A Simple π-Calculus Manipulation Tool

Arie van Deursen

Formal Methods Group, Dept. of Computing Science
TU Eindhoven, 5600 MB P.O. Box 513, The Netherlands
arie@win.tue.nl, http://www.win.tue.nl/win/cs/fm/arie/

Abstract. The π-calculus is a challenge for tool generation software, as it is subject to frequent changes. A specification of desirable functionality of tools for manipulating π-calculus expressions is given, and its extensibility is studied. The difficulties encountered during specification are discussed, as well as several proposals for easier tool specification.
N.B. This paper describes ongoing work: Comments welcome!

Key Words & Phrases: π-calculus, algebraic specification, SOS rules, tool support, tool generation, executable specifications

Note: Supported by NWO project HOOP: Higher-Order and Object-Oriented Processes.

1 Introduction

1.1 The π-calculus

The π-calculus [MPW92] is theory for concurrent behavior. It is a calculus with a very small syntax, and some intuitively easy to understand rules for computation. These rules only involve *name passing*: Processes can transmit names to each other along named links. This name passing caters for two desirable features:

- It can be used to give a direct description of systems which change their connection configuration, such as a mobile telephone network or a distributed operating system. As an example, the π-calculus has been applied to verify the hand-over protocol of the GSM mobile telephone network (see [OP92] and Appendix B).
- It facilitates a form of *higher-order communication*, where (names of links to) processes are transmitted between agents [San92].

Neither of these were easily expressible in more classical process algebras such as CCS, CSP or ACP. Moreover, the π-calculus features [Mil93a]:

- A uniform way for defining data structures such as lists, numbers, Booleans, etc. [MPW92, Mil93b]
- A simple translation of the λ-calculus into the π-calculus [MPW92].
- A clear relation to object-oriented programming, where naming of messages, objects, and classes is essential [Wal94].
- Possibilities for an extension with a *type discipline* [Mil93b].

These arguments led the creators of the calculus to baptizing it the π-calculus: The λ-calculus has been successful for understanding functions and sequential programming; The hope is that the π-calculus could play a similar role for understanding the semantic ingredients of concurrency.

Proceedings of ASF+SDF95. A workshop on Generating Tools from Algebraic Specifications. May 11 & 12, 1995, CWI, Amsterdam, M.G.J. van den Brand, A. van Deursen, T.B. Dinesh, J.F.Th. Kamperman & E. Visser (eds.) Technical Report P9504, Programming Research Group, University of Amsterdam

1.2 Tools for the π-calculus

In this paper, we will be concerned with tool support for the π-calculus. Computer support when dealing with π-calculus should help in defining data structures [MPW92], telecommunication protocols [OP92], languages (λ-calculus, object-oriented [Wal94], a notation for design [Jon93], or even full Prolog [Li94]) using the π-calculus.

We expect the tools to perform syntactic checks, to allow for recursive agent definitions, to support some form of simulation, to enable one to manipulate π-calculus expressions conform the structural congruence or inference rules, and to help when wishing to show the existence of a bisimulation between to agents.

Such functionality is offered by MWB, the Mobility WorkBench [VM94]. It is a tool implemented in Standard ML with a terminal based interface, allowing one to step through a π-calculus expression, and capable of finding (open) bisimulations. The tools we will be discussing will reuse as much as is possible from the MWB; the precise relation will be discussed in Section 3.3.

Tools involving the π-calculus, including the MWB, are faced with one major problem: the theory underlying it, and the syntactic conventions used are far from stable. Papers discussing applications typically propose their own syntactic conveniences (the case statement, polyadicity, definitions based on pattern matching). In addition to that, the theory changes: the inference rules in [Mil93b], e.g., are rather different from the original ones in [MPW92].

Such a changing and unstable situation makes the π-calculus a suitable case-study for automatic *tool generation*. In the ideal case, the π-calculus would be described in an easily maintainable formalism, from which tools can be derived in one way or another. Changes in the π-calculus lead to simple changes in this description, and the tools are updated.

Working out this case study is the subject of this paper. We describe the essentials of the π-calculus in the specification formalism ASF+SDF (see [BHK89, Kli93, Deu94]) and use these to obtain a set of tools for manipulating π-calculus-expressions. We particularly study whether the properties of the π-calculus are sufficiently easily expressible in ASF+SDF, and to what extend future changes can be handled conveniently.

1.3 Overview

The structure of the paper is as follows:

- In Section 2 a brief introduction to the essentials of the π-calculus is given.
- Next, in Section 3, an analysis is presented of the requirements a π-calculus-tool should meet in order to be useful. Naturally, this analysis does not presuppose any implementation platform: this section can be read with ASF+SDF, Standard ML, C/Lex/Yacc, Tcl/Tk, or any other language in mind.
- Then, in Section 4, the implementation in ASF+SDF is described. It is discussed to what extend the specification is sufficiently flexible, and the tools sufficiently useful. In order to make this discussion as concrete as possible, some modules from the ASF+SDF specification are contained in the appendices.
- The most important section, finally, is 5, which elaborates on the experiences covered in Section 4. A series of proposals is presented, which all aim at improving tool specification and generation techniques.

This last section is the heart of the paper, and contains the new results. Moreover, the paper includes some remarks on the pragmatics of π-calculus specification, yet its main focus is on tool generation techniques.

2 The π-calculus

In this section we recapitulate the most important concepts of the π-calculus. We closely follow [MPW91].

2.1 Syntax

Let $\mathcal{N}$ be a set of *names*, and let x, y, z, w, v, u range over names. Moreover, we assume a set of *agent identifiers* ranged over by C, where each C has an *arity* $r(C)$.

The set of *agents* is defined as follows (we use P, Q, R to range over agents):

$$
\begin{array}{rclr}
P & ::= & \mathbf{0} & \text{(inaction)} \\
 & | & \overline{x}y.P & \text{(output prefix)} \\
 & | & x(y).P & \text{(input prefix)} \\
 & | & \tau.p & \text{(silent prefix)} \\
 & | & (\nu y)P & \text{(restriction)} \\
 & | & [x = y]P & \text{(match)} \\
 & | & P \mid Q & \text{(composition)} \\
 & | & P + Q & \text{(summation)} \\
 & | & C(y_1, \ldots, y_{r(C)}) & \text{(defined agent)}
\end{array}
$$

The order of precedence among the operators is the order listed above. Each agent constant C has a unique *defining equation* of the form

$$C(x_1, \ldots, x_{r(C)}) := P$$

In both $x(y).P$ and $(\nu y).P$ the occurrence of y is a *binding* occurrence whose scope is P. The read action $x(y).P$ can accept a value for y resulting in a substitution of that value for the occurrences of y in P. The restriction operator $(\nu y).P$ makes all names y in P local to just this process. We write $\mathrm{fn}(P)$ for the names occurring free in P.

2.2 Actions

The effect of π-calculus expressions can be described by *transitions* of the form $P \stackrel{\alpha}{\longrightarrow} Q$. Four different *actions* α are possible: (1) the *silent* action τ indicating that a successful communication took place, (2) the *input* action $x(y)$ which reads in a value for y along channel x, (3) the *free output* action $\overline{x}y$ emitting the name y over channel x, and (4) the *bound output* action $\overline{x}(y)$, which occurs if a *local* (i.e., restricted) name is communicated to the outer world (referred to as *scope extrusion*).

For an action α, we define its *free names* $\mathrm{fn}(\alpha)$ and *bound names* $\mathrm{bn}(\alpha)$ as:

$$
\begin{array}{lcllcl}
\mathrm{fn}(\tau) & = & \emptyset & \mathrm{bn}(\tau) & = & \emptyset \\
\mathrm{fn}(x(y)) & = & \{x\} & \mathrm{bn}(x(y)) & = & \{y\} \\
\mathrm{fn}(\overline{x}y) & = & \{x, y\} & \mathrm{bn}(\overline{x}y) & = & \emptyset \\
\mathrm{fn}(\overline{x}(y)) & = & \{x\} & \mathrm{bn}(\overline{x}(y)) & = & \{y\}
\end{array}
$$

$$\text{ACT: } \frac{-}{\alpha.P \xrightarrow{\alpha} P} \qquad \text{SUM: } \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'}$$

$$\text{COM: } \frac{P \xrightarrow{\overline{x}y} P', Q \xrightarrow{x(z)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'[y/z]} \qquad \text{CLOSE: } \frac{P \xrightarrow{\overline{x}(y)} P', Q \xrightarrow{x(y)} Q'}{P \mid Q \xrightarrow{\tau} (\nu y)(P' \mid Q')}$$

$$\text{RES: } \frac{P \xrightarrow{\alpha} P'}{(\nu y)P \xrightarrow{\alpha} (\nu y)P'} \; y \notin \mathrm{n}(\alpha) \qquad \text{OPEN: } \frac{P \xrightarrow{\overline{x}y} P'}{(\nu y)P \xrightarrow{\overline{x}(y)} P'} \; y \neq x$$

$$\text{PAR: } \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \; \mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \emptyset$$

Figure 1: Transition rules for the π-calculus

2.3 Structural Congruence

Before defining which actions are possible for π-calculus expressions, we define *structural congruence* $\equiv$ on agents:

1. If P and Q differ only in the choice of bound names, i.e., they are alpha-equivalent in the standard sense, then $P \equiv Q$.
2. $P \mid Q \equiv Q \mid P$, $P \mid \mathbf{0} \equiv P$
3. $P + Q \equiv Q + P$, $P + \mathbf{0} \equiv P$
4. $[x = x]P \equiv P$
5. If $C(x_1, \ldots, x_n) := P$ then $C(y_1, \ldots, y_n) \equiv P[y_1/x_1, \ldots, y_n/x_n]$, where $n = r(C)$.

2.4 Transitions

A *variant* of the transition $P \xrightarrow{\alpha} Q$ is a transition which only differs in that P and Q have been replaced by structurally congruent agents, and α has been alpha-converted, where a name bound in α includes Q in its scope.

The rules for inferring transitions from π-calculus-expressions are given in Figure 1. In each rule, the transition in the conclusion stands for all variants of the transition.

2.5 Bisimulation

Two π-calculus-expressions are considered equal if they can "mimic" each other, i.e., one can simulate the other and vice versa. More formally: A binary relation $\mathcal{S}$ on agents is a *late simulation* if $P\mathcal{S}Q$ implies that:

1. If $P \xrightarrow{\alpha} P'$ and $\alpha \in \{\tau, \overline{x}z, \overline{x}(y)\}$ with $y \notin \mathrm{fn}(P, Q)$ then for some Q', $Q \xrightarrow{\alpha} Q'$ and $P'\mathcal{S}Q'$.
2. If $P \xrightarrow{x(y)} P'$ and $y \notin \mathrm{fn}(P, Q)$, then for some Q', $Q \xrightarrow{x(y)} Q'$ and for all w, $P'[w/y]\mathcal{S}Q'[w/y]$.

A relation $\mathcal{S}$ is a *late bisimulation* if both $\mathcal{S}$ and its inverse $\mathcal{S}^{-1}$ are late simulations.

2.6 Extensions and Further Developments

When used for more practical purposes, the π-calculus is extended with certain useful abbreviations. As in CCS, the trailing **0** is usually omitted. Other examples are:

- Polyadic input and output can be modeled by introducing [Mil93b, p.10]:

$$\begin{aligned} x(y_1 \cdots y_n) &= x(w).w(y_1).\cdots.w(y_n) \\ \overline{w}y_1 \cdots y_n &= (\nu w)\overline{x}w.\overline{w}y_1 \cdots .\overline{w}y_n \end{aligned}$$

  where $n \geq 0$.

- Case distinction can be defined by [MPW92, p.14,19]:

$$x : [y_1 \Rightarrow P_1, y_2 \Rightarrow P2, \cdots] = x(v).([v = y_1]P_1 + [v = y_2]P_2 + \cdots)$$

  or, if we want to receive extra arguments over x:

$$x : [\ldots, v(y_1 \cdots y_n) \Rightarrow P, \ldots] = x : [\ldots, v \Rightarrow x(y_1 \cdots y_n)P, \ldots]$$

- Restricted output can be written $\overline{x}(y)$, and defined as:

$$\overline{x}(y).P = (\nu y)\overline{x}y.P$$

For an example usage of these constructs we refer to Appendix B and [OP92].

In [Mil93b] an alternative approach to polyadicity is presented. Instead of taking monadic prefixes (with just *one* argument) as fundamental, prefixes with *no* arguments are used. After each prefix a so-called *abstraction* or *concretion* can occur. An input action $x(v_1 \cdots v_n).P$ is written

$$x.(\lambda v_1 \cdots v_n)P$$

which is an expression that is *committed* to synchronize over the location x, after which it will wait for n inputs as expressed by the n-ary abstraction. Likewise, output $\overline{x}y_1 \cdots y_n$ is written

$$\overline{x}.[y_1 \cdots y_n]P$$

which after synchronizing over x will send out n values, as indicated by the *abstraction*$[y_1 \cdots y_n]P$.

