Metatheory of the π-Calculus

Sam Weber*
Cornell University
samuel@cs.cornell.edu

Bard Bloom†
IBM Watson Research Laboratory
bard@watson.ibm.com

January 17, 1996

Abstract

Milner's π-calculus is a very influential process algebra in which communication channels are first-class objects. One of the basic concepts in the language is the transmission of one channel along another. This leads to immensely powerful programming techniques, which have been used for modelling things from cellular telephones to object-oriented languages.

However, the π-calculus lacks many operations, such as broadcasting a value to many processes, interrupting processes, checkpointing, and even such basics as sequencing and `while`-loops in full generality. Adding all useful operations to the π-calculus would make it unusably large and complex. We thus propose a *rule format*, called Meta-π. The π-calculus, and a vast range of other calculi treating channels as first-class data, can be expressed with Meta-π rules.

Any operations defined by Meta-π rules have the same essential theory as the π-calculus. For example, all such operations respect the appropriate notion of strong bisimulation. Furthermore, the π-calculus, and all the operations in the previous paragraph, have Meta-π equivalents. Meta-π describes the heart of the π-calculus without prejudice towards the particular communication mechanisms of the calculus, and thus gives a general framework for working with π-like calculi. Further, it can be argued that the Meta-π rule format is the most general of its kind, in the sense that any obvious extensions to the format would cause important language properties to be violated.

*Supported by NSF grants CCR-9003441 and CCR-9223183.
†Supported by NSF grants CCR-9003441 and CCR-9223183.

1 Introduction

Milner's π-calculus [14, 13] is a core language for a process algebra in which communication channels are first-class data objects. The π-calculus enjoys most of the advantages of standard process algebras like CCS [12], CSP [10], and ACP [3, 2]. In addition, it is considerably more expressive: it can describe systems in which communication patterns evolve over time, such as cellular telephones [15] and object-oriented languages [11].

Milner's π-calculus is a fixed calculus, designed to some extent for minimality. It is designed around a specific form of concurrency: asynchronous execution with synchronizing point-to-point communication. This concurrency primitive is flexible and expressive, and a wide variety of concepts may be simulated using it. However, there are situations which are awkward or impossible to model using π-calculus concurrency. A huge amount of the process-algebraic literature involves the introduction of new operations to express new concepts. For example, it is useful in many circumstances to have an *atomic broadcast* operation, sending the same message to all interested processes and guaranteeing that all of them receive it in the same event. In many other circumstances, the concept of *interrupting* is important: running one process freely, but stopping it when some external event occurs. The literature is full of operators introduced for special purposes.

Many of these situations can be simulated to some extent in the π-calculus. The simulation of interrupting can be done, for sequential processes at least, by adding an extra option at every point that the process could take an action. Other operations, including atomic broadcast, cannot be simulated at all; various degrees of approximations can be made, requiring more or less programming.

Even when operations can be simulated, simulations leave several things to be desired. One must verify the simulations, show that they are suitable and that the simulations are properly applied. The above simulation of interrupting has the flaw that it is easy to accidentally leave out the option at one point, thus invalidating the simulation. Modelling is made more complicated. A phenomenon with a simple English description as "p, q, and r communicating by broadcasting" will have very complicated π-calculus programs — with most of the complexity being in the simulations of the communication. Indeed, the processes p, q, and r may well get lost in the noise.

To overcome these problems, we provide a method by which one can tailor the π-calculus so as to define new operations, or remove existing ones. Expressing important concepts as operations rather than simulations solves both problems: validating operations is generally straightforward, and expressing concepts as operations keeps specifications clean and concise. In addition, including in one's language exactly the operations that one needs, and no more, further simplifies the specification and increases its readability.

It has been found to be useful to use *rule formats* to describe classes of languages, preserving the essential properties of specific languages, while allowing the introduction of new operations [7, 6, 8, 4]. Typically, and as in the Meta-π format, new operations are described using *structural operational semantics*, rules in which the behavior of a composite process $p \star q$ (where $\star$ is any operation) is given in terms of the behaviors of its arguments p and q.

The design philosophy of Meta-π was to insist that all Meta-π languages have certain basic properties, and that it should be trivial to check that a rule is in the proper format. For instance, whether or not a rule is in the Meta-π format is not affected by which other rules are in the language, and all the format restrictions are purely syntactic.

1.1 Rule Formats for the π-calculus

The π-calculus is extraordinarily subtle. There are several different formulations, none of them obviously generalizable. One of the key features of the π-calculus is the treatment of *channel creation* or *bound names*, a syntactic calculation of the region in which a channel may be used. The rules which ensure that bound names actually stay bound and are only used in legitimate ways are quite different in the different formulations. Getting the details right is no easy matter. The formulation of [14] requires a substantial proof that α*-conversion* is respected — and α-conversion is so trivial as to barely require mentioning in most settings with bound variables.

In this study, we present some metatheory allowing the invention of a wide variety of new operations, including broadcast, interrupt, and much more. For example, consider broadcasting on a bus. Let the action $!!c$ represent broadcasting the value c on that channel: $p \xrightarrow{!!c} p'$ indicates that p broadcasts c. Let $??c$ represent the ability to receive a value: $q \xrightarrow{??} q'$ indicates that q can receive any value on the broadcast channel. In Meta-π, this is represented by q' being an *abstraction*, a function capable of taking a value c and producing a process $q'c$. Some processes are not able to read on the broadcast channel at all times. When one cannot, it simply cannot take a $??$ action: $q \xrightarrow{??}\!\!\!\!\!\not\;\;$. An atomic broadcast combinator can be described by the rules:

$$\frac{p \xrightarrow{!!c} p', \quad q \xrightarrow{??} q'}{p \bowtie q \xrightarrow{!!c} p' \bowtie q'c} \qquad \frac{p \xrightarrow{!!c} p', \quad q \stackrel{??}{\not\longrightarrow}}{p \bowtie q \xrightarrow{!!c} p' \bowtie q} \tag{1}$$

(And symmetrically.) A broadcast system has the form $p \bowtie q \bowtie r \cdots \bowtie s$. If one process, say p, broadcasts c, then all other processes which can receive c do so; all the processes which cannot receive it, ignore it.

Interrupting can be programmed similarly. Suppose we want to allow p to run freely, except that whenever the signal k is sent to it, it will stop. This can be programmed with rules of the form:

$$\frac{p \xrightarrow{a} p', a \neq k}{\mathsf{inter}(p,k) \xrightarrow{a} \mathsf{inter}(p',k)} \qquad \frac{}{\mathsf{inter}(p,k) \xrightarrow{k} \mathbf{0}} \tag{2}$$

We show that the Meta-π format confers the essential advantages of the other key rule formats, as well as the functionality of the π-calculus. Some properties are basic but essential. Any language defined in Meta-π has a unique sensible operational semantics. Processes in any Meta-π language have a syntactically-computable set of free channels, and are guaranteed to have no access to other channels unless they are given access to them. Others are more subtle. In particular, there is a suitable notion of process equivalence — a kind of strong bisimulation, which generalizes the "strong congruