.: A calculus of communicating systems with label passing. Technical report DAIMI PB-208. Computer Science Department, Aarhus University 1986

[GiaMisPra90] Giacalone, A., Mishra, P., Prasad, S.: Operational and algebraic semantics for FACILE: A symmetric integration of concurrent and functional programming. Proceedings of ICALP 90. (Lect. Notes Comput. Sci., vol. 443, pp. 765–780) Berlin Heidelberg New York: Springer 1990

[GolRob83] Goldberg, A., Robson, D.: Smalltalk 80: The language and its implementation. Reading: Addison-Wesley 1983

[HenNic87] Hennessy, M., Nicola de, R.: CCS without τ's (Lect. Notes Comput Sci., vol. 249, pp. 138–152). Berlin Heidelberg New York: Springer 1987

[KenSle83] Kennaway, J.R., Sleep, M.R.: Syntax and informal semantics of DyNe, a parallel language. Proceedings of workshop on the analysis of concurrent systems 1983. (Lect. Notes Comput. Sci., vol. 207, pp. 222–230) Berlin Heidelberg New York: Springer 1985

[KenSle88] Kennaway, J.R., Sleep, M.R.: A denotational semantics for first class processes (Draft). School of Information Systems, University of East Anglia, Norwich 1988

[Lar86] Larsen, K.G.: Context dependent bisimulation between processes. Ph.D. Thesis, Edinburgh University 1986

[Mil80] Milner, R.: A calculus of communicating systems (Lect. Notes Comput. Sci., vol. 92) Berlin Heidelberg New York: Springer 1980

[Mil83] Milner, R.: Calculi for synchrony and asynchrony. Theor. Comput. Sci. **25**, 267–310 (1983)

[Mil89] Milner, R.: Communication and concurrency. Englewood Cliffs, NJ: Prentice Hall 1989

[MilParWal89] Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, Part I. Report ECS-LFCS-89-85, University of Edinburgh 1989
[MilParWal89b] Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, Part II. Report ECS-LFCS-89-86, University of Edinburgh 1989
[Mil91] Milner, R.: The polyadic π-calculus: A Tutorial Report ECS-LFCS-91-180, University of Edinburgh 1991
[MilSan92] Milner, R., Sangiorgi, D.: Barbed bisimulation. Proceedings of ICALP 92. (Lect. Notes Comput. Sci., vol. 623, pp. 685–695) Berlin Heidelberg New York: Springer 1992
[Nie89] Nielson, F.: The typed λ-calculus with first-class processes. Proceedings of PARLE 89. (Lect. Notes Comput. Sci., vol. 336, pp. 357–376) Berlin Heidelberg New York: Springer 1989. (Preliminary version: Technical Report ID-TR: 1988-43 ISSN 0902-2821, Department of Computer Science, Technical University of Denmark, August 1988)
[Par81] Park, D.: Concurrency and automata on infinite sequences. In: Deussen, P. (ed.) Theoretical computer science. (Lect. Notes Comput. Sci., vol. 104, pp. 196–223) Berlin Heidelberg New York: Springer 1981
[Plo81] Plotkin, G.: A structural approach to operational semantics. Technical report DAIMI FN-19, Computer Science Department, Aarhus University 1981
[San92] Sangiorgi, D.: Forthcoming Ph. D. thesis, Computer Science Department, Edinburgh University 1992
[Tar55] Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pac. J. Math. 5, 285–309 (1955)
[Tho89] Thomsen, B.: A calculus of higher order communicating systems. Proceedings of POPL 89, pp. 143–154. The Association for Computing Machinery 1989
[Tho89a] Thomsen, B.: Plain CHOCS. Technical report 89/4, Department of Computing, Imperial College, London University 1989
[Tho90] Thomsen, B.: Calculi for higher order communicating systems. Ph.D. Thesis, Imperial College, London University 1990
[Wal88] Walker, D.: Bisimulation and divergence. Proceedings of LICS 88, pp. 186–192. Oxford: Computer Society Press 1988
[Wal91] Walker, D.: π-Calculus semantics of object-oriented programming languages. Proceedings of Conference on Theoretical Aspects of Computer Software (Lect. Notes Comput. Sci., vol. 526, pp. 532–547) Berlin Heidelberg New York: Springer 1991

Appendix A

Lemma A.1 (Lemma 3.4) *If R is an applicative higher order bisimulation up to restriction then $R \subseteq \dot{\sim}$.*

Proof. We show that the relation $R^{\backslash} = \bigcup_{n \in \omega} R_n$ where

$$R_0 = R$$
$$R_{n+1} = \{(p\backslash b, q\backslash b) : (p, q) \in R_n, b \in Names\}$$

is an applicative higher order bisimulation.

First we show by induction on n that if $pR_n q$ and $c \notin fn(p) \cup fn(q)$ then $\{c/a\}\, pR_n \{c/a\}\, q$.

For $n=0$ this is immediate from the definition of applicative higher order bisimulation up to restriction. Suppose $n>0$ and $p\backslash bR_n q\backslash b$ where $pR_{n-1} q$ and $c \notin fn(p\backslash b) \cup fn(q\backslash b)$. If $a=b$ then $\{c/a\}(p\backslash b) \equiv p\backslash bR^{\backslash} q\backslash b \equiv \{c/a\}(q\backslash b)$. If $a \neq b$ then $\{c/a\}(p\backslash b) \equiv (\{c/a\}(\{b_1/b\}\, p))\backslash b_1\, R^{\backslash} (\{c/a\}(\{b_1/b\}\, q))\backslash b_1 \equiv \{c/a\}(q\backslash b)$. Next we show by induction on n that if $pR_n q$ then

(i) Whenever $p \xrightarrow{a?x} p'$, then $q \xrightarrow{a?y} q'$ for some q', y and $p'[r/x]\, R^{\backslash} q'[r/y]$ for all $r \in CPr$

(ii) Whenever $p \xrightarrow{a!_B p'} p''$ with $B \cap (fn(p) \cup fn(q)) = \emptyset$, then $q \xrightarrow{a!_B q'} q''$ for some q', q'' with $p' R^{\backslash} q'$ and $p'' R^{\backslash} q''$

(iii) Whenever $p \xrightarrow{\tau} p'$, then $q \xrightarrow{\tau} q'$ for some q' and $p' R^{\backslash} q'$

$n=0$ This case is immediate from the fact that R_0 is an applicative higher order bisimulation up to restriction and from the definition of $R^{\backslash}$.

$n>0$ Suppose $p R_n q$ where $p \equiv p_1 \backslash b$ and $q \equiv q_1 \backslash b$.

1. If $p \xrightarrow{a?x} p'$ this must have been inferred by the res-rule and $p_1 \xrightarrow{a?x} p'_1$ with $a \neq b$ and $p' \equiv p'_1 \backslash b$. Then for some q'_1, y we have $q_1 \xrightarrow{a?y} q'_1$ and $p'_1[r/x]\, R^{\backslash} q'_1[r/y]$ for all $r \in CPr$. Then $q \xrightarrow{a?y} q' \equiv q'_1 \backslash b$ and for all $r \in CPr$ and some $c \notin fn(p'_1 \backslash b) \cup fn(q'_1 \backslash b) \cup fn(r)$ we have

$$p'[r/x] \equiv (((\{c/b\}\, p'_1)[r/x]) \backslash c\, R^{\backslash} (((\{c/b\}\, q'_1)[r/x]) \backslash c \equiv q'[r/x].$$

2. Suppose $B \cap (fn(p) \cup fn(q)) = \emptyset$. If $p \xrightarrow{a!_B p'} p''$ and this has been inferred by the res-rule then $p_1 \xrightarrow{a!_B p'_1} p''_1$ with $a \neq b$ and $b \notin B \cup fn(p'_1)$ and $p' \equiv p'_1$ and $p'' \equiv p''_1 \backslash b$. So for some q'_1, q''_1 we have $q_1 \xrightarrow{a!_B q'_1} q''_1$ and $p'_1 R^{\backslash} q'_1$ and $p''_1 R^{\backslash} q''_1$. Thus $q \xrightarrow{a!_B q'} q''$ with $q' \equiv q'_1$ and $q'' \equiv q''_1 \backslash b$ and $p' R^{\backslash} q'$ and $p'' R^{\backslash} q''$.

If $p \xrightarrow{a!_B p'} p''$ and this has been inferred by the open-rule then $p_1 \xrightarrow{a!_{B'} p'_1} p''_1$ with $a \neq b$, $B = B' \cup \{c\}$ and $b \in fn(p'_1)$ and $c \notin fn(p_1 \backslash b) \cup B'$ and $p' \equiv \{c/b\}\, p'_1$ and $p'' \equiv \{c/b\}\, p''_1$. So for some q'_1, q''_1 we have $q_1 \xrightarrow{a!_{B'} q'_1} q''$ and $p'_1 R \backslash q'_1$ and $p''_1 R \backslash q''_1$. Thus $q \xrightarrow{a!_B q'} q''$ with $q' \equiv \{c/b\}\, q'_1$ and $q'' \equiv \{c/b\}\, q''_1$ and $p' R \backslash q'$ and $p'' R^{\backslash} q''$.

If $p \xrightarrow{a!_B p'} p''$ and this has been inferred by the non-struct-rule then $p \xrightarrow{a!_{B'} p'} p''$ with $B' \cap (fn(p') \cup fn(p'')) = B \cap (fn(p') \cup fn(p''))$. So for some q', q'' we have $q \xrightarrow{a!_{B'} q'} q''$ and $p' R^{\backslash} q'$ and $p'' R^{\backslash} q''$. Thus $B' \cap (fn(q') \cup fn(q'')) = B \cap (fn(q') \cup fn(q''))$ and $q \xrightarrow{a!_B q'} q''$ and we already know $p' R^{\backslash} q'$ and $p'' R^{\backslash} q''$.

3. If $p \xrightarrow{\tau} p'$ then $p_1 \xrightarrow{\tau} p'_1$ and $p' \equiv p'_1 \backslash b$. Then $q_1 \xrightarrow{\tau} q'_1$ and $p'_1 R^{\backslash} q'_1$. Thus $q \xrightarrow{\tau} q' \equiv q'_1$ and $p' R^{\backslash} q'$. □

Proposition A.2 (*Proposition 3.6*) $\dot{\sim}$ *is a congruence relation on processes* (*closed expressions*).

1. $p[\bar{q}_1/\bar{x}] \dot{\sim} p[\bar{q}_2/\bar{x}]$ *if* $\bar{q}_1 \dot{\sim} \bar{q}_2$ *and* $\bar{x} \subseteq FV(p)$
2. $a?x.p \dot{\sim} a?x.q$ *if* $p[r/x] \dot{\sim} q[r/x]$ *for all* r
3. $a!p'.p \dot{\sim} a!q'.q$ *if* $p \dot{\sim} q$ *and* $p' \dot{\sim} q'$
4. $\tau.p \dot{\sim} \tau.q$ *if* $p \dot{\sim} q$
5. $p + p' \dot{\sim} q + q'$ *if* $p \dot{\sim} q$ *and* $p' \dot{\sim} q'$
6. $p \mid p' \dot{\sim} q \mid q'$ *if* $p \dot{\sim} q$ *and* $p' \dot{\sim} q'$
7. $p \backslash a \dot{\sim} q \backslash a$ *if* $p \dot{\sim} q$
8. $p[S] \dot{\sim} q[S]$ *if* $p \dot{\sim} q$.

Proof. 1. We prove this by showing that the relation ACR^*, the reflexive and transitive closure of ACR, where

$$ACR = \{(p[\bar{q}_1/\bar{x}], p[\bar{q}_2/\bar{x}]) : p \in Pr \;\&\; \bar{x} \subseteq FV(p) \;\&\; \bar{q}_1 \dot{\sim} \bar{q}_2 \;\&\; \bar{q}_i \in CPr\},$$

is an applicative higher order bisimulation up to restriction.

Note if $q_1 \dot{\sim} q_2$ then $(x[q_1/x], x[q_2/x]) \in ACR^*$ and we write $(q_1, q_2) \in ACR^*$.

First we show a few useful lemmas about ACR and ACR^*:

Lemma A.3 *If* $(p_1', p_1'') \in ACR$ *and* $(p_2', p_2'') \in ACR$ *then* $(p_1' | p_2', p_1'' | p_2'') \in ACR$

Proof. Since $(p_1', p_1'') \in ACR$ there exist p_3, $\bar{q}_1^1$, $\bar{q}_2^1$ and $\bar{x}^1$ such that $p_1' \equiv p_3[\bar{q}_1^1/\bar{x}^1]$ and $p_1'' \equiv p_3[\bar{q}_2^1/\bar{x}^1]$ with $FV(p_3) \subseteq \bar{x}^1$ and $\bar{q}_1^1 \dot{\sim} \bar{q}_2^1$. Also, since $(p_2', p_2'') \in ACR$ there exist p_4, $\bar{q}_1^2$, $\bar{q}_2^2$ and $\bar{x}^2$ such that $p_2' \equiv p_4[\bar{q}_1^2/\bar{x}^2]$ and $p_2'' \equiv p_4[\bar{q}_2^2/\bar{x}^2]$ with $FV(p_4) = \bar{x}^2$ and $\bar{q}_1^2 \dot{\sim} \bar{q}_2^2$. We may assume $\bar{x}^1 \cap \bar{x}^2 = \emptyset$ since if $\bar{x}^1 \cap \bar{x}^2 \neq \emptyset$ we proceed by choosing $\bar{y}$ such that $\bar{y} \cap (FV(p_3) \cup FV(p_4) \cup \bar{x}^1 \cup \bar{x}^2) = \emptyset$ and we have $p_3[\bar{q}_i^1/\bar{x}^1] \equiv (p_3[\bar{y}/\bar{x}^1])[\bar{q}_i^1/\bar{y}]$ by Proposition 2.5. Therefore we have $p_1' | p_2' \equiv (p_3[\bar{q}_1^1/\bar{x}^1]) | (p_4[\bar{q}_1^2/\bar{x}^2]) \equiv (p_3 | p_4)[\bar{q}_1^1 \cup \bar{q}_1^2/\bar{x}^1 \cup \bar{x}^2]$ and $p_1'' | p_2'' \equiv (p_3[\bar{q}_2^1/\bar{x}^1]) | (p_4[\bar{q}_2^2/\bar{x}^2]) \equiv (p_3 | p_4)[\bar{q}_2^1 \cup \bar{q}_2^2/\bar{x}^1 \cup \bar{x}^2]$ and $(p_1' | p_2', p_1'' | p_2'') \in ACR$. (Note that if we have introduce a "new" $\bar{y}$ it is because two or more occurrences of the same x_i refer to different q_i's after the transition.) □

Lemma A.4 *If* $(p_1', p_1'') \in ACR^*$ *and* $(p_2', p_2'') \in ACR$ *then* $(p_1' | p_2', p_1'' | p_2'') \in ACR^*$

Proof. Since $(p_1', p_1'') \in ACR^*$ there is a sequence $q_1 \ldots q_n$ such that $(q_i, q_{i+1}) \in ACR$ for $1 \leq i < n$ with $p_1' \equiv q_1$ and $p_1'' \equiv q_n$. Thus for each pair $(q_i, q_{i+1}) \in ACR$ we may apply Lemma A.3 and conclude $(q_i | p_2', q_{i+1} | p_2'') \in ACR$ thus $(p_1' | p_2', p_1'' | p_2'') \in ACR^*$. □

Lemma A.5 *If* $(p_1', p_1'') \in ACR^*$ *and* $(p_2', p_2'') \in ACR^*$ *then* $(p_1' | p_2', p_1'' | p_2'') \in ACR^*$

Proof. Since $(p_2', p_2'') \in ACR^*$ there is a sequence $q_1 \ldots q_n$ such that $(q_i, q_{i+1}) \in ACR$ for $1 \leq i < n$ with $p_2' \equiv q_1$ and $p_2'' \equiv q_n$. Thus for each pair $(q_i, q_{i+1}) \in ACR$ we may apply Lemma A.4 and conclude $(p_1' | q_i, p_1'' | q_{i+1}) \in ACR^*$ thus $(p_1' | p_2', p_1'' | p_2'') \in ACR^*$. □

Lemma A.6 *If* $(r', r'') \in ACR$ *then* $(p_1''[r'/z], p_1''[r''/z]) \in ACR$.

Proof. If $(r', r'') \in ACR$ then $r' \equiv r_1[\bar{q}_1/\bar{x}]$ and $r'' \equiv r_1[\bar{q}_2/\bar{x}]$ for some r_1 with $FV(r_1) \subseteq \bar{x}$ and $\bar{q}_1 \dot{\sim} \bar{q}_2$ for some closed $\bar{q}_i$. Then $p_1''[r'/z] \equiv p_1''[r_1[\bar{q}_1/\bar{x}]/z] \equiv (p_1''[r_1/z])[\bar{q}_1/\bar{x}]$ and $p_1''[r''/z] \equiv p_1''[r_1[\bar{q}_2/\bar{x}]/z] \equiv (p_1''[r_1/z])[\bar{q}_2/\bar{x}]$ since $FV(p_1'') = \{z\}$ and $FV(r_1) = \bar{x}$ we have $FV(p_1''(r_1/z]) = \bar{x}$ and $((p_1''[r_1/z])[\bar{q}_1/\bar{x}], (p_1''[r_1/y])[\bar{q}_2/\bar{x}]) \in ACR$. Thus $(p_1''[r'/z], p_1''[r''/z]) \in ACR$. □

Lemma A.7 *If* $(r', r'') \in ACR^*$ *then* $(p_1''[r'/z], p_1''[r''/z]) \in ACR^*$.

Proof. Since $(r', r'') \in ACR^*$ there is a sequence $q_1 \ldots q_n$ such that $(q_i, q_{i+1}) \in ACR$ for $1 \leq i < n$ with $r' \equiv q_1$ and $r'' \equiv q_n$. Thus for each pair $(q_i, q_{i+1}) \in ACR$ we may apply Lemma A.7 and conclude $(p_1''[q_i/z], p_1''[q_{i+1}z]) \in ACR$ thus $(p_1''[r'/z], p_1''[r''/z]) \in ACR^*$. □

Lemma A.8 *If* $(p_1', p_2') \in ACR$ *then* $(p_1' \backslash b, p_2' \backslash b) \in ACR$.

Proof. Since $(p'_1, p'_2) \in ACR$ there exist p_3, $\bar{q}_1$, $\bar{q}_2$ and $\bar{x}$ for each r such that $p'_1 \equiv p_3[\bar{q}_1/\bar{x}]$ and $p'_2 \equiv p_3[\bar{q}_2/\bar{x}]$ with $FV(p_3) \subseteq \bar{x}$ and $\bar{q}_1 \dot{\sim} \bar{q}_2$ and $(p'_1 \backslash b) \equiv (p_3 \backslash b)[\bar{q}_1/\bar{x}]$ and $(p'_2 \backslash b) \equiv (p_3 \backslash b)[\bar{q}_2/\bar{x}]$ thus $(p'_1 \backslash b, p'_2 \backslash b) \in ACR$. □

Lemma A.9 *If* $(p'_1, p'_2) \in ACR^*$ *then* $(p'_1 \backslash b, p'_2 \backslash b) \in ACR^*$.

Proof. Follows the pattern of the proof of Lemma A.7. □

We now return to the main proof. We only show that ACR^* is an applicative higher order simulation up to restriction, symmetry of ACR^* then yields the result. To see that ACR^* is an applicative higher order simulation up to restriction we show that if $(p_1, p_2) \in ACR$ then $p_i \equiv p[\bar{q}_i/\bar{x}]$ and:

(i) If $b \notin fn(p[\bar{q}_1/\bar{x}]) \cup fn(p[\bar{q}_2/\bar{x}])$ then $\{b/a\}(p[\bar{q}_1/x])\ ACR^*\ \{b/a\}(p[\bar{q}_2/\bar{x}])$

(ii) Whenever $p[\bar{q}_1/\bar{x}] \xrightarrow{a?x} p'$, then $p[\bar{q}_2/\bar{x}] \xrightarrow{a?y} q'$ for some q', y and $p'[r/x]\ ACR^*\ q'[r/y]$ for all $r \in CPr$

(iii) Whenever $p[\bar{q}_1/\bar{x}] \xrightarrow{a!_B p'} p''$ with $B \cap (fn(p) \cup fn(q)) = \emptyset$, then $p[\bar{q}_2/\bar{x}] \xrightarrow{a!_B q'} q''$ for some q', q'' with $p'\ ACR^*\ q'$ and $p''\ ACR^*\ q''$

(iv) Whenever $p[\bar{q}_1/\bar{x}] \xrightarrow{\tau} p'$, then $p[\bar{q}_2/\bar{x}] \xrightarrow{\tau} q'$ for some q' and either $p'\ ACR^*\ q'$ or for some p'', q'' and b: $p' \equiv p'' \backslash b$, $q' \equiv q'' \backslash b$ and $p''\ ACR^*\ q''$.

If $(p, q) \in ACR^*$ then there is a sequence $p_1 \ldots p_n$ such that $(p, p_1) \in ACR$, $(p_i, p_{i+1}) \in ACR$ for $1 \leq i < n$ and $(p_n, q) \in ACR$. The result then follows by induction on the length of the transitive sequence $p_1 \ldots p_n$ of ACR^*.

First (i) is easily proved by structural induction on p using Lemma 3.5 in the case $p \equiv y$.

Next we show (ii)–(iv) simultaneously. We proceed by induction on the length of the inference used to establish the transitions of $p[\bar{q}_1/\bar{x}]$ and cases of the structure of p. We only need to consider transitions inferred by use of the structural rules since we may transform any derivation of a transition into an equivalent one where we use the non-struct-rule exactly once after each application of a structural rule.

$p \equiv nil$ Trivial since $p[\bar{q}_i/\bar{x}] \not\rightarrow$.

$p \equiv a?y.p_1$ Assume $y \notin \bar{x}$ (otherwise use α-conversion on y). Then $p[\bar{q}_i/\bar{x}] \equiv a?y.(p_1[\bar{q}_i/\bar{x}])$ and $p[\bar{q}_i/\bar{x}] \xrightarrow{a?y} p_1[\bar{q}_i/\bar{x}]$. Since $FV(p_1) \subseteq (\bar{x} \cup \{y\})$ and $y \notin \bar{x}$ we have

$$(p_1[\bar{q}_1/\bar{x}])[r/y] \equiv p_1[\bar{q}_1, r/\bar{x}, y]\ ACR^*\ p_1[\bar{q}_2, r/\bar{x}, y] \equiv (p_1[\bar{q}_2/\bar{x}])[r/y]$$

for all $r \in CPr$, since $r \dot{\sim} r$ and $\bar{q}_i$ are closed.

$p \equiv a!p_1.p_2$ Then $p[\bar{q}_i/\bar{x}] \xrightarrow{a!_\emptyset(p_1[\bar{q}_i/\bar{x}])} (p_2[\bar{q}_i/\bar{x}])$ and $p_2[\bar{q}_1/\bar{x}]\ ACR^*\ p_2[\bar{q}_2/\bar{x}]$ and $p_1[\bar{q}_1/\bar{x}]\ ACR^*\ p_1[\bar{q}_2/\bar{x}]$

$p \equiv \tau.p_1$ An argument similar to the argument given in the case above yields this case.

$p \equiv p_1 + p_2$ If $p[\bar{q}_1/\bar{x}] \xrightarrow{\Gamma} p'$ then

either $p_1[\bar{q}_1/\bar{x}] \xrightarrow{\Gamma} p'$ by a shorter inference. There are three cases depending on the structure of Γ. We show the case when $\Gamma = a?x$: By induction $p_1[\bar{q}_2/\bar{x}] \xrightarrow{a?z} p''$ and $p'[r/x]\ ACR^*\ p''[r/z]$ for all

$r \in CPr$. By the operational semantics for choice we have $(p_1+p_2)[\bar{q}_2/\bar{x}] \xrightarrow{a?z} p''$ which is a matching move.

or $p_2[\bar{q}_2/\bar{x}] \xrightarrow{\Gamma} p'$ and we may argue as above.

$p \equiv p_1 | p_2$ If $p[\bar{q}_1/\bar{x}] \xrightarrow{\Gamma} p'$ then

either $p_1[\bar{q}_1/\bar{x}] \xrightarrow{\Gamma} p_1'$ by a shorter inference and $p' \equiv p_1' | p_2[\bar{q}_1/\bar{x}]$. There are three cases depending on the structure of Γ:

$\Gamma = a?x$ Then by induction $p_1[\bar{q}_2/\bar{x}] \xrightarrow{a?z} p_1''$ and $(p_1'[r/x], p_1''[r/z]) \in ACR^*$ for all $r \in CPr$. Then by the operational semantics for parallel we have $(p_1|p_2)[\bar{q}_2/\bar{x}] \equiv (p_1[\bar{q}_2/\bar{x}])|(p_2[\bar{q}_2/\bar{x}]) \xrightarrow{a?z} p_1''|p_2[\bar{q}_2/\bar{x}]$. We have $(p_1'|p_2[\bar{q}_1^2/\bar{x}])[r/x] \equiv p_1'[r/x]|p_2[\bar{q}_1^2/\bar{x}]$ and $(p_1''|p_2[\bar{q}_2^2/\bar{x}])[r/z] \equiv p_1''[r/z]|p_2[\bar{q}_2^2/\bar{x}]$ for each $r \in CPr$ by Proposition 2.5 since r and q_i^1 and q_i^2 are all closed. Since $(p_1'[r/x], p_1''[r/z]) \in ACR^*$ for all $r \in CPr$ and $p_2[\bar{q}_2^2/\bar{x}], p_2[\bar{q}_2^2/\bar{x}] \in ACR^*$ we may apply Lemma A.5 and conclude $((p_1'|p_2[\bar{q}_1^2/\bar{x}])[r/x], (p_1''|p_2[\bar{q}_2^2/\bar{x}])[r/z]) \in ACR^*$ for all $r \in CPr$.

$\Gamma = a!_B p'$ Then $B \cap fn(p_2[\bar{q}_1/\bar{x}]) = \emptyset$. By induction $p_1[\bar{q}_2/\bar{x}] \xrightarrow{a!_B p''} p_1''$ with $(p', p'') \in ACR^*$ and $(p_1', p_1'') \in ACR^*$ and $B \cap (fn(p_1[\bar{q}_1/\bar{x}]) \cup fn(p_1[\bar{q}_2/\bar{x}])) = \emptyset$. Thus $B \cap fn(p_2[\bar{q}_2/\bar{x}]) = \emptyset$ and by the operational semantics for parallel $p_1[\bar{q}_2/\bar{x}]|p_2[\bar{q}_2/\bar{x}] \xrightarrow{a!_B p''} p_1''|p_2[\bar{q}_2/\bar{x}]$. Since $(p_1', p_1'') \in ACR^*$ and $p_2[\bar{q}_2^2/\bar{x}], p_2[\bar{q}_2^2/\bar{x}] \in ACR^*$ we may apply Lemma A.5 and conclude $((p_1'|p_2[\bar{q}_1^2/\bar{x}]), (p_1''|p_2[\bar{q}_2^2/\bar{x}])) \in ACR^*$.

$\Gamma = \tau$ and we may argue as above.

or $p_2[\bar{q}_1/\bar{x}] \xrightarrow{\Gamma} p_2'$ and we may argue as above.

or $\Gamma = \tau$ and w.l.o.g. $p_1[\bar{q}_1/\bar{x}] \xrightarrow{a?x} p_1'$ and $p_2[\bar{q}_1/\bar{x}] \xrightarrow{a!_B r'} p_2'$ by shorter inferences and $p' \equiv (p_1'[r'/x]|p_2')\backslash B$ and $B \cap fn(p_1') = \emptyset$. By induction $p_2[\bar{q}_2/\bar{x}] \xrightarrow{a!_B r''} p_2''$ with $(r', r'') \in ACR^*$ and $(p_2', p_2'') \in ACR^*$ and $p_1[\bar{q}_2/\bar{x}] \xrightarrow{a?z} p_1''$ with $(p_1'[r/x], p_1''[r/z]) \in ACR^*$ for all $r \in CPr$. By proposition 2.7 we may assume that $B \cap fn(p_1'') = \emptyset$. By the operational semantics for parallel $(p_1|p_2)[\bar{q}_2/\bar{x}] \xrightarrow{\tau} (p_1''[r''/z]|p_2'')\backslash B$. To see that $p_1'[r'/x]|p_2' \; ACR^* \; p_1''[r''/z]|p_2''$ and thus showing that ACR^* is an applicative higher order bisimulation up to restriction we observe that $(p_1'[r/x], p_1''[r/z]) \in ACR^*$ for all $r \in CPr$, in particular this is true for r'. By Lemma A.7 we have $p_1''[r'/z] \; ACR^* \; p_1''[r''/z]$ since $r' \; ACR^* \; r''$. Thus $(p_1'[r'/x], p_1''[r''/z]) \in ACR^*$. Since also $(p_2', p_2'') \in ACR^*$ we may apply Lemma A.5 and conclude $(p_1'[r'/x]|p_2', p_1''[r''/z]|p_2'') \in ACR^*$.

$p \equiv p_1\backslash b$ Then $p[\bar{q}_i/\bar{x}] \equiv (((\{d_i/b\}\, p_1)[\bar{q}_i/\bar{x}])\backslash d_i$ for some $d_i \notin fn(p_1\backslash b) \cup fn(\bar{q}_i)$. By (i) we may assume $b = d_1 = d_2 \notin fn(p_1\backslash b) \cup fn(\bar{q}_1) \cup fn(\bar{q}_2)$. If $p[\bar{q}_1/\bar{x}] \xrightarrow{\Gamma} p''$ then if

$\Gamma = a?x$ Then $p_1[\bar{q}_1/\bar{x}] \xrightarrow{a?x} p_1'$ by a shorter inference and $p' \equiv p_1'\backslash b$ and $a \neq b$. By induction $p_1[\bar{q}_2/\bar{x}] \xrightarrow{a?z} p_2'$ and $(p_1'[r/x], p_2'[r/z]) \in ACR^*$ for all $r \in CPr$. By the operational semantics for restriction

$(p_1 \backslash b)[\bar{q}_2/\bar{x}] \xrightarrow{a?z} p_2' \backslash b$. Since $(p_1'[r/x], p_2'[r/z]) \in ACR^*$ for all $r \in CPr$ we may apply Lemma A.9 and conclude $((p_1' \backslash b)[r/x], (p_2' \backslash b)[r/z]) \in ACR^*$ assuming $r \notin fn(r)$ (otherwise use α-conversion).

$\Gamma = a!_B p'$ Then

either $p_1[\bar{q}_1/\bar{x}] \xrightarrow{a!_B p_1'} p_1''$ by a shorter inference and $p' \equiv p_1'$, $p'' \equiv p_1'' \backslash b$, $b \neq a$, $b \notin B$, $b \notin fn(p_1')$. Then by induction $p_1[\bar{q}_2/\bar{x}] \xrightarrow{a!_B p_2'} p_2''$ with $(p_1', p_2') \in ACR^*$ and $(p_1'', p_2'') \in ACR^*$ and $B \cap (fn(p_1[\bar{q}_1/\bar{x}]) \cup fn(p_1[\bar{q}_2/\bar{x}])) = \emptyset$. Then by the res-rule we have $(p_1[\bar{q}_2/\bar{x}]) \backslash b \xrightarrow{a!_B p_2'} p_2'' \backslash b$ and we may argue as above that $(p_1'' \backslash b, p_2'' \backslash b) \in ACR^*$.

or $p_1[\bar{q}_1/\bar{x}] \xrightarrow{a!_{B'} p_1'} p_1''$ by a shorter inference and $p' \equiv \{d/b\}\, p_1'$, $p'' \equiv \{d/b\}\, p_1''$, $b \neq a$, $b \notin B'$, $b \in fn(p_1')$, $B = B' \cup \{d\}$, $d \notin B' \cup fn((p_1[\bar{q}_i/\bar{x}]) \backslash b)$. Then by induction $p_1[\bar{q}_2/\bar{x}] \xrightarrow{a!_{B'} p_2'} p_2''$ with $(p_1', p_2') \in ACR^*$ and $(p_1'', p_2'') \in ACR^*$ and $B' \cap (fn(p_1[\bar{q}_1/\bar{x}]) \cup fn(p_1[\bar{q}_2/\bar{x}])) = \emptyset$. If $b \in fn(p_2') \cap fn(p_2'')$ then by the open-rule we have $(p_1[\bar{q}_2/\bar{x}]) \backslash b \xrightarrow{a!_B \{d/b\} p_2'} \{d/b\}\, p_2''$ and by (i) we have $(\{d/b\}\, p_1', \{d/b\}\, p_2') \in ACR^*$ and $(\{d/b\}\, p_1'', \{d/b\}\, p_2'') \in ACR^*$. If $b \notin fn(p_2') \cap fn(p_2'')$ then by the non-struct-rule we have $(p_1[\bar{q}_2/\bar{x}]) \backslash b \xrightarrow{a!_B p_2'} p_2''$ and $p_2' \equiv \{d/b\}\, p_2'$ and $p_2'' \equiv \{d/b\}\, p_2''$ and by (i) we have $(\{d/b\}\, p_1', \{d/b\}\, p_2') \in ACR^*$ and $(\{d/b\}\, p_1'', \{d/b\}\, p_2'') \in ACR^*$.

$\Gamma = \tau$ and we may argue as above.

$p \equiv p_1[S]$ If $p[\bar{q}_1/\bar{x}] \equiv (p_1[\bar{q}_1/\bar{x}])[S] \xrightarrow{\Gamma} p''$ then if

$\Gamma = a?x$ we have $p_1[\bar{q}_1/\bar{x}] \xrightarrow{b?x} p_1''$ by a shorter inference and $a = S(b)$ and $p'' \equiv p_1''[S]$. By induction $p_1[\bar{q}_2/\bar{x}] \xrightarrow{b?z} p_2''$ and $p_1'[r/x]\ ACR^*\ p_2''[r/z]$ for all $r \in CPr$. Then $(p_1[\bar{q}_1/\bar{x}])[S] \xrightarrow{a?z} p_2''[S]$ with $(p_1''[r/x])[S] \equiv (p_1''[S])[r/x]\ ACR^*\ (p_2''[S])[r/z] \equiv (p_2''[r/z])[S]$ for all $r \in CPr$.

$\Gamma = a!_B p'$ we have $p_1[\bar{q}_1/\bar{x}] \xrightarrow{b!_B p_1'} p_1''$ by a shorter inference and $a = S(b)$ and $B \cap (Dom(S) \cup Im(S)) = \emptyset$ and $p' \equiv p_1'$ and $p'' \equiv p_1''[S]$. By induction $p_1[\bar{q}_2/\bar{x}] \xrightarrow{b!_B p_2'} p_2''$ and $p_1'\ ACR^*\ p_2'$ and $p_1''\ ACR^*\ p_2''$. Then $(p_1[\bar{q}_1/\bar{x}])[S] \xrightarrow{a!_B p_2'} p_2''[S]$ with $p_1'\ ACR^*\ p_2'$ and $p_1''[S]\ ACR^*\ p_2''[S]$.

$\Gamma = \tau$ this case is similar to the above.

$p \equiv y$ By assumption $FV(p) \subseteq \bar{x}$ thus $\bar{x} = (y)$ and if $p[\bar{q}_1/\bar{x}] \equiv q_1 \xrightarrow{\Gamma} q_1'$ then if

$\Gamma = a?x$ we have $p[\bar{q}_2/\bar{x}] \equiv q_2 \xrightarrow{a?z} q_2'$ for some q_2' and z. Since $q_1 \dot{\sim} q_2$ we have $(q_1'[r/x], q_2'[r/z]) \in \dot{\sim}$ for all $r \in CPr$ and thus $(q_1'[r/x], q_2'[r/z]) \in ACR^*$ for all $r \in CPr$

$\Gamma = a!_B p'$ we have $p[\bar{q}_2/\bar{x}] \equiv q_2 \xrightarrow{a!_B q_2'} q_2'$ for some q_2' and q_2''. Since $q_1 \dot{\sim} q_2$ we have $(q_1', q_2') \in \dot{\sim}$ and $(q_1'', q_2'') \in \dot{\sim}$ and thus $(q_1', q_2') \in ACR^*$ and $(q_1'', q_2'') \in ACR^*$

$\Gamma = \tau$ A similar argument as above applies.

Thus in each case we have a matching move for $p[\bar{q}_2/\bar{x}]$.

2. This is proved by showing that the relation $R_1 = R \cup \dot{\sim}$, where:

$$R = \{(a?x.p, a?x.q) : FV(p) = FV(q) \subseteq \{x\}\ \&\ \forall r \in CPr.\, p[r/x] \dot{\sim} q[r/x]\}$$

is an applicative higher order bisimulation. Note that the relation R_1 consists of two parts; one part covers the structure we are interested in and the second component is a kind of closure to cover the processes sent and received. The second component is necessary since the processes sent and received do not necessarily have the structure of the first part.

That the above relation is indeed an applicative higher order bisimulation is easily established. Assume $(p, q) \in R_1$. Then

either $p \dot{\sim} q$ and we are done since if $p \xrightarrow{\Gamma} p'$ then $q \xrightarrow{\Gamma'} q'$ for some q', Γ'. If $\Gamma = a?x$ then $\Gamma' = a?y$ and for all $r \in CPr$ we have $(p'[r/x], q'[r/y]) \in \dot{\sim} \subseteq R_1$. If $\Gamma = a!_B p''$ then $\Gamma' = a!_B q''$ and we have $B \cap (fn(p) \cup fn(q)) = \emptyset$ and $(p'', q'') \in \dot{\sim} \subseteq R_1$ and $(p', q') \in \dot{\sim} \subseteq R_1$. If $\Gamma = \tau$ then $\Gamma' = \tau$ and we have $(p', q') \in \dot{\sim} \subseteq R_1$.

or $p \equiv a?x.p'$ and $q \equiv a?x.q'$. If $a?x.p' \xrightarrow{\Gamma} p'$ then $\Gamma = a?x$. Then $a?x.q' \xrightarrow{a?x} q'$ and by assumption $p'[r/x] \dot{\sim} q'[r/x]$ for all $r \in CPr$ which implies $(p'[r/x], q'[r/x]) \in R_1$.