These abstractions and concretions have been introduced because (1) the inference rules become much simpler (in particular Open and Res can be combined into just Res), and (2) it allows for a *sorting* (or typing) mechanism over the π-calculus. Moreover, in [Mil93b] some extra structural congruence rules are given, which allow for a further simplification of the inference rules. The most notable one is:

$$(\nu x)(P|Q) \equiv P|(\nu x)Q$$

It can be used to move restrictions outward or inward. For the inference rules, it catches the behavior of the (much more complicated) Close rule.

Other recent papers on the π-calculus have proposed further extensions. Some examples include the πI-calculus [San95], *open* bisimulations [San93], action structures, and π-nets.

3 Requirements Analysis

3.1 Aim and User Group

Tool support for the π-calculus should be helpful for:

- Understanding simple manipulations on π-calculus expressions, such as reduction, communication, or commitment.
- Analyzing interesting encodings in the calculus of, e.g., numbers, combinators, lists, λ-calculus, ...
- Specifying and verifying applications in the area of mobile telecommunication, such as the hand-over procedure in GSM networks [OP92].
- Experimenting with extensions of the π-calculus.

Users of such a system might include students and teachers, researchers, and people who wish to use the π-calculus for specification or verification purposes.

3.2 Intended Functionality

A useful π-calculus tool should meet the following criteria:

- Syntax-directed editing of π-calculus expressions.
- Performing transformations according to the structural congruence relation, inference rules, and the algebraic laws given for the π-calculus [MPW92].

  A user should see which laws or rules are applicable, and then must be able to select one of these.
- Interactive entry of recursively defined agents.

  All definitions such as occurring in, e.g., [OP92, MPW92] must be directly expressible.

  When manipulating a π-calculus expression, expansion of defined agents to their full body must be possible. However, if it is not strictly necessary to do so (i.e., the expression can make a step without expanding the definition), the tool should not require it.
- Computation of all possible (modulo α-conversion) commitments.
- Automatic testing for bisimulation between two agents.
- Automatic simplification of expressions to "normal forms" (if possible).
- Asking simple questions about π-calculus expressions. Is this name occurrence free or bound? Is this expression guarded? Friendly? Name-bearing? Observable? (see [Mil93b] for the definitions of these)
- Support for writing "larger" π-calculus specifications. This includes a some form of modularization, and the possibility to indicate explicitly which names act like *constants* and which ones as real variables.

  Moreover the abbreviations like the ones discussed in Section 2.6 should be included, and users must easily be able to extend the tool with other useful abbreviations.
- Static analysis of agent declarations, which amounts to checking whether all agent identifiers are declared, and used according to the declarations.

- Easy extensibility to new π-calculus varieties, such as the polyadic π-calculus.
- Editor support for fonts (e.g.: agents in italic, constants in small caps, inaction in bold face, other names in teletype), Greek symbols (ν), and over-lining for output actions.
- Generation of pictorial representations, such as π-nets [Mil94] or interaction diagrams [Par93].

3.3 Related Work: The Mobility Workbench

The best known existing tool for the π-calculus at this moment is MWB, the *Mobility Workbench* [VM94]. It is an interactive tool with a textual interface. It is implemented in Standard ML.

In MWB, recursive agent definitions can be given. The basic functionality is to decide the *open equivalences* of Sangiorgi [San93], for agents in the polyadic π-calculus with the positive match operator. Moreover, a "step" command can be given, which interactively simulates an agent, by presenting the possible commitments of the agent and letting the user select one, repeating this until there are no further commitments.

The MWB has put quite some ingenuity in its bisimilarity decision procedure, and the stepping facility is very useful. The tools we will be proposing are more tailored towards using equations, inference rules, and congruence rules for manipulating π-calculus expressions. It will be clear, however, that we wish to benefit from the MWB. Therefore, a last requirement is that there must exist an automatic translation from the notations we propose to the mobility workbench.

4 Implementation

We now proceed to discuss an example implementation, using the ASF+SDF algebraic specification formalism to describe the details of the functionality desired, and the ASF+SDF Meta-environment to obtain tools from such a specification. For an introduction to ASF+SDF we refer to [BHK89, Deu94]; The latter describes the specification of a tool for the λ-calculus which bears some similarity with what is being done here.

The reason for choosing ASF+SDF is that its specifications are easily extended with additional functionality, a requirement formulated in the previous section. Moreover, its implementation is based on *reduction*, which seems to fit in reasonably well with the congruence, commitment, and equivalence relations defined over π-calculus expressions. Last but not least, syntax-directed editors come for free in the ASF+SDF Meta-environment.

4.1 Context-Free Syntax

The syntax definition of Section 2.1 can be directly translated to SDF (see Module A.1.2). Straightforward agent identifiers are introduced in Module A.2.1. Together with the definitions of Module A.2.4, simple agents can be defined, like

$$Buf(i,o) = i(x).\overline{o}x.Buf(i,o)$$

More interesting is the notation for introducing semantic functions. A typical definition given in [MPW92, p.19] is the following:

$$\begin{array}{rcl} [\![\mathrm{nil}]\!](x) & := & \overline{x}.\mathtt{nil} \\ [\![\mathrm{cons}(L_1,L_2)]\!](x) & := & (\nu y)(\nu z)(\overline{x}\mathtt{cons}.\overline{x}y.\overline{x}z|[\![L_1]\!](y)|[\![L_2]\!](z)) \end{array}$$

Entry	Functionality
Alpha	$(\nu y)P$ $x(y).P$ Ask for fresh y' and replace y by it.
Swap	$P + Q \rightarrow Q + P,$ $P \mid Q \rightarrow Q \mid P,$ $(\nu x)(\nu y)P \rightarrow (\nu y)(\nu x)P$
Assoc	$(P + Q) + R \leftrightarrow P + (Q + R),$ $(P \mid Q) \mid R \leftrightarrow P \mid (Q \mid R)$
Match	$[x = x]P \rightarrow P$
Defined	Replace $C(x_1, \ldots x_n)$ by its instantiated body.
Scope	$(\nu x)(P \mid Q) \leftrightarrow P \mid (\nu x)Q \quad (x \notin fn(P))$

Figure 2: Button Entries for Structural Congruence

The agent defined includes some term built from the constructors "nil" and "cons", where the brackets $[\![\cdots]\!]$ denote a semantic function mapping such terms to the π-calculus. The notation[1] allowing this is specified in Module A.2.2.

In some situations, terms occurring in such semantic definitions recur as π-calculus name. For example, [MPW92, p.25] defines:

$$\begin{aligned} [\![\lambda x M]\!](u) &:= u(x,v).[\![M]\!](v) \\ [\![x]\!](u) &:= \overline{x}u \end{aligned}$$

Observe that the variable x occurs free in the translation of the λ-term x (the last equation), and therefore will usually occur free in the translation of M in the first definition (but will be bound by $u(x,v)$). This reuse of names from the term domain in the π-calculus domain is allowed by the definitions of Module A.2.4, which introduces a \$-sign to distinguish such shared names. Using that notation, the former definitions read:

```
[[ lambda($x,M) ]](u) := u($x,v).[[M]](v)
           [[ $x ]](u) := $x!u
```

A last syntactic proposal involves constant declarations. In the definition of the cons and the nil given before, the names `cons` and `nil` occurring in the right-hand sides act as constants. To indicate this, a specifier can include a line like

```
constants: nil cons
```

4.2 Some Abbreviations

In Section 2.6 we discussed extensions like polyadicity, zero-adic communication, and some varieties of a case statement. Their syntax and translation to the core π-calculus is straightforward in ASF+SDF, and shown in Modules A.3.1, A.3.2, and A.3.3.

An observation to make is that these translations are automatic, as they are specified as rewrite rules, which will immediately eliminate these new constructs. As a result, other functions defined over π-calculus expressions (computation of bound or free names, substitution, etc.) need not be adapted because of these adaptations.

[1] Observe that the definitions using these should be *compositional*: a π-calculus "type checker" should check this.

Entry	Functionality
SUM-L	$P + Q \to P$
SUM-R	$P + Q \to Q$
COM	$(\overline{x}y.P) \mid x(z).Q \to \tau.(P \mid (Q[z/y]))$
RES	$(\nu y)(\alpha.P) \to \alpha.(\nu y)P \quad (y \notin n(\alpha))$
PAR	$(\alpha.P) \mid Q \to \alpha.(P \mid Q) \quad (bn(\alpha) \cap fn(Q) = \emptyset)$
OPEN	$(\nu y)(\overline{x}y.P) \leftrightarrow \overline{x}(y).P \quad (x \neq y)$
CLOSE	$\overline{x}(y).P \mid x(y).Q \to \tau.(\nu y)(P \mid Q)$

Figure 3: Menu Entries for Inference Rules

4.3 Structural Congruence

The context-free syntax of Section 4.1 permits simple syntax-directed editing. A user will also wish to be able to replace (sub)agents by structurally congruent alternatives in an interactive manner. To that end, the π-calculus syntax-directed editor can be extended with, e.g., a pulldown menu with entries to perform such replacements. Figure 2 lists a number of entries together with their associated functionality, as needed for replacing structurally congruent agents.

In ASF+SDF, this table can be implemented by an explicit function (e.g.: swap(AGENT)$\to$ AGENT) for each button entry, which replaces its agent if it matches one of the cases mentioned in the table, and returns it unchanged otherwise. Conditional equations can be used to express a rule like Scope. Subsequently, these functions can be attached to user interface events using the SEAL language (Semantics-Directed Environment Adaptation Language: see [Koo94] or [Deu94, Section 2.4.3])

In Section 5 we will return to this subject and discuss a more attractive definition method.

4.4 Inference Rules

The transitions $P \xrightarrow{\alpha} Q$ can be regarded as a transformation of P into $\alpha.Q$. This requires the addition of an extra prefix $\overline{x}(y)$ for restricted output (which we also encountered in Section 2.6). The menu entries for allowing interactive replacement of agents by their reducts is given in Figure 3. The specification based on this figure is given in Module A.4.

Observe that such inferences do not preserve (a form of) equality. A more accurate treatment is $P \xrightarrow{\alpha} Q$ by $P \to P + \alpha.Q$. It seems that the tool is easier useable with the approach chosen, when it is being used to trace down one particular execution path. Clearly necessary is a "duplication" button, which transforms an agent P into $P + P$ (see the next section) and can be used to find alternative execution paths.

4.5 Algebraic Laws

Given the inference rules from Figure 1 and the notion of bisimulation of Section 2.5, one can prove certain equivalences between agents. These so-called algebraic laws are described in [MPW92, OP92]

We will use the same approach for the algebraic laws as for structural equivalence: we define a number of buttons which can transform an agent in correspondence with a given law. The menu entries are listed in Figure 4. If the so-called τ-laws (concerning the silent prefix τ) are taken into account as well, one can use

Entry	Functionality
Sum0	$P + \mathbf{0} \to P$
SumPP	$P + P \to P$
SumDouble	$P \to P + P$
NoMatch	$[x = y]P \to \mathbf{0}$ when $x \neq y$.
Scope	$(\nu x)P \to P$ **when** $x \notin P$, $(\nu x)(P + Q) \leftrightarrow (\nu x)P + (\nu x)Q$ $(\nu x)\overline{x}y.P \to \mathbf{0}$ $(\nu x)x(y).P \to \mathbf{0}$
Expansion	Generalization of $x(y).P\vert\overline{x}z.Q \to x(y).(P\vert\overline{x}z.Q) + \overline{x}z.(x(y).P\vert Q) + \tau.(P[z/y]\vert Q)$

Figure 4: Menu Entries for Algebraic Laws.

Entry	Functionality
Internal	$\alpha.\tau.P \leftrightarrow alpha.P$
Summand	$P + \tau.P \leftrightarrow \tau.P$
Double	$\alpha.(P + \tau.Q) + \alpha.Q) \leftrightarrow \alpha.(P + \tau.Q)$

Figure 5: Menu Entries for τ-Laws.

the algebraic laws (see Figure 5) to prove the equivalence between an abstractly defined agent specification and a more detailed agent implementation.

There is one important point to make here. The π-calculus notion of bisimilarity is *not* a congruence. In particular, input prefix does not preserve equality. Therefore the congruence formulation given in [OP92] reads:

C0 All operators except input preserve $=$

C1 If $P[y := z] = Q[y := z]$ for all z then $x(y).P = x(y).Q$

(In practice one only needs to check all z occurring free in P or Q, and one z not in P or Q).

It is unclear what the best way is to enforce this in a tool where buttons are used to replace agent terms by other agent terms. One approach could be to forbid replacement of equals by equals under a prefix, unless the user has indicated that he has checked that it is safe to do so. Again, for a further discussion of this problem we refer to Section 5.