3. follows from $((a!x.y)[(p, p')/(x, y)], (a!x.y)[(q, q')/(x, y)]) \in ACR$ if $p \dot{\sim} q$ and $p' \dot{\sim} q'$ and $x \neq y$.

4. follows from $((\tau.x)[p/x], (\tau.x)[q/x]) \in ACR$ if $p \dot{\sim} q$.

5. follows from $((x+y)[(p, p')/(x, y)], (x+y)[(q, q')/(x, y)]) \in ACR$ if $p \dot{\sim} q$ and $p' \dot{\sim} q'$ and $x \neq y$.

6. follows from $((x|y)[(p, p')/(x, y)], (x|y)[(q, q')/(x, y)]) \in ACR$ if $p \dot{\sim} q$ and $p' \dot{\sim} q'$ and $x \neq y$.

7. follows from $(x[p/x], x[q/x]) \in ACR$ if $p \dot{\sim} q$ and the fact that ACR^* is an applicative bisimulation up to restriction.

8. follows from $((x[S])[p/x], (x[S])[q/x]) \in ACR$ if $p \dot{\sim} q$. □

Proposition A.10 (Proposition 3.14) *$p_1 \backslash a | p_2 \dot{\sim} (p_1 | p_2) \backslash a$ if $a \notin fn(p_2)$*

Proof. This proposition is proved by showing that the relation

$$R = \{(p_1 \backslash a | p_2, (p_1 | p_2) \backslash a) : p_i \in CPr, a \notin fn(p_2)\} \cup Id$$

is an applicative higher order bisimulation up to $\dot{\sim}$ and restriction. To see this we show that when $(p, q) \in R$ and $p \xrightarrow{\Gamma} p'$ then $q \xrightarrow{\Gamma'} q'$ with a move which satisfies the conditions of applicative higher order bisimulation up to $\dot{\sim}$ and restriction. If $(p, q) \in Id$ the case is obvious so assume that $p \equiv p_1 \backslash a | p_2$ and $q \equiv (p_1 | p_2) \backslash a$ and $a \notin fn(p_2)$.

If $p \xrightarrow{\Gamma} p'$ this transition must have been inferred in the following way:

either this has been inferred from the par-rule and $p_2 \xrightarrow{\Gamma} p_2''$ and $p' \equiv p_1 \backslash a | p_2''$. There are three cases:

$\Gamma = b?x$ then $b \neq a$ since $a \notin fn(p_2)$. Then by the par-rule and the res-rule we have $(p_1 | p_2) \backslash a \xrightarrow{b?x} (p_1 | p_2'') \backslash a$ and for all $r \in CPr$ we have $(p_1 \backslash a | p_2'')[r/x] \dot{\sim} (\{d/a\}\, p_1) \backslash b | p_2''[r/x]\ R\ (\{d/a\}\, p_1 | p_2''[r/x]) \backslash d \dot{\sim} ((p_1 | p_2'') \backslash a)[r/x]$ for some $d \notin fn(p_1) \cup fn(p_2) \cup fn(r)$.

$\Gamma = b!_B p_2'$ then $b \neq a$ and we may assume $B \cap (\{a\} \cup fn(p_1)) = \emptyset$. Then by the par-rule and the res-rule we have $(p_1 | p_2)\backslash a \xrightarrow{b!_B p_2'} (p_1 | p_2'')\backslash a$ which is a matching move.

$\Gamma = \tau$ and we may argue as in the above case.

or this transition has been inferred by the par-rule and $p_1 \backslash a \xrightarrow{\Gamma} p_1''$ and this has been inferred from the res-rule and $p_1 \xrightarrow{\Gamma} p_1'''$ and $p' \equiv p_1'' | p_2$. There are three cases:

$\Gamma = b?x$ then $p_1'' \equiv p_1'''\backslash a$ and $b \neq a$. Then by the par-rule and the res-rule we have $(p_1 | p_2)\backslash a \xrightarrow{b?x} (p_1''' | p_2)\backslash a$. This is a matching move since for all $r \in CPr$ we have
$(p_1'''\backslash a | p_2)[r/x] \dot{\sim} ((\{d/a\}\, p_1''')[r/x])\backslash b | p_2\, R\, ((\{d/a\}\, p_1''')[r/x] | p_2)\backslash d$
$\dot{\sim} ((p_1''' | p_2)\backslash a)[r/x]$ for some $d \notin fn(p_1) \cup fn(p_2) \cup fn(r)$.

$\Gamma = b!_B p_1'$ then $p_1 \xrightarrow{b!_B p_1''''} p_1'''$ and $b \neq a$ and
either $a \notin fn(p_1'''')$ in which case $p_1' \equiv p_1''''$ and $p_1'' \equiv p_1'''\backslash a$. Then by the par-rule and the res-rule we have $(p_1 | p_2)\backslash a \xrightarrow{b!_B p_1'} (p_1''' | p_2)\backslash a$ which is a matching move.
or $a \in fn(p_1'''')$ in which case $p_1' \equiv p_1''''$ and $p_1'' \equiv p_1'''$ and $B = B' \cup \{a\}$ for some B' with $a \notin B'$. We may assume $B' \cap fn(p_2) = \emptyset$. Then by the par-rule and the open-rule we have $(p_1 | p_2)\backslash a \xrightarrow{b!_B p_1'} p_1''' | p_2$ which is a matching move.

$\Gamma = \tau$ and we may argue as in the above case.

or $\Gamma = \tau$ and the transition has been inferred by the com-close-rule and $p_1 \backslash a \xrightarrow{b?x} p_1' \backslash a$ which has been inferred by the res-rule and $p_1 \xrightarrow{b?x} p_1'$ with $b \neq a$ and $p_2 \xrightarrow{b!_B p_2'} p_2''$ and $p' \equiv ((p_1' \backslash a)[p_2'/x] | p_2'')\backslash B$. We may assume $B \cup fn(p_1) = \emptyset$ and $a \notin fn(p_2')$. Thus by the com-close-rule and the res-rule we may infer that $(p_1 | p_2)\backslash a \xrightarrow{\tau} (p_1'[p_2'/x] | p_2'')\backslash B \backslash a$ which is a matching move since $(p_1'[p_2'/x] | p_2'')\backslash B \backslash a \dot{\sim} (p_1'[p_2'/x] | p_2'')\backslash a \backslash B$ by Proposition 3.8 and $((p_1' \backslash a)[p_2'/x] | p_2'')\, R\, (p_1'[p_2'/x] | p_2'')\backslash a$.

or $\Gamma = \tau$ and the transition has been inferred by the com-close-rule and $p_1 \backslash a \xrightarrow{b!_B p_1'} p_1''$ which has been inferred by the res-rule and $p_1 \xrightarrow{b!_B p_1''''} p_1'''$ and $b \neq a$ and $a \notin fn(p_1'''')$ and $p_1' \equiv p_1''''$ and $p_1'' \equiv p_1'''\backslash a$ and $p_2 \xrightarrow{b?x} p_2''$ and $p' \equiv (p_1''' | p_2''[p_1'/x])\backslash B$. We assume $B \cap fn(p_2'') = \emptyset$. Then by the com-close-rule and the res-rule we have $(p_1 | p_2)\backslash a \xrightarrow{\tau} (p_1''' | p_2''[p_1''''/x])\backslash B \backslash a$ which is a matching move since $(p_1''' | p_2''[p_1''''/x])\backslash B \backslash a \dot{\sim} (p_1''' | p_2''[p_1''''/x])\backslash a \backslash B$ by Proposition 3.8 and $p_1'' | p_2''[p_1'/x]\, R\, (p_1''' | p_2''[p_1''''/x])\backslash a$.

or $\Gamma = \tau$ and the transition has been inferred by the com-close-rule and $p_1 \backslash a \xrightarrow{b!_B p_1'} p_1''$ which has been inferred by the open-rule and $p_1 \xrightarrow{b!_B p_1''''} p_1'''$ and $b \neq a$ and $a \in fn(p_1''') \cap fn(p_1'')$ and $p_1' \equiv p_1''''$ and $p_1'' = p_1''$ and $B = B' \cup \{a\}$ for some B' with $a \notin B'$ and $p_2 \xrightarrow{b?x} p_2''$ and $p' \equiv (p_1'' | p_2''[p_1'/x])\backslash B$. We assume $B' \cap fn(p_2'') = \emptyset$. Then by the com-close-rule and the res-rule we have $(p_1 | p_2)\backslash a \xrightarrow{\tau} (p_1''' | p_2''[p_1''''/x])\backslash B' \backslash a$ which is a matching move since $(p_1''' | p_2''[p_1''''/x])\backslash B' \backslash a \dot{\sim} (p_1''' | p_2''[p_1''''/x])\backslash B$ by Proposition 3.8 and $p_1'' | p_2''[p_1'/x]\, R\, p_1''' | p_2''[p_1''''/x]$.

We omit the proof for the cases showing R^{-1} is an applicative higher order simulation up to $\dot{\sim}$ and restriction. The arguments in these cases are very similar to the above and follow almost from symmetry. □

Proposition A.11 (Proposition 3.15)

$$p|nil \dot{\sim} p$$
$$p_1|p_2 \dot{\sim} p_2|p_1$$
$$p_1|(p_2|p_3) \dot{\sim} (p_1|p_2)|p_3.$$

Proof. This proposition is proved by showing that the first two of the following relations are applicative higher order bisimulations and that the last relation is an applicative higher order bisimulation up to $\dot{\sim}$ and restriction:

$$R_1 = \{(p|nil, p): p \in CPr\} \cup Id$$
$$R_2 = \{(p_1|p_2, p_2|p_1): p_i \in CPr\} \cup Id$$
$$R_3 = \{(p_1|(p_2|p_3), (p_1|p_2)|p_3): p_i \in CPr\} \cup Id.$$

The Id component in each of the above relations is necessary to cover the cases when processes are communicated since these processes might not have the structure of the first part of the relation. To see that the above relations are indeed applicative higher order bisimulations respectively applicative higher order bisimulations up to $\dot{\sim}$ and restriction we analyse each relation in turn. (The Id part of the above relations is obvious.)

R_1 Any transitions of $p|nil$ must have been inferred from a transition of p and the rule for parallel composition since *nil* has no transitions, thus p has a matching move for each move of $p|nil$ and vice versa.

R_2 This is easily established by noting that both rules (par and com-close) involving the parallel operator are symmetric.

R_3 The proof that this relation is an applicative higher order bisimulation up to $\dot{\sim}$ and restriction is surprisingly complicated. This is due to the fact that the communication of processes may introduce restrictions and thus alter the structure of the term. To illustrate this point we show the case when $p_1|(p_2|p_3) \xrightarrow{\tau} p'$ and this transition has been inferred by the com-close-rule and $p_1 \xrightarrow{b?x} p_1'$ and $(p_2|p_3) \xrightarrow{b!_B p''} p'''$ and this is due to an application of the par-rule and $p_2 \xrightarrow{b!_B p''} p_2'$ with $B \cap fn(p_3) = \emptyset$ and $p''' \equiv p_2'|p_3$ and $p' \equiv (p_1'[p''/x]|(p_2'|p_3))\backslash B$. Then by the com-close-rule $p_1|p_2 \xrightarrow{\tau} (p_1'[p''/x]|p_2')\backslash B$ and by the par-rule $(p_1|p_2)|p_3 \xrightarrow{\tau} (p_1'[p''/x]|p_2')\backslash B|p_3$. Since $B \cap fn(p_3) = \emptyset$ we can apply Proposition 3.8 and $(p_1'[p''/x]|p_2')\backslash B|p_3 \dot{\sim} ((p_1'[p''/x]|p_2')|p_3)\backslash B$ and we have established a matching move which satisfies the conditions of an applicative higher order bisimulation up to $\dot{\sim}$ and restriction. There are five other similar cases: one when p_1 does an output transition and p_2 does an input transition, two when p_1 and p_3 communicate and two when p_2 and p_3 communicate. These cases follow the same pattern of argument as above. The only three remaining cases are when either of the three components does a transition on its own but in each case a matching move can be established by two applications of the par-rule. □

Proposition A.12 (Proposition 3.18). *Let* $\bar{x}=\{x_1 \ldots x_n\}$, $\bar{y}=\{y_1 \ldots y_n\}$ *and* $\bar{x} \cap \bar{y} \neq \emptyset$ *and* $A_j \cap fn(q)=\emptyset$ *and* $B_l \cap fn(p)=\emptyset$ *then*

$$\begin{aligned} \textit{if} \quad & p=\Sigma_i\, a_i?x_i.p_i+\Sigma_j\, a_j!_{A_j}\, p'_j.p_j \\ \textit{and} \quad & q=\Sigma_k\, b_k?y_k.q_k+\Sigma_l\, b_l!_{B_l}\, q'_l.q_l \\ \textit{then} \quad & p|q \dot{\sim} \Sigma_i\, a_i?x_i.(p_i|q)+\Sigma_j\, a_j!_{A_j}\, p'_j.(p_j|q) \\ & \quad +\Sigma_k\, b_k?y_k.(p|q_k)+\Sigma_l\, b_l!_{B_l}\, q'_l.(p|q_l) \\ & \quad +\Sigma_{(i,l)\in\{(i,l):a_i=b_l\}}\, \tau.(p_i[q'_l/x_i]|q_l)\backslash B_l \\ & \quad +\Sigma_{(j,k)\in\{(j,k):a_j=b_k\}}\, \tau.(p_j|q_k[p'_j/y_k])\backslash A_j \end{aligned}$$

where $\Sigma_i\, \Gamma_i.p_i$ *describes the sum* $\Gamma_1.p_1+\ldots+\Gamma_n.p_n$ *when* $n>0$ *and nil if* $n=0$, *knowing this notation is unambiguous because of Proposition* 3.7.