4.6 Mobility WorkBench

A translation to the Mobility Workbench makes it possible to reuse MWB's simulation and bisimulation checking facilities. A translation from the syntax of Sections A.1 and A.2 to a syntax of the MWB is easily expressed in ASF+SDF. The syntactic extensions of Section A.3 can all be reduced to core π-calculus, so the tranlation to MWB need not cater for these. Elimination of polyadicity is not necessary, as this is supported by MWB.

The MWB requires that all names occurring in the right-hand side of an agent definition are bound, either by restriction or input, or as one of the formal parameters. The translation must therefore extend the parameter lists with those names declared as constants.

4.7 Remaining Requirements

There are some requirements from Section 3 which have not yet discussed.

One of them concerns the checks on guardedness of agents, or on free or bound occurrences of names. Given a full term, it is possible to yield a list of names that occur freely, or of agents that are guarded. Far more difficult is the setting where a user indicates a given subagents or name, and asks whether this particular occurrence is guarded or free.

The extension to static analysis of agent declarations is straightforward. Editor support for fonts is at the moment impossible: on the other hand it is possible to generate LaTeX type setters [BV94] (e.g., we intend to generate such a type setter and use it to improve the verbatim code of Appendix B) Generation of pictures is not yet supported either: a translation to some visual language should be relatively easy (we are planning efforts in the direction of [Üsk94].

5 Easier Tool Generation

5.1 Occurrence Algebra

One of the nice things of algebraic specifications is that the notion of *occurrences* is well-hided: terms are considered as such, and it should make no difference in which context they occur. However, as we have seen in the previous section, in some cases this is too restrictive: one would like to get hold of a context (to walk a tree upwards). When doing so, it is often necessary to remember "where one came from".

A more concrete example is the following. A user selects a name occurrence. Is it bound or free? To answer this question, walk the tree upwards, and see if one encounters a binder (input or restriction). Alternatively, find some representation of occurrences (e.g.: a sequence of numbers indicating the path from the root, as in [DKT93]), and compute a set with all occurrences of free names. The particular occurrence is free if it is an element of this set.

How can such a setting be achieved? A possibility is to have a special operation on sorts, which transforms a sort to its "occurrence" variant. Terms over this sort are pairs of a term of the original sort and its an occurrence in a term. Occurrence-terms in equations can be propagated using the origin tracking rules of [DKT93].

Such a setting makes treatment of bound or free occurrence easy, as well as checks on guardedness. Moreover, it can be used to check whether a given subterm occurs under an input prefix, and thus which algebraic laws can be used. The setting is applicable outside the area of π-calculus as well: think of the scope rules in arbitrary languages.

5.2 Specifying Non-Determinism

The ASF+SDF implementation is based on *reduction*, yet it is not so easy to specify the reduction relations occurring in the π-calculus conveniently: ASF+SDF specifies equality rather than a reduction relation.

Non-determinism is particularly hard to define: One might be tempted to think that the fact that in an ASF+SDF specification the order of the rewrite rules is not taken into account guarantees a certain amount of non-determinism. However, this only makes the ASF+SDF rewrite machinery unpredictable: semantically it equates the various possibilities.

Needed is a setting where rewrite systems are intentionally non-confluent. Unique normal forms are obtained by taking set of all possible normal forms of a term. If one is just interested in picking one element from such a set the whole ASF+SDF rewriting mechanism can remain as it is: the only thing needed is a possibility to mark certain rewrite rules as semantically different.

5.3 Interactive Equations

Some equations, like $X + Y = Y + X$ or $!P = !P|P$ have a non-terminating effect on the rewrite behavior of an ASF+SDF specification. In our specification, we do need the effect of these equations, but we are satisfied if we only can use them interactively.

A useful extension of ASF+SDF would be the possibility to mark certain rewrite rules as "interactive", which means that whenever such a rule is the only applicable rule, the reduction process stops and waits until a user has agreed to apply this rule. If there are several interactive rules available, the user can choose the one to apply. For many specifications this would be a useful debugging aid as well (making critical equations temporarily interactive).

5.4 Multiple Focusing

In a series like $\alpha_1.P_1|\cdots|\alpha_n.P_n$, a communication can be possible between any two agents $\alpha_i.P_i$ and $\alpha_j.P_j$ $(1 \leq i, j \leq n, i \neq j)$, where one of the $\alpha_{i,j}$ is an input and the other an output action, and the subject of these (the port used) is the same.

In an interactive editor, a user might wish to indicate two such agents and initiate the communication between them. An approach to allow this could be supporting multiple focuses. A user would put a focus on the ith and jth agent, and click on some button to initiate a communication. The system should then check that the agents are indeed occurring in a series of communication merges, and that they include an input and output action over the same channel.

To get this behavior, it should only be necessary to specify the associativity and commutativity of $|$, the result of performing a communication between two agents (i.e., the *com* function of Module A.4).

A further extension might be to support restricted output as well: After alpha-converting the input variable and the restricted name to be sent out to a single new fresh name the CLOSE inference rule can be applied. Again, it must be possible to define this just for two agents, and get the effect for arbitrary agents in a communication series for free.

References

[Bar84] H.P. Barendregt. *The Lambda Calculus; its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathatematics.* North-Holland, 1984.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[BV94] M. van den Brand and E. Visser. From Box to TEX: An algebraic approach to the generation of documentation tools. Technical Report P9420, Programming Research Group, University of Amsterdam, 1994.

[Deu94] A. van Deursen. *Executable Language Definitions: Case Studies and Origin Tracking Techniques.* PhD thesis, University of Amsterdam, 1994.

[DKT93] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15:523–545, 1993. Special Issue on Automatic Programming.

[Jon93] C. Jones. A π-calculus semantics for an object-based design notation. In E. Best, editor, *Proceedings CONCUR'93*, volume 715 of *LNCS*, pages 158–172. Springer-Verlag, 1993.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[Koo94] J.W.C. Koorn. *Generating Uniform User-Interfaces for Interactive Programming Environments*. PhD thesis, University of Amsterdam, 1994.

[Li94] B.Z. Li. A π-calculus specification of Prolog. In D. Sanella, editor, *Programming Languages and Systems – ESOP'94*, volume 788 of *LNCS*, pages 379–393. Springer-Verlag, 1994.

[Mil93a] R. Milner. Elements of interaction. *Communications of the ACM*, 36(1):78–89, 1993. Turing Award Lecture.