Proof. Assume the premises of the proposition. Let rhs denote the right hand side of the above equation. Let

$$R=\{(p|q, rhs)\} \cup Id.$$

Then R is an applicative higher order bisimulation. For each transition of $p|q$ we may find a matching transition of rhs and vice versa.

If $p|q \xrightarrow{\Gamma} r$ then

either $p \xrightarrow{\Gamma} p'$ and $r \equiv p'|q$. If $\Gamma=a_i?x_i$ then $p' \equiv p_i$ for some i and $rhs \xrightarrow{\Gamma} p_i|q$ which is a matching move since $x_i \notin FV(q)$. If $\Gamma=a_j!_{A_j}\, p'_j$ then $p' \equiv p_j$ for some j and $rhs \xrightarrow{\Gamma} p_j|q$ which is a matching move.

or $q \xrightarrow{\Gamma} q'$ and $r \equiv p|q'$. Then similar arguments as above apply.

or $\Gamma=\tau$. Then

either $p \xrightarrow{a_i?x_i} p_i$ and $q \xrightarrow{b_l!_{B_l}q'_l} q_l$ and $r \equiv (p_i[q'_l/x_i]|q_l)\backslash B_l$ and $a_i=b_l$. Then $rhs \xrightarrow{\tau} r$ which is a matching move.

or $q \xrightarrow{b_k?x_k} q_k$ and $p \xrightarrow{a_j!_{A_j}p'_j} p_j$ and $r \equiv (p_j|q_k[p'_j/y_k]|q_k)\backslash A_j$ and $a_j=b_k$. Again $rhs \xrightarrow{\tau} r$ which is a matching move.

If $rhs \xrightarrow{\Gamma} r$ then a similar case analysis as above will yield matching moves for $p|q$. □

Appendix B

We briefly review the π-Calculus as presented in [MilParWal89]. This calculus is a description tool for Mobile Processes with link passing as a means for expressing process networks with dynamically changing interconnection structure.

Processes are built from the following range of constructs: The inactive process 0, three types of prefixes; input prefix $x(y)$, output prefix $\bar{x}y$ and τ prefix, (non-deterministic) choice, parallel composition, restriction, match and recursion.

This is summarised by the syntax of the π-Calculus:

$$P::=0 \mid x(z).P \mid \bar{x}y.P \mid \tau.P \mid P+P' \mid P|P' \mid (y)P \mid [x=y]P \mid \mathtt{rec}\, X.P \mid X.$$

Here $X \in Var$ (a set of variables to be bound by the recursion construct). In [MilParWal89] agent identifiers are used to express recursion, but we prefer the equivalent but more explicit recursion construct above. We shall use the notation $\bar{x}(y).P$ as shorthand for $(y)(\bar{x}y.P)$ for $y \neq x$. This construct creates a new name and sends it out immediately.

In the π-Calculus the communicable values are links or rather names of links, thus x, y above belong to the set *Names* of port names. The constructs of input prefix and restriction bind port names in their scope. The set of free names of a process is denoted by $fn(P)$, the set of bound names of a process is denoted by $bn(P)$ and the set of names of a process is $n(P) = bn(P) \cup fn(P)$.

We may substitute one name for another and name substitution in the π-Calculus follows the pattern of name substitution in Plain CHOCS. We have to take care not to bind free names by input prefix or restriction. If the names coincide we do α-conversion:

$$\{z'/z\}(x(y).P) \equiv \{z'/z\}\, x(y').(\{z'/z\}(\{y'/y\}\, P)) \quad \text{where } y' \notin fn((y)\,P) \cup \{z'\}$$
$$\{z'/z\}((y)\,P) \equiv (y')(\{z'/z\}(\{y'/y\}\, P)) \quad \text{where } y' \notin fn((y)\,P) \cup \{z'\}.$$

Free and bound (recursion) variables are defined as usual and substitution of processes is the usual one taking care of not accidentally binding free names by restriction and free recursion variables by the recursion construct.

The dynamic behaviour of processes is defined in terms of an operational semantics given as a labelled transition system. Processes may evolve by performing actions of the following kind: input actions $x(y)$, free output actions $\bar{a}y$, τ actions and bound output actions $\bar{x}(y)$. Actions are ranged over by α. A name occurring in brackets in an action is said to be a bound name and the set of bound names of an action is denoted by $bn(\alpha)$. $fn(\alpha)$ denotes the set of free names of an action and $n(\alpha)$ denotes the set of all names of an action. $c(\alpha)$ denotes x in $\alpha = x(y)$ and $\alpha = \bar{x}(y)$.

In the following we give the operational semantics for the π-Calculus as presented in [MilParWal89]. Formally the operational semantics is given as the smallest relation $\xrightarrow{\alpha}$ satisfying the following rules:

Table 4. Operational semantics for the π-Calculus. Rules involving the binary operators + and | additionally have symmetric forms

TAU-ACT:	$\tau.P \xrightarrow{\tau} P$
OUTPUT-ACT:	$\bar{x}y.P \xrightarrow{\bar{x}y} P$
INPUT-ACT:	$x(z).P \xrightarrow{x(w)} \{w/z\}\, P,\ w \notin fn((z)\,P)$
SUM:	$\dfrac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'}$
MATCH:	$\dfrac{P \xrightarrow{\alpha} P'}{[x=x]\,P \xrightarrow{\alpha} P'}$
REC:	$\dfrac{P[\mathtt{rec}\ X.P/X] \xrightarrow{\alpha} P'}{\mathtt{rec}\ X.P \xrightarrow{\alpha} P'}$
PAR:	$\dfrac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}, \ bn(\alpha) \cap fn(Q) = \emptyset$

Table 1 (continued)

COM: $$\frac{P \xrightarrow{\bar{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'\{y/z\}}$$

CLOSE: $$\frac{P \xrightarrow{\bar{x}(w)} P' \quad Q \xrightarrow{x(w)} Q'}{P \mid Q \xrightarrow{\tau} (w)(P' \mid Q')}$$

RES: $$\frac{P \xrightarrow{\alpha} P'}{(y)P \xrightarrow{\alpha} (y)P'}, \; y \notin n(\alpha)$$

OPEN: $$\frac{P \xrightarrow{\bar{x}y} P'}{(y)P \xrightarrow{\bar{x}w} \{w/y\}P'}, \; y \neq x, \; w \notin fn((y)P')$$

To compare terms in the π-Calculus we use a generalisation of the notion of bisimulation called strong ground bisimulation:

Definition B.1 *A* strong ground simulation R is a binary relation on CPr such that whenever $(P, Q) \in R$ then:

(i) Whenever $P \xrightarrow{x(y)} P'$ and $y \notin n(P) \cup n(Q)$, then $Q \xrightarrow{x(y)} Q'$ for some Q' and $(\{w/y\}P', \{w/y\}Q') \in R$ for all $w \in$ Names

(ii) Whenever $P \xrightarrow{\bar{x}y} P'$, then $Q \xrightarrow{\bar{x}y} Q'$ for some Q' and $P' R Q'$

(iii) Whenever $P \xrightarrow{\bar{x}(y)} P'$ and $y \notin (n(P) \cup n(Q))$, then $Q \xrightarrow{\bar{x}(y)} Q'$ for some Q' with $P' R Q'$

(iv) Whenever $P \xrightarrow{\tau} P'$, then $Q \xrightarrow{\tau} Q'$ for some Q' with $P' R Q'$

A relation R is a strong ground bisimulation if both it and its inverse are strong ground simulations. Two processes P and Q are said to be strong ground bisimulation equivalent iff there exists a strong ground bisimulation R containing (P, Q). In this case we write $P \sim Q$.

In [MilParWal89] the relation $\sim$ is shown to be an equivalence relation and it has the expected congruence properties with respect to the constructs of the π-Calculus. It also satisfies a set of expected properties.

$$\begin{aligned}
P + 0 &\sim P \\
P + P &\sim P \\
P + Q &\sim Q + P \\
P + (Q + R) &\sim (P + Q) + R \\
(x)P &\sim P \text{ if } x \notin fn(P) \\
(x)(y)P &\sim (y)(x)P \\
(x)(P + Q) &\sim (x)P + (x)Q \\
(x)\alpha.P &\sim \alpha.(x)P \text{ if } x \notin n(\alpha) \\
(x)\alpha.P &\sim 0 \text{ if } x = c(\alpha) \\
P \mid 0 &\sim P \\
P \mid Q &\sim Q \mid P \\
(x)(P \mid Q) &\sim (x)P \mid Q \text{ if } x \notin fn(Q) \\
P \mid (Q \mid R) &\sim (P \mid Q) \mid R.
\end{aligned}$$

The relation $\sim$ is however not preserved by arbitrary name substitutions. A notion of strong bisimulation equivalence $\sim$ is introduced in [MilParWal89] as $P \sim Q$ iff $\{a/b\}P \sim \{a/b\}Q$ for all name substitutions $\{a/b\}$. We shall not concern ourselves with this relation since the strong ground bisimulation relation suffices for the presentation in this paper.

Before turning to translations between the π-Calculus and Plain CHOCS we present a useful construct and show a few facts about this. We shall need communications which carry no parameters. This could be modelled by presupposing a special name ε which is never bound and we write $\bar{x}.P$ in place of $\bar{x}\varepsilon.P$ and $x.P$ in Place of $x(y).P$ where y is not free in P.

Definition B.2 *Let*

$$b \Rightarrow P = \mathrm{rec}\, X.b.(P \mid X)$$

where $b \notin n(P)$ and $X \notin FV(P)$.

This construction is intended to provide copies of P when triggered by $\bar{b}$ actions e.g.:

$$(b)(\bar{b}.nil \mid \bar{b}.nil \mid b \Rightarrow P) \xrightarrow{\tau} \xrightarrow{\tau} (b)(nil \mid nil \mid P \mid P \mid b \Rightarrow P) \sim P \mid P.$$

This construct satisfies several interesting properties:

Lemma B.3 (Lemma 4.2) *If* $b \notin n(Q)$ *then*

$$(b)(P_1 \mid b \Rightarrow Q) + (b)(P_2 \mid b \Rightarrow Q) \sim (b)((P_1 + P_2) \mid b \Rightarrow Q)$$

Proof. Let LHS denote the left hand side and RHS denote the right hand side of the above equation. First note that the P_i's are allowed to trigger $b \Rightarrow Q$, but we assume that only $b \Rightarrow Q \xrightarrow{b} Q \mid b \Rightarrow Q$. We use b as a private name in both summands of LHS and in RHS, this is convenient and obtainable by a suitable α-conversion on the private names.

To prove the lemma we show that the relation:

$$R = \{(LHS, RHS)\} \cup Id$$

is a strong ground bisimulation.

To see this observe that if $LHS \xrightarrow{\alpha} R$ then

either $(b)(P_1 \mid b \Rightarrow Q) \xrightarrow{\alpha} R$ and this is because

either $P_1 \xrightarrow{\alpha} P_1'$ with $\alpha \neq \bar{b}$ and $R \equiv (b)(P_1' \mid b \Rightarrow Q)$. Then $RHS \xrightarrow{\alpha} (b)(P_1' \mid b \Rightarrow Q)$ which is a matching move.

or $P_1 \xrightarrow{\bar{b}} P_1'$ with $\alpha = \tau$ and $R \equiv (b)(P_1' \mid Q \mid b \Rightarrow Q)$. Then $RHS \xrightarrow{\tau} (b)(P_1' \mid Q \mid b \Rightarrow Q)$ which is a matching move.

or $(b)(P_2 \mid b \Rightarrow Q) \xrightarrow{\alpha} R$ and an argument as above applies.