[Mil93b] R. Milner. The polyadic π-calculus: a tutorial. In F.L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*. Springer-Verlag, 1993. Also available from ftp: //www.dcs.ed.ac.uk /staff/rm.

[Mil94] R. Milner. Pi-nets: A graphical form of π-calculus. In D. Sannella, editor, *Programming Languages and Systems – ESOP'94*, volume 788 of *LNCS*, pages 26–42, 1994.

[MPW91] R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. In J.C.M. Baeten and J.F. Groote, editors, *Proceedings CONCUR'91*, volume 527 of *LNCS*, pages 45–60. Springer-Verlag, 1991.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Control*, pages 1–77, 1992.

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

[Par93] J. Parrow. Interaction diagrams. In J.W. de Bakker, W.-P de Roever, and G Rozenberg, editors, *A Decade of Concurrency: REX*, volume 803 of *LNCS*, pages 477–508. Springer-Verlag, 1993.

[San92] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, University of Edinburgh, 1992. Report CST-99-93.

[San93] D. Sangiorgi. A theory of bisimimulation for the π-calculus. In E. Best, editor, *Proceedings of CONCUR'93*, volume 715 of *LNCS*, pages 127–142. Springer-Verlag, 1993.

[San95] D. Sangiorgi. π-Calculus, internal mobility, and agent-passing calculi. Technical report, University of Edinburgh, 1995. To appear in TAPSOFT'95.

[Üsk94] Susan Üsküdarlı. Generating visual editors for formally specified languages. In *Proc. 1994 IEEE Symposium Visual Languages*, October 1994.

[VM94] B. Victor and F. Moller. The Mobility Workbench — a tool for the π-calculus. In D. Dill, editor, *Proceedings Computer-Aided Verification – CAV'94*, LNCS. Springer-Verlag, 1994.

[Wal94] D. Walker. Objects in the π-calculus. *Information and Computation*, 1994. To appear.

A A Selection from the Implementation

Here we present some of the more interesting modules from the π-calculus specification, allowing the reader to assess the extensibility. As soon as the tools are in more stable state, the full specification sources will be made available through WWW.

A.1 The Syntax of the π-calculus

A.1.1 Prefix

Define the syntax of the input, output, and silent prefix. Names used in them are alphanumeric tokens starting with a lowercase letter.

imports Layout$^{(A.5.1)}$
exports
  sorts NAME-ID NAME PREFIX
  lexical syntax
    [a-z][\-_a-zA-Z0-9′]∗ → NAME-ID
  context-free syntax
    NAME-ID → NAME
    NAME “!” NAME → PREFIX
    NAME “(” NAME “)” → PREFIX
    “τ” → PREFIX
  variables
    “′”*[xyzuvw][0-9′]∗* → NAME
    “α”*[0-9]∗* → PREFIX

A.1.2 Agents

The definition of agents in the π-calculus.

imports Prefix$^{(A.1.1)}$
exports
  sorts AGENT
  context-free syntax
    “**0**” → AGENT
    PREFIX “.” AGENT → AGENT
    “(” “ν” NAME “)” AGENT → AGENT
    “[” NAME “=” NAME “]” AGENT → AGENT
    AGENT “|” AGENT → AGENT {**left**}
    AGENT “+” AGENT → AGENT {**left**}
    “(” AGENT “)” → AGENT {**bracket**}
  priorities
    {“.”, “(” “ν” “)”, “[” “=” “]”} > “|” > “+”
  variables
    “′”*[PQRS][0-9′]∗* → AGENT

A.2 Support for writing larger π-scripts

A.2.1 Ident

Define parameterized agent identifiers, which start with a capital letter. The formal parameters are just names of the π-calculus.

imports Prefix$^{(A.1.1)}$
exports
sorts IDENT AGENT-ID NAMES AGENT-PAR
lexical syntax
[A-Z][A-Z\-a-z0-9′]∗ → IDENT
context-free syntax
IDENT → AGENT-ID
{NAME ","}∗ → NAMES
AGENT-ID "(" NAMES ")" → AGENT-PAR
variables
"′"*[AB][0-9]∗* → AGENT-ID
"′"[xyz]","*[0-9]∗*"∗" → {NAME ","}∗
"′"*I [0-9]∗* → AGENT-PAR

A.2.2 Terms

Extend the agent identifiers such that agents of the form $[\![f(X_1,\ldots,X_n)]\!]$ are allowed as well. To that end, define a sort *TERM* which can be used inside [[...]] brackets to form a new agent identifier.
imports Ident$^{(A.2.1)}$
exports
sorts TERM FUN-SYM VAR-SYM TERMS
lexical syntax
[A-Z][a-zA-Z0-9′\−]∗ → VAR-SYM
[a-z][a-zA-Z0-9′\−]∗ → FUN-SYM
context-free syntax
VAR-SYM → TERM
{TERM ","}∗ → TERMS
FUN-SYM "(" TERMS ")" → TERM
"⟦" TERM "⟧" → AGENT-ID

A.2.3 Parname

Allow parameterized names to occur in term expressions under [[...]] identifiers, and as names in π-calculus expresions.
imports Terms$^{(A.2.2)}$ Agents$^{(A.1.2)}$
exports
sorts PAR-NAME
context-free syntax
"*$*" NAME-ID → PAR-NAME
PAR-NAME → NAME
PAR-NAME → TERM

A.2.4 A-defs

Introduce syntax for defining Agents, and for using defined agents.
imports Agents$^{(A.1.2)}$ Terms$^{(A.2.2)}$ Ident$^{(A.2.1)}$ Parname$^{(A.2.3)}$
exports
context-free syntax
AGENT-PAR → AGENT

Syntax for the definition of agent identifiers. The optional were clause can be used to obtain a variable not yet occurring in an Agent Identification term.
exports
sorts PI-DEF WHERE WHERE-CLAUSE

context-free syntax
AGENT-PAR ":=" AGENT WHERE → PI-DEF
where WHERE-CLAUSE∗ → WHERE
PAR-NAME ":=" *fresh* "(" TERM ")" → WHERE-CLAUSE
→ WHERE

A.2.5 Pi-spec

Provide syntax for writing π-calculus specifications.
imports A-defs$^{(A.2.4)}$
exports
sorts PI-LINE PI-SPEC
context-free syntax
PI-DEF → PI-LINE
constants ":" NAME∗ → PI-LINE
interface ":" NAME∗ → PI-LINE
begin PI-LINE∗ *end* → PI-SPEC