Also if $RHS \xrightarrow{\alpha} R$ then

either $P_1 \xrightarrow{\alpha'} P_1'$ with

either $\alpha' = \alpha \neq \bar{b}$ and $R \equiv (b)(P_1' \mid b \Rightarrow Q)$. Then $LHS \xrightarrow{\alpha} (b)(P_1' \mid b \Rightarrow Q)$ which is a matching move.

or $\alpha' = \bar{b}$ and $\alpha = \tau$ and $R \equiv (b)(P_1' \mid Q \mid b \Rightarrow Q)$. Then $LHS \xrightarrow{\tau} (b)(P_1' \mid Q \mid b \Rightarrow Q)$ which is a matching move.

or $P_2 \xrightarrow{\alpha'} P_2'$ and an argument as above applies.

We have abused the notation slightly when $\alpha \neq \bar{b}$ in the above proof since we should analyse each case of α: $a(x)$, $\bar{a}b$, $\bar{a}(c)$ or τ. We shall not do so since it is not hard (only elaborate) and each case follows the general pattern. □

Lemma B.4 (Lemma 4.3) *If* $P_i' \not\xrightarrow{b}$ *for all derivatives* P_i' *of* P_i, $i \in \{1, 2\}$ *and* $b \notin fn(Q)$ *then*

$$(b)(P_1 \mid b \Rightarrow Q) \mid (b)(P_2 \mid b \Rightarrow Q) \sim (b)((P_1 \mid P_2) \mid b \Rightarrow Q).$$

Proof. Let *LHS* denote the left hand side and *RHS* denote the right hand side of the above equation. To prove the lemma we show that the relation:

$$R = \{(LHS, RHS)\}$$

is a strong ground bisimulation.

To see this observe that if $LHS \xrightarrow{\alpha} R$ then

either $(b)(P_1 \mid b \Rightarrow Q) \xrightarrow{\alpha} R'$ and $R \equiv R' \mid (b)(P_2 \mid b \Rightarrow Q)$ and this is because

either $P_1 \xrightarrow{\alpha} P_1'$ with $\alpha \neq \bar{b}$ and $R' \equiv (b)(P_1' \mid b \Rightarrow Q)$. Then $RHS \xrightarrow{\alpha} (b)(P_1' \mid P_2 \mid b \Rightarrow Q)$ which is a matching move.

or $P_1 \xrightarrow{\bar{b}} P_1'$ with $\alpha = \tau$ and $R' \equiv (b)(P_1' \mid Q \mid b \Rightarrow Q)$. Then $RHS \xrightarrow{\tau} (b)(P_1' \mid P_2 \mid Q \mid b \Rightarrow Q)$ which is a matching move.

or $(b)(P_2 \mid b \Rightarrow Q) \xrightarrow{\alpha} R'$ and an argument as above applies.

or $P_1 \xrightarrow{\alpha} P_1'$ and $P_2 \xrightarrow{\bar{\alpha}} P_2'$ and $\alpha = \tau$ and $R \equiv (b)(P_1' \mid b \Rightarrow Q) \mid (b)(P_2' \mid b \Rightarrow Q)$, where $\bar{\alpha}$ is an action with opposite polarity of α [MilParWal89]. Then $RHS \xrightarrow{\tau} (b)(P_1' \mid P_2' \mid b \Rightarrow Q)$ which is a matching move.

Also if $RHS \xrightarrow{\alpha} R$ we may argue in a similar way as above. □

Lemma B.5 (Lemma 4.4) *If* $P_i' \not\xrightarrow{b}$ *for all derivatives* P_i' *of* P_i, $i \in \{1, 2\}$ *and* $b \notin fn(Q)$ and $c \notin fn(P_1) \cup fn(P_2) \cup fn(Q)$ *then*

$$(c \Rightarrow (b)(P_1 \mid b \Rightarrow Q)) \mid (b)(P_2 \mid b \Rightarrow Q) \sim (b)(c \Rightarrow P_1 \mid P_2 \mid b \Rightarrow Q)$$

Proof. Let *LHS* denote the left hand side and *RHS* denote the right hand side of the above equation. To prove the lemma we show that the relation:

$$R = \{(LHS, RHS)\}$$

is a strong ground bisimulation up to $\sim$ (strong ground bisimulation up to $\sim$ is defined similarly to the definition of bisimulation up to $\sim$ in [Mil89]).

To see this observe that if $LHS \xrightarrow{\alpha} R$ then

either $\alpha = c$ and $R \equiv (b)(P_1 \mid b \Rightarrow Q) \mid (c \Rightarrow (b)(P_1 \mid b \Rightarrow Q)) \mid (b)(P_2 \mid b \Rightarrow Q)$
$\sim (c \Rightarrow (b)(P_1 \mid b \Rightarrow Q)) \mid (b)(P_1 \mid P_2 \mid b \Rightarrow Q)$ which follows by Lemma 4.3. Then
$RHS \xrightarrow{c} (b)(P_1 \mid c \Rightarrow P_1 \mid P_2 \mid b \Rightarrow Q)$ which is a matching move.

or $\alpha \neq c$ and $(b)(P_2 \mid b \Rightarrow Q) \xrightarrow{\alpha} R'$ and $R \equiv c \Rightarrow (b)(P_1, b \Rightarrow Q) \mid R'$ and this is because

either $P_2 \xrightarrow{\alpha'} P_2'$ with $\alpha = \alpha' \neq \bar{b}$ and $R' \equiv (b)(P_2' \mid b \Rightarrow Q)$. Then
$RHS \xrightarrow{\alpha} (b)(c \Rightarrow P_1 \mid P_2' \mid b \Rightarrow Q)$ which is a matching move.

or $P_2 \xrightarrow{\bar{b}} P_2'$ with $\alpha = \tau$ and $R' \equiv (b)(P_2 \mid Q \mid b \Rightarrow Q)$. Then
$RHS \xrightarrow{\tau} (b)(c \Rightarrow P_1 \mid P_2 \mid Q \mid b \Rightarrow Q)$ which is a matching move.

Also if $RHS \xrightarrow{\alpha} R$ we may argue in a similar way as above. □

Proposition B.6 (Proposition 4.6) $[\![p[q/x]_\tau]\!] \sim (b)([\![p]\!]\{b/x\} \mid b \Rightarrow [\![q]\!])$ *where* $b \notin fn(p) \cup fn(q)$

Proof. By structural induction on p using Lemma 4.2 to Lemma 4.4.

$p \equiv nil$	$[\![nil[q/x]_\tau]\!]$	$\equiv$	by definition of $[\ /\ ]_\tau$
	$[\![nil]\!]$	$=$	by definition of $[\![\ ]\!]$
	0	$\sim$	by algebraic laws
	$(b)(0\{b/x\} \mid (b \Rightarrow [\![q]\!]))$	$\sim$	by definition of $[\![\ ]\!]$
	$(b)([\![nil]\!]\{b/x\} \mid (b \Rightarrow [\![q]\!]))$		
$p \equiv a?y.p_1$	Assume $y \neq x$ and $y \notin FV(q)$		(otherwise use α-conversion)
	$[\![(a?y.p_1)[q/x]_\tau]\!]$	$\equiv$	by definition of $[\ /\ ]_\tau$
	$[\![a?y.(p_1[q/x]_\tau)]\!]$	$=$	by definition of $[\![\ ]\!]$
	$a(y).[\![(p_1[q/x]_\tau)]\!]$	$\sim$	by I.H.
	$a(y).(b)([\![p_1]\!]\{b/x\} \mid (b \Rightarrow [\![q]\!]))$	$\sim$	since $b \notin fn(p)$
	$(b)(a(y).([\![p_1]\!]\{b/x\} \mid (b \Rightarrow [\![q]\!])))$	$\sim$	since $y \notin FV(q)$ and $y \neq x$
	$(b)((a(y).[\![p_1]\!])\{b/x\} \mid (b \Rightarrow [\![q]\!]))$	$=$	by definition of $[\![\ ]\!]$
	$(b)(([\![a?y.p_1]\!])\{b/x\} \mid (b \Rightarrow [\![q]\!]))$		
$p \equiv a!p_1.p_2$	$[\![(a!p_1.p_2)[q/x]_\tau]\!]$		
	$\equiv$ by definition of $[\ /\ ]_\tau$		
	$[\![a!(p_1[q/x]_\tau).(p_2[q/x]_\tau)]\!]$		
	by definition of $[\![\ ]\!]$		
	$(b)(\bar{a}b.((b \Rightarrow [\![p_1[q/x]_\tau]\!]) \mid [\![p_2[q/x]_\tau]\!]))$		
	$\sim$ by I.H. $(b \notin fn(p_1) \cup fn(p_2) \cup fn(q))$		
	$(b)(\bar{a}b.((c)((b \Rightarrow (c)([\![p_1]\!]\{c/x\} \mid (c \Rightarrow [\![q]\!])))$		
	$\mid (c)([\![p_2]\!]\{c/x\} \mid (c \Rightarrow [\![q]\!])))))$		
	$\sim$ by Lemma 4.4.		
	$(b)(\bar{a}b.(c)((b \Rightarrow [\![p_1]\!]\{c/x\}) \mid [\![p_2]\!]\{c/x\} \mid (c \Rightarrow [\![q]\!])))$		
	$\sim$ since $c \neq a$ and $c \neq b$, otherwise use α-conversion on c		
	$(c)(b)(\bar{a}b.((b \Rightarrow [\![p_1]\!]\{c/x\}) \mid [\![p_2]\!]\{c/x\} \mid (c \Rightarrow [\![q]\!])))$		
	$\equiv$ by definition of $\{\ /\ \}$		
	$(c)(b)(\bar{a}b.(((b \Rightarrow [\![p_1]\!]) \mid [\![p_2]\!])\{c/x\} \mid (c \Rightarrow [\![q]\!])))$		
	$\sim$		
	$(c)((b)(\bar{a}b.(((b \Rightarrow [\![p_1]\!]) \mid [\![p_2]\!])\{c/x\}) \mid (c \Rightarrow [\![q]\!])))$		
	$\sim$ since $b \notin fn(q)$		
	$(c)([\![a!p_1.p_2]\!]\{c/x\} \mid (c \Rightarrow [\![q]\!]))$		

$p \equiv p_1 + p_2$

$[\![(p_1+p_2)[q/x]_\tau]\!]$
$\equiv$ by definition of $[\ /\ ]_\tau$ and by definition of $[\![\ ]\!]$
$[\![p_1[q/x]_\tau]\!] + [\![p_2[q/x]_\tau]\!]$
$\sim$ by I.H.
$(b)([\![p_1]\!]\{b/x\}|(b \Rightarrow [\![q]\!])) + (b)([\![p_2]\!]\{b/x\}|(b \Rightarrow [\![q]\!]))$
$\sim$ by Lemma 4.2.
$(b)(([\![p_1]\!]\{b/x\} + [\![p_2]\!]\{b/x\})|(b \Rightarrow [\![q]\!]))$
$\equiv$ by definition of $\{\ /\ \}$
$(b)([\![p_1 + p_2]\!]\{b/x\}|(b \Rightarrow [\![q]\!]))$

$p \equiv p_1 | p_2$

$[\![(p_1|p_2)[q/x]_\tau]\!]$
$\equiv$ by definition of $[\ /\ ]_\tau$ and by definition of $[\![\ ]\!]$
$[\![p_1[q/x]_\tau]\!] | [\![p_2[q/x]_\tau]\!]$
$\sim$ by I.H.
$(b)([\![p_1]\!]\{b/x\}|(b \Rightarrow [\![q]\!]))|(b)([\![p_2]\!]\{b/x\}|(b \Rightarrow [\![q]\!]))$
$\sim$ by Lemma 4.3.
$(b)(([\![p_1]\!]\{b/x\} | [\![p_2]\!]\{b/x\})|(b \Rightarrow [\![q]\!]))$
$\equiv$ by definition of $\{\ /\ \}$
$(b)([\![p_1|p_2]\!]\{b/x\}|(b \Rightarrow [\![q]\!]))$

$p \equiv p_1 \backslash a$

Assume $a \notin fn(q)$ otherwise use α-conversion.

$[\![(p_1 \backslash a)[q/x]_\tau]\!]$	$\equiv$	by definition of $[\ /\ ]_\tau$
$[\![(p_1[q/x]_\tau)\backslash a]\!]$	$=$	by definition of $[\![\ ]\!]$
$(a)([\![p_1[q/x]_\tau]\!])$	$\sim$	by I.H.
$(a)(b)([\![p_1]\!]\{b/x\}\|(b \Rightarrow [\![q]\!]))$	$\sim$	since $a \notin fn(q)$
$(b)((a)([\![p_1]\!]\{b/x\})\|(b \Rightarrow [\![q]\!]))$	$\equiv$	by definition of $\{\ /\ \}$
$(b)([\![p_1 \backslash a]\!]\{b/x\}\|(b \Rightarrow [\![q]\!]))$		

$p \equiv y$

if $y \neq x$ then

$[\![y[q/x]_\tau]\!]$	$\equiv$	
$[\![y]\!]$	$=$	
$\bar{y}.0$	$\sim$	
$(b)((\bar{y}.0)\{b/x\}\|(b \Rightarrow [\![q]\!]))$		

if $y = x$ then

$[\![y[q/x]_\tau]\!]$	$\equiv$	
$[\![\tau.q]\!]$	$=$	
$\tau.[\![q]\!]$	$\sim$	
$(b)((\bar{y}.0)\{b/x\}\|(b \Rightarrow [\![q]\!]))$	$\square$	

Proposition B.7 (Proposition 4.7)

1. *if* $p \xrightarrow{a?x} p'$ *then* $[\![p]\!] \xrightarrow{a(x)} [\![p']\!]$
2. *if* $p \xrightarrow{a!_B p'} p''$ *then* $[\![p]\!] \xrightarrow{\bar{a}(b)} Q \sim (b_1)\ldots(b_n)(b \Rightarrow [\![p']\!] | [\![p'']\!])$ *where* $B = \{b_1, \ldots, b_n\}$ *for some* Q.
3. *if* $p \xrightarrow{\tau} p'$ *then* $[\![p]\!] \xrightarrow{\tau} [\![p']\!]$
4. *if* $Q \sim [\![p]\!]$ *and* $Q \xrightarrow{a(x)} Q'$ *then* $p \xrightarrow{a?x} p'$ *for some* p' *with* $Q'\{b/x\} \sim [\![p']\!]\{b/x\}$ *for all* $b \in Names$.
5. *if* $Q \sim [\![p]\!]$ *then* $Q \xrightarrow{\bar{a}b} \not\rightarrow$.
6. *if* $Q \sim [\![p]\!]$ *and* $Q \xrightarrow{\bar{a}(b)} Q'$ *then* $p \xrightarrow{a!_B p'} p''$ *with* $Q' \sim (b_1)\ldots(b_n)(b \Rightarrow [\![p']\!] | [\![p'']\!])$ *for some* B, p', p'' *where* $B = \{b_1, \ldots, b_n\}$.
7. *if* $Q \sim [\![p]\!]$ *and* $Q \xrightarrow{\tau} Q'$ *then* $p \xrightarrow{\tau} p'$ *with* $Q' \sim [\![p']\!]$ *for some* p'.

Proof. 1. By induction on the length of the inference used to establish $p \xrightarrow{a?x} p'$ observing the structure of the process p. The cases when $p \equiv nil$, $p \equiv a!p_1.p_2$ and $p \equiv \tau.p_1$ are trivial since $p \overset{a?x}{\not\longrightarrow}$.

$p \equiv a?x.p_1$ Then $a?x.p_1 \xrightarrow{a?x} p_1$ by the input-rule and $p' \equiv p_1$. Also $[\![a?x.p_1]\!] = a(x).[\![p_1]\!] \xrightarrow{a(x)} [\![p_1]\!]$ by the INPUT-ACT-rule.

$p \equiv p_1 + p_2$ If $p \xrightarrow{a?x} p'$ then

either $p_1 \xrightarrow{a?x} p'$ by a shorter inference, and by induction we have $[\![p_1]\!] \xrightarrow{a(x)} [\![p']\!]$ and by the SUM-rule we have $[\![p_1 + p_2]\!] \xrightarrow{a(x)} [\![p']\!]$.

or $p_2 \xrightarrow{a?x} p'$ by a shorter inference, and by induction we have $[\![p_2]\!] \xrightarrow{a(x)} [\![p']\!]$ and by the SUM-rule we have $[\![p_1 + p_2]\!] \xrightarrow{a(x)} [\![p']\!]$.

$p \equiv p_1 | p_2$ If $p \xrightarrow{a?x} p'$ then

either $p_1 \xrightarrow{a?x} p_1'$ and $p' \equiv p_1' | p_2$ by a shorter inference, and by induction we have $[\![p_1]\!] \xrightarrow{a(x)} [\![p_1']\!]$ and by the PAR-rule we have $[\![p_1 | p_2]\!] \xrightarrow{a(x)} [\![p']\!]$.

or $p_2 \xrightarrow{a?x} p_2'$ and $p' \equiv p_1 | p_2'$ by a shorter inference, and by induction we have $[\![p_2]\!] \xrightarrow{a(x)} [\![p_2']\!]$ and by the PAR-rule we have $[\![p_1 | p_2]\!] \xrightarrow{a(x)} [\![p']\!]$.

$p \equiv p_1 \backslash b$ If $p \xrightarrow{a?x} p'$ then $p_1 \xrightarrow{a?x} p_1'$ with $a \neq b$ and $p' \equiv p_1' \backslash b$ by a shorter inference, and by induction we have $[\![p_1]\!] \xrightarrow{a(x)} [\![p_1']\!]$ and by the RES-rule we have $[\![p_1 \backslash b]\!] \xrightarrow{a(x)} [\![p']\!]$.

2. By induction on the length of the inference used to establish $p \xrightarrow{a!_B p'} p''$ observing the structure of the process p. The cases when $p \equiv nil$, $p \equiv a?x.p_1$ and $p \equiv \tau.p_1$ are trivial since $p \overset{a!_B p'}{\not\longrightarrow}$.

$p \equiv a!p_1.p_2$ Then $a!p_1.p_2 \xrightarrow{a!_\emptyset p_1} p_2$ by the output-rule. Also $[\![a!p_1.p_2]\!] \xrightarrow{\bar{a}(b)} (b \Rightarrow [\![p_1]\!]) | [\![p_2]\!]$ by the INPUT-ACT-rule.

$p \equiv p_1 + p_2$ If $p \xrightarrow{a!_B p'} p''$ then

either $p_1 \xrightarrow{a!_B p'} p''$ by a shorter inference, and by induction we have $[\![p_1]\!] \xrightarrow{\bar{a}(b)} P''' \sim (b_1)\ldots(b_n)((b \Rightarrow [\![p']\!]) | [\![p'']\!])$ and by the SUM-rule we have $[\![p_1 + p_2]\!] \xrightarrow{\bar{a}(b)} P''' \sim (b_1)\ldots(b_n)((b \Rightarrow [\![p']\!]) | [\![p'']\!])$

or $p_2 \xrightarrow{a!_B p'} p''$ by a shorter inference, and by induction we have $[\![p_2]\!] \xrightarrow{\bar{a}(b)} P''' \sim (b_1)\ldots(b_n)((b \Rightarrow [\![p']\!]) | [\![p'']\!])$ and by the SUM-rule we have $[\![p_1 + p_2]\!] \xrightarrow{\bar{a}(b)} P''' \sim (b_1)\ldots(b_n)((b \Rightarrow [\![p']\!]) | [\![p'']\!])$

$p \equiv p_1 | p_2$ If $p \xrightarrow{a!_B p'} p''$ then

either $p_1 \xrightarrow{a!_B p'} p_1''$ and $p'' \equiv p_1'' | p_2$ by a shorter inference, and by induction we have $[\![p_1]\!] \xrightarrow{\bar{a}(b)} P_1''' \sim (b_1)\ldots(b_n)((b \Rightarrow [\![p']\!]) | [\![p_1'']\!])$ and by the PAR-rule we have $[\![p_1 | p_2]\!] \xrightarrow{\bar{a}(b)} P''' \sim (b_1)\ldots(b_n)((b \Rightarrow [\![p']\!]) | [\![p_1'']\!]) | [\![p_2]\!] \sim (b_1)\ldots(b_n)((b \Rightarrow [\![p']\!]) | [\![p'']\!])$ (using a suitable α-conversion).

or $p_2 \xrightarrow{a!_B p'} p_2''$ and $p'' \equiv p_1 | p_2''$ by a shorter inference, and by induction we have $[\![p_2]\!] \xrightarrow{\bar{a}(b)} P_2''' \sim (b_1)\ldots(b_n)((b \Rightarrow [\![p']\!]) | [\![p_2'']\!])$ and by the PAR-rule we have $[\![p_1 | p_2]\!] \xrightarrow{\bar{a}(b)} P''' \sim [\![p_1]\!] | (b_1)\ldots(b_n)((b \Rightarrow [\![p']\!]) | [\![p_2'']\!]) \sim (b_1)\ldots(b_n)((b \Rightarrow [\![p']\!]) | [\![p'']\!])$ (using a suitable α-conversion).

$p \equiv p_1 \backslash d$ If $p \xrightarrow{a!_B p'} p''$ then

either $p_1 \xrightarrow{a!_{B'} p'} p_1''$ by a shorter inference and $d \in fn(p')$ and $B = B' \cup \{d\}$ and $a \neq d$ and $p'' \equiv p_1'' \backslash d$. By induction we have $[\![p_1]\!] \xrightarrow{\bar{a}(b)} p_1''' \sim (b_1)\ldots(b_k)((b \Rightarrow [\![p']\!]) | [\![p_1'']\!])$ and by the RES-rule we have $[\![p_1 \backslash d]\!] = (d)([\![p_1]\!]) \xrightarrow{\bar{a}(b)} P''' \sim (d)(b_1)\ldots(b_n)((b \Rightarrow [\![p']\!]) | [\![p_1'']\!])$.

or $p_1 \xrightarrow{a!_B p_1'} p_1''$ by shorter inference and $d \notin fn(p')$ and $a \neq d$ and $p' \equiv p_1'$ and $p'' \equiv p_1'' \backslash d$. By induction we have $[\![p_1]\!] \xrightarrow{\bar{a}(b)} p_1''' \sim (b_1)\ldots(b_k)((b \Rightarrow [\![p']\!]) | [\![p_1'']\!]$ and by the RES-rule we have $[\![p_1 \backslash d]\!] = (d)([\![p_1]\!]) \xrightarrow{\bar{a}(b)} P''' \sim (d)(b_1)\ldots(b_n)((b \Rightarrow [\![p']\!]) | [\![p_1'']\!]) \sim (b_1)\ldots(b_n)((b \Rightarrow [\![p_1']\!]) | (d)[\![p_1'']\!])$.

3. By induction on the length of the inference used to establish $p \xrightarrow{\tau} p'$ observing the structure of the process p. The cases when $p \equiv nil$, $p \equiv a?x.p_1$ and $p \equiv a!p_1.p_2$ are trivial since $p \not\xrightarrow{\tau}$.

$p \equiv \tau.p_1$ Then $p \xrightarrow{\tau} p_1$ by the tau-rule and $p' \equiv p_1$. Also $[\![p]\!] = \tau.[\![p_1]\!] \xrightarrow{\tau} [\![p_1]\!]$ by the TAU-ACT-rule.

$p \equiv p_1 + p_2$ If $p \xrightarrow{\tau} p'$ then

either $p_1 \xrightarrow{\tau} p'$ by a shorter inference, and by induction we have $[\![p_1]\!] \xrightarrow{\tau} [\![p']\!]$ and by the SUM-rule we have $[\![p_1 + p_2]\!] \xrightarrow{\tau} [\![p']\!]$.

or $p_2 \xrightarrow{\tau} p'$ by a shorter inference, and by induction we have $[\![p_2]\!] \xrightarrow{\tau} [\![p']\!]$ and by the SUM-rule we have $[\![p_1 + p_2]\!] \xrightarrow{\tau} [\![p']\!]$.

$p \equiv p_1 | p_2$ If $p \xrightarrow{\tau} p'$ then

either $p_1 \xrightarrow{\tau} p_1'$ and $p' \equiv p_1' | p_2$ by a shorter inference, and by induction we have $[\![p_1]\!] \xrightarrow{\tau} [\![p_1']\!]$ and by the PAR-rule we have $[\![p_1 | p_2]\!] \xrightarrow{\tau} [\![p']\!]$.

or $p_2 \xrightarrow{\tau} p_2'$ and $p' \equiv p_1 | p_2'$ by a shorter inference, and by induction we have $[\![p_2]\!] \xrightarrow{\tau} [\![p_2']\!]$ and by the PAR-rule we have $[\![p_1 | p_2]\!] \xrightarrow{\tau} [\![p']\!]$.

or $p_1 \xrightarrow{a?x} p_1'$ and $p_2 \xrightarrow{a!_B p_2'} p_2''$ by shorter inferences and $p' \equiv (p_1'[p_2'/x]_\tau | p_2'') \backslash B$. By induction and Propositions 4.7.1 and 4.7.2 we have $[\![p_1]\!] \xrightarrow{a(x)} [\![p_1']\!]$ and $[\![p_2]\!] \xrightarrow{\bar{a}(b)} (b_1)\ldots(b_n)((b \Rightarrow [\![p_2']\!]) | [\![p_2'']\!])$. Then by the CLOSE-rule we have $[\![p_1 | p_2]\!] \xrightarrow{\tau} (b)([\![p']\!]\{b/x\} | (b_1)\ldots(b_n)((b \Rightarrow [\![p_2']\!]) | [\![p_2'']\!])) \sim (b_1)\ldots(b_n)((b)([\![p']\!]\{b/x\} | (b \Rightarrow [\![p_2']\!]) | [\![p_2'']\!])) = [\![(p_1'[p_2'/x]_\tau | p_2'') \backslash B]\!]$ by Proposition 4.6 and assuming $B \cap fn(p') = \emptyset$ (otherwise use α-conversion).

or $p_2 \xrightarrow{a?x} p_2'$ and $p_1 \xrightarrow{a!_B p_1'} p_1''$ and we may argue as above.

$p \equiv p_1 \backslash b$ If $p \xrightarrow{\tau} p'$ then $p_1 \xrightarrow{\tau} p_1'$ by a shorter inference, and by induction we have $[\![p_1]\!] \xrightarrow{\tau} [\![p_1']\!]$ and by the RES-rule we have $[\![p_1 \backslash b]\!] = (b)([\![p_1]\!]) \xrightarrow{\tau} (b)([\![p_1]\!]) = [\![p']\!]$.

4. Assume $Q \sim [\![p]\!]$ and $Q \xrightarrow{a(x)} Q'$. Then $[\![p]\!] \xrightarrow{a(x)} Q''$ for some Q'' with $Q'\{b/x\} \sim Q''\{b/x\}$ for all $b \in Names$ since $Q \sim [\![p]\!]$.

We proceed by induction on the length of the inference used to establish $[\![p]\!] \xrightarrow{a(x)} Q''$ observing the structure of p.

If $[\![p]\!] \xrightarrow{a(x)} Q''$ then p must have one of the following forms:

$p \equiv a?x.p_1$ In this case $[\![p]\!] \xrightarrow{a(x)} [\![p_1]\!]$. By the input-rule we have $a?x.p_1 \xrightarrow{a?x} p_1$ which proves the lemma in this case.

$p \equiv p_1 + p_2$ In this case

either $[\![p_1]\!] \xrightarrow{a(x)} Q''$ by a shorter inference and by induction $p_1 \xrightarrow{a?x} p_1'$ and $Q''\{b/x\} \sim [\![p_1']\!]\{b/x\}$ for all $b \in Names$. By the sum-rule we have $p_1 + p_2 \xrightarrow{a?x} p_1'$ and $Q''\{b/x\} \sim [\![p_1']\!]\{b/x\}$ for all $b \in Names$.

or $[\![p_2]\!] \xrightarrow{a(x)} Q''$ by a shorter inference and by induction $p_2 \xrightarrow{a?x} p_2'$ and $Q''\{b/x\} \sim [\![p_2']\!]\{b/x\}$ for all $b \in Names$. By the sum-rule we have $p_1 + p_2 \xrightarrow{a?x} p_2'$ and $Q''\{b/x\} \sim [\![p_2']\!]\{b/x\}$ for all $b \in Names$.

$p \equiv p_1 | p_2$ In this case

either $[\![p_1]\!] \xrightarrow{a(x)} Q_1''$ by a shorter inference and $Q'' \equiv Q_1'' | [\![p_2]\!]$. By induction $p_1 \xrightarrow{a?x} p_1'$ and $Q_1'\{b/x\} \sim [\![p_1']\!]\{b/x\}$ for all $b \in Names$. By the par-rule we have $p_1 | p_2 \xrightarrow{a?x} p_1' | p_2$ and $Q''\{b/x\} \sim [\![p_1' | p_2]\!]\{b/x\}$ for all $b \in Names$.

or $[\![p_2]\!] \xrightarrow{a(x)} Q_2''$ by a shorter inference and $Q'' \equiv [\![p_1]\!] | Q_2''$. By induction $p_2 \xrightarrow{a?x} p_2'$ and $Q_2' \sim [\![p_2']\!]$. By the par-rule we have $p_1 | p_2 \xrightarrow{a?x} p_1 | p_2'$ and $Q''\{b/x\} \sim [\![p_1 | p_2']\!]\{b/x\}$ for all $b \in Names$.

$p \equiv p_1 \backslash c$ In this case $[\![p_1]\!] \xrightarrow{a(x)} Q_1''$ and $a \neq c$. By induction $p_1 \xrightarrow{a?x} p_1'$ and $[\![p_1]\!]\{b/x\} \sim Q_1''\{b/x\}$ for all $b \in Names$. By the res-rule we have $p_1 \backslash c \xrightarrow{a?x} p_1' \backslash c$ and $Q''\{b/x\} \sim [\![p_1' \backslash c]\!]\{b/x\}$ for all $b \in Names$.

5. From the definition of $[\![\ ]\!]$ it is easy to see that $[\![p]\!] \xrightarrow{\bar{a}b}$. Since $Q \sim [\![p]\!]$ this must be true for Q.

6. Assume $Q \sim [\![p]\!]$ and $Q \xrightarrow{\bar{a}(b)} Q'$. Then $[\![p]\!] \xrightarrow{\bar{a}(b)} Q''$ with $Q' \sim Q''$, since $Q \sim [\![p]\!]$. We proceed by induction on the length of the inference used to establish $[\![p]\!] \xrightarrow{\bar{a}(b)} Q''$ observing the structure of p.

If $[\![p]\!] \xrightarrow{\bar{a}(b)} Q''$ then p must have one of the following forms:

$p \equiv a!p_1.p_2$ From the output-rule we have $p \xrightarrow{a!_\emptyset p_1} p_2$ and from the OUTPUT-ACT-rule we have $[\![p]\!] \xrightarrow{\bar{a}(b)} (b \Rightarrow [\![p_1]\!]) | [\![p_2]\!]$ which proves the lemma in this case.

$p \equiv p_1 + p_2$ **either** $[\![p_1]\!] \xrightarrow{\bar{a}(b)} Q''$ by a shorter inference and by induction we have $p_1 \xrightarrow{a!_B p_1'} p_1''$ and $Q'' \sim (b_1) \ldots (b_n)((b \Rightarrow [\![p_1']\!]) | [\![p_1'']\!])$. Then $p_1 + p_2 \xrightarrow{a!_B p_1'} p_1''$ by the sum-rule and by the SUM-rule we have $[\![p]\!] \xrightarrow{\bar{a}(b)} Q''$ which proves the lemma in this case.

or $[\![p_2]\!] \xrightarrow{\bar{a}(b)} Q''$ and an argument as above applies.

$p \equiv p_1 | p_2$ **either** $[\![p_1]\!] \xrightarrow{\bar{a}(b)} Q_1''$ by a shorter inference and $Q'' \sim Q_1'' | [\![p_2]\!]$. By induction we have $p_1 \xrightarrow{a!_B p_1'} p_1''$ with $Q_1'' \sim (b_1) \ldots (b_n)((b \Rightarrow [\![p_1']\!]) | [\![p_1'']\!])$. By the par-rule we have $p_1 | p_2 \xrightarrow{a!_B p_1'} p_1'' | p_2$ and by the PAR-rule and RES-rule we have $[\![p_1 | p_2]\!] \xrightarrow{\bar{a}(b)} Q'' \sim (b_1) \ldots (b_n)((b \Rightarrow [\![p_1']\!]) | [\![p_1'' | p_2]\!])$ by a suitable α-conversion such that $B \cap fn(p_2) = \emptyset$.

or $[\![p_2]\!] \xrightarrow{\bar{a}(b)} Q_2''$ and symmetric arguments as above yield the result.

$p \equiv p_1 \backslash d$ Then $[\![p_1]\!] \xrightarrow{\bar{a}(b)} Q_1''$ by a shorter inference and $a \neq d$ and $Q'' \equiv (d)(Q_1'')$. By induction we have $p_1 \xrightarrow{a!_B p_1'} p_1''$ with $Q_1'' \sim (b_1)\ldots(b_n)((b \Rightarrow [\![p_1']\!]) | [\![p_1'']\!])$. If $d \notin fn(p_1')$ then by the res-rule we have $p_1 \backslash d \xrightarrow{a!_B p_1'} p_1'' \backslash d$ and by the RES-rule we have $[\![p]\!] \xrightarrow{\bar{a}(b)} Q'' \sim (b_1)\ldots(b_n)(d)((b \Rightarrow [\![p_1']\!]) | [\![p_1'']\!] \sim (b_1)\ldots(b_n)((b \Rightarrow [\![p_1']\!])|(d)[\![p_1'']\!])$.
If $d \in fn(p_1')$ then by the open-rule we have $p_1 \backslash d \xrightarrow{a!_{B \cup \{d\}} p_1'} p_1''$ and by the RES-rule we have $[\![p]\!] \xrightarrow{\bar{a}(b)} Q'' \sim (b_1)\ldots(b_n)(d)((b \Rightarrow [\![p_1']\!]) | [\![p_1'']\!])$.

7. Assume $Q \sim [\![p]\!]$ and $Q \xrightarrow{\tau} Q'$. Then $[\![p]\!] \xrightarrow{\tau} Q''$ with $Q' \sim Q''$ since $Q \sim [\![p]\!]$.
We proceed by induction on the length of the inference used to establish $[\![p]\!] \xrightarrow{\tau} Q''$ observing the structure of the process p.
If $[\![p]\!] \xrightarrow{\tau} Q''$ then p must have one of the following forms:

$p \equiv \tau.p_1$ Then by the tau-rule we have $p \xrightarrow{\tau} p_1$ and by the TAU-rule $[\![p]\!] \xrightarrow{\tau} [\![p_1]\!]$ which proves the lemma in this case.

$p \equiv p_1 + p_2$ **either** $[\![p_1]\!] \xrightarrow{\tau} Q''$ by a shorter inference. By induction we have $p_1 \xrightarrow{\tau} p_1'$ with $Q'' \sim [\![p_1']\!]$. By the sum-rule $p_1 + p_2 \xrightarrow{\tau} p_1'$ and by the SUM-rule we have $[\![p_1 + p_2]\!] \xrightarrow{\tau} Q''$

or $[\![p_2]\!] \xrightarrow{\tau} Q''$ and a similar argument as above applies.

$p \equiv p_1 | p_2$ **either** $[\![p_1]\!] \xrightarrow{\tau} Q_1''$ by a shorter inference and $Q'' \equiv Q_1'' | [\![p_2]\!]$. By induction we have $p_1 \xrightarrow{\tau} p_1'$ with $Q_1'' \sim [\![p_1']\!]$. By the par-rule $p_1 | p_2 \xrightarrow{\tau} p_1' | p_2$ and by the PAR-rule we have $[\![p_1 | p_2]\!] \xrightarrow{\tau} Q_1'' | [\![p_2]\!] \sim [\![p_1' | p_2]\!]$

or $[\![p_2]\!] \xrightarrow{\tau} Q_2''$ and an argument as above applies.

or $[\![p_1]\!] \xrightarrow{a(x)} Q_1'$ and $[\![p_2]\!] \xrightarrow{\bar{a}(b)} Q_2'$ by shorter inferences and $Q'' \sim (b)(Q_1'\{b/x\} | Q_2'')$ modulo the appropriate α-conversions. By Proposition 4.7.4 we have $p_1 \xrightarrow{a?x} p_1'$ with $Q_1'\{b/x\} \sim [\![p_1']\!]\{b/x\}$ for all $b \in Names$ and by Proposition 4.7.6 we have $p_2 \xrightarrow{a!_B p_2'} p_2''$ with $Q_2' \sim (b_1)\ldots(b_n)((b \Rightarrow [\![p_2']\!]) | [\![p_2'']\!])$. By the com-close-rule we have $p_1 | p_2 \xrightarrow{\tau} (p_1'[p_2'/x]_\tau | p_2'') \backslash B$ assuming $B \cap fn(p_1') = \emptyset$ (otherwise use a suitable α-conversion). By the COM-rule we have $[\![p_1 | p_2]\!] \xrightarrow{\tau} Q'' \sim (b)([\![p_1']\!]\{b/x\} | (b_1)\ldots(b_n)((b \Rightarrow [\![p_2']\!]) | [\![p_2'']\!])) \sim [\![(p_1'[p_2'/x]_\tau | p_2'') \backslash B]\!]$ according to Proposition 4.6.

or $[\![p_1]\!] \xrightarrow{\bar{a}(b)} Q_1'$ and $[\![p_2]\!] \xrightarrow{a(x)} Q_2'$ which is a symmetric case to the above.

$p \equiv p_1 \backslash b$ Then $[\![p_1]\!] \xrightarrow{\tau} Q_1''$ with $Q'' \equiv (b)(Q_1'')$ by a shorter inference. By induction $p_1 \xrightarrow{\tau} p_1'$ with $Q_1' \sim [\![p_1']\!]$. By the res-rule we have $p_1 \backslash b \xrightarrow{\tau} p_1' \backslash b$ and by the RES-rule we have $[\![p_1 \backslash b]\!] \xrightarrow{\tau} [\![p_1' \backslash b]\!]$. □