A.3 Macros

A.3.1 Trailing

Treat expressions without trailing **0** as if they were written with it [MPW92, p.14]
imports Agents$^{(A.1.2)}$
exports
context-free syntax
PREFIX → AGENT
equations

$$\alpha \;=\; \alpha \,.\, \mathbf{0} \qquad [1]$$

A.3.2 Multiple

Generalize several operators such that then can work with zero or more than one argument as well. Here we only show the input prefix.
imports Agents$^{(A.1.2)}$ Fn-bn$^{(A.5.2)}$
exports
context-free syntax
NAME → PREFIX
NAME "(" NAME NAME+ ")" → PREFIX
variables
"'"[xyz][*0-9*]∗"+" → NAME+
hiddens
context-free syntax
poly-in(NAME, NAME+, AGENT) → AGENT
equations
The input action without arguments (which is in fact a synchronization action).

$$\text{'x} \,.\, \text{'}P \;=\; \text{'x}(\mathit{get\text{-}fresh}(\mathtt{y}, \mathit{fn}(\text{'}P))) \,.\, \text{'}P \qquad [1]$$

Polyadic I/O, [Mil93b, p.10]. The auxiliary function "poly-in" is used to traverse all arguments (like a map-function).

$$\frac{\text{'w} = \mathit{get\text{-}fresh}(\mathtt{w}, \mathit{fn}(\text{'}P))}{\text{'x}(\text{'}y_1 \;\, \text{'}y^+) \,.\, \text{'}P = \text{'x}(\text{'w}) \,.\, \mathit{poly\text{-}in}(\text{'w}, \; \text{'}y_1 \;\, \text{'}y^+, \; \text{'}P)} \qquad [2]$$

$$poly\text{-}in(\text{'w}, \text{'y}, \text{'}P) \quad = \quad \text{'w('y)} \;.\; \text{'}P \tag{3}$$

$$poly\text{-}in(\text{'w}, \text{'y 'y}^+, \text{'}P) \;=\; \text{'w('y)} \;.\; poly\text{-}in(\text{'w}, \text{'y}^+, \text{'}P) \tag{4}$$

A.3.3 Macros

Define (three varieties of) a case statement, taken from [MPW92, p.14].
imports Agents$^{(A.1.2)}$ Fn-bn$^{(A.5.2)}$ Trailing$^{(A.3.1)}$ Multiple$^{(A.3.2)}$
exports
sorts CASE
context-free syntax
NAME ":" "[" {CASE ","}* "]" → AGENT
NAME "⇒" AGENT → CASE
NAME "(" NAME+ ")" "⇒" AGENT → CASE
NAME "[" NAME "]" → PREFIX
hiddens
context-free syntax
get-proc({CASE ","}*) → AGENT
do-cases(NAME, NAME, {CASE ","}*) → AGENT
variables
Case [*0-9'*]* → CASE
Case [*0-9'*]*"*" → {CASE ","}*
equations
Case analysis. Read some name over a channel $'x$, and take action depending on the actual name received.

$$\frac{\text{'v} = get\text{-}fresh(\mathtt{v},\, fn(get\text{-}proc(Case^*)))}{\text{'x :[}Case^*\text{]} = \text{'x('v)} \;.\; do\text{-}cases(\text{'x}, \text{'v}, Case^*)} \tag{1}$$

The notation $u[y].P$ is an abbreviation [OP92, p.506] for the case where we have only one case element:

$$\text{'u['y]} \;.\; \text{'}P \;=\; \text{'u :['y} \Rightarrow \text{'}P] \tag{2}$$

Translate each case to a positive matching agent.

$$do\text{-}cases(\text{'x}, \text{'v},) \quad = \quad \mathbf{0} \tag{3}$$

$$do\text{-}cases(\text{'x}, \text{'v}, \text{'y} \Rightarrow \text{'}P, Case^*) \;=\; [\text{'v} = \text{'y}]\; \text{'}P + do\text{-}cases(\text{'x}, \text{'v}, Case^*) \tag{4}$$

Translate cases with extra inputs to ordinary cases.

$$\begin{gathered} do\text{-}cases(\text{'x}, \text{'v}, \text{'y('z)} \Rightarrow \text{'}P, Case^*) = \\ do\text{-}cases(\text{'x}, \text{'v}, \text{'y} \Rightarrow \text{'x('z)} \;.\; \text{'}P, Case^*) \end{gathered} \tag{5}$$

$$\begin{gathered} do\text{-}cases(\text{'x}, \text{'v}, \text{'y('z 'z}^+) \Rightarrow \text{'}P, Case^*) = \\ do\text{-}cases(\text{'x}, \text{'v}, \text{'y} \Rightarrow \text{'x('z 'z}^+) \;.\; \text{'}P, Case^*) \end{gathered} \tag{6}$$

An axiliary function to translate all processes occurring in an case into one summation.

$$get\text{-}proc() \quad = \quad \mathbf{0} \tag{7}$$

$$get\text{-}proc(\text{'y} \Rightarrow \text{'}P, Case^*) \;=\; \text{'}P + get\text{-}proc(Case^*) \tag{8}$$

A.4 Infer

Specify the functionality needed for manipulating π-calculus expressions according to the inference rules.
imports Agents$^{(A.1.2)}$ Subs$^{(A.5.3)}$ Fn-bn$^{(A.5.2)}$
exports
  context-free syntax
    NAME "!" "(" NAME ")" → PREFIX
    sum-l(AGENT) → AGENT
    com(AGENT) → AGENT
    res(AGENT) → AGENT
    par(AGENT) → AGENT
    open(AGENT) → AGENT
    close(AGENT) → AGENT
equations
Operations on sums.

$$\begin{aligned} \mathit{sum\text{-}l}(\text{'}P + \text{'}Q) &= \text{'}P && [1] \\ \mathit{sum\text{-}l}(\text{'}P) &= \text{'}P \quad \textbf{otherwise} && [2] \end{aligned}$$

Communication

$$\mathit{com}(\text{'x ! 'y . 'P} \mid \text{'x('z) . 'Q}) = \tau \,.\, (\text{'}P \mid \text{'}Q[\text{'z} := \text{'y}]) \qquad [3]$$

$$\mathit{com}(\text{'}P) = \text{'}P \quad \textbf{otherwise} \qquad [4]$$

Restriction

$$\frac{\text{'}y \in \mathit{fn}(\alpha) \cup \mathit{bn}(\alpha) = \mathit{false}}{\mathit{res}((\nu\ \text{'}y)\ \alpha\ .\ \text{'}P) = \alpha\ .\ (\nu\ \text{'}y)\ \text{'}P} \qquad [5]$$

$$\mathit{res}(\text{'}P) = \text{'}P \quad \textbf{otherwise} \qquad [6]$$

Parallel

$$\frac{\mathit{bn}(\alpha) \cap \mathit{fn}(\text{'}Q) = \{\}}{\mathit{par}(\alpha\ .\ \text{'}P \mid \text{'}Q) = \alpha\ .\ (\text{'}P \mid \text{'}Q)} \qquad [7]$$

$$\mathit{par}(\text{'}P) = \text{'}P \quad \textbf{otherwise} \qquad [8]$$

Open scope

$$\frac{\text{'x} \neq \text{'}y}{\mathit{open}((\nu\ \text{'}y)\ \text{'x}\ !\ \text{'}y\ .\ \text{'}P) = \text{'x}\ !(\text{'}y)\ .\ \text{'}P} \qquad [9]$$

$$\mathit{open}(\text{'x}\ !(\text{'}y)\ .\ \text{'}P) = (\nu\ \text{'}y)\ \text{'x}\ !\ \text{'}y\ .\ \text{'}P \qquad [10]$$

$$\mathit{open}(\text{'}P) = \text{'}P \quad \textbf{otherwise} \qquad [11]$$

Close scope

$$\begin{aligned} \mathit{close}(\text{'x}\ !(\text{'}y)\ .\ \text{'}P \mid \text{'x}(\text{'}y)\ .\ \text{'}Q) &= \tau\ .\ (\nu\ \text{'}y)\ (\text{'}P \mid \text{'}Q) && [12] \\ \mathit{close}(\text{'}P) &= \text{'}P \quad \textbf{otherwise} && [13] \end{aligned}$$

Restricted output

$$\begin{aligned} \mathit{fn}(\text{'x}\ !(\text{'}y)) &= \{\text{'x}\} && [14] \\ \mathit{bn}(\text{'x}\ !(\text{'}y)) &= \{\text{'}y\} && [15] \end{aligned}$$

A.5 Omitted

A.5.1 Layout

The standard module defining white space.

A.5.2 Fn-bn

This module defines sets of free and bound names occurring in agents. Its definition is straightforward and not shown. It also includes a function "get-fresh", which appends sufficiently many quotes to a given name that it does not occur free in the given agent (see also [Deu94, Section 2.2.3]).

A.5.3 Subs

Module Subs defines valid substitutions over π-calculus agents, following the rules of [Bar84, Definition C.1] (again, see also [Deu94])

B Example: the GSM handover procedure

`op92.pi, Sat Apr  8 16:58:33 MET DST 1995`

The GSM handover procedure of [OP92], specified for one moving car, two base stations (an active and a passive one), and a switching center.

```
begin

  %% Transmitting the following names indicate
  %% certain atomic events.
  constants:
    data        %% announce communication of data
    ho_cmd      %% hand-over command
    ch_rel      %% channel release
    ho_fail     %% hand-over failed
    ho_acc      %% hand-over access
    ho_com      %% hand-over completed

  interface:
    in          %% channel over which full system receives input
    out         %% channel over which full system gives its output
```

The full system is split into an active and a passive part. which share a channel `act`.

```
  System() := (new act pas)( Active(act) | Passive(act,pas) )
```

The active part consists of an active Base Station communicating with a Mobile Station over a local channel.

```
  Active(act) := (new loc)( ActiveBS(act,loc) | MS(loc) )
```

The passive part consists of a Mobile Switching Center MSC communicating with a base station over a local channel.

```
  Passive(act,pas) := (new loc)(  MSC(act,pas,loc) | PassiveBS(pas,loc)  )
```

The Mobile Station, which has a local channel `ch` connected to some (active) base station. It can either receive a `data` command and then transfer data, or it can receive a `ho_cmd` command, which can either be accepted (`ho_acc`), or rejected for some reason (`ho_fail`) After doing so, it restarts.

```
MS(ch) :=
  ch : [       data(v)   => out!v . MS(ch),
         ho_cmd(newch)   => newch!ho_acc.MS(newch) + ch!ho_fail.MS(ch) ]
```

A passive base station can't do anything but wait for a hand-over access message from an arbitrary mobile station (the center has taken care that this station uses the proper channel `new-mob`). It subsequently informs the MSC that it has completed, and turns into an active BS.

```
PassiveBS( cent , new-mob ) :=
  new-mob [ho_acc] . cent!ho_com . ActiveBS( cent, new-mob )
```

An active base station repeatedly receives `data` messages from MSC on the fixed link `fix`, and forwards them to MS via the `mob` link. If it receives a hand-over command `ho_cmd` over the fixed channel, it forwards this command to the mobile station, and then has two options: (1) The fixed channels informs the base about a channel release `ch_rel`, and the base changes into a passive one; or (2) the mobile station reports a hand-over failure `ho_fail`, and the active base sends the failure information to the center.

```
ActiveBS( fix, mob ) :=
  fix : [   data(v)  => mob! data v . ActiveBS(fix,mob),
          ho_cmd(v)  => mob! ho_cmd v .
                          ( fix[ch_rel] . fix!m . PassiveBS(fix,mob) +
                            mob[ho_fail] . fix!ho_fail . ActiveBS(fix,mob) )
        ]
```

The Mobile Switching Center is connected to the passive base. It invokes the Communication Controller (for communication with the fixed network) and the Handover Controller (for storing unused radio channels).

```
MSC(act, pas, new-mob) :=
  (new l)( HandoverCon(l,new-mob) | CommunicCon(act, pas, l) )
```

The Handover Control stores one free radio channel represented by `free`, and a handover is intiated by sending this `free` link to the Communication Control. The Handover Control then waits for a new free channel to be delivered from the Communication Control, and then reverts back.

```
HandoverCon(extern, free) :=
  extern!free . extern(newch) . HandoverCon(extern,newch)
```

The Communication Control either receives data along `in` and forwards it, or it engages a handover. In that case, it receives a new channel name, informs the active base, and has two options: (1) the handover completes which is seen by the receipt of the `ho_com` by the up-til then passive base; (2) the active base informs that the handover failed.

```
  CommunicCon(act, pas, hocon) :=
    in(v) . act! data v . CommunicCon( act, pas, hocon)
    +
    hocon(new-ch) . act! ho_cmd new-ch .
      (  pas[ho_com] . act!ch_rel .
               act(old-ch) . hocon!old-ch . CommunicCon(pas, act, hocon)
         + act[ho_fail] . hocon!new-ch . CommunicCon(act, pas, hocon)
      )

end
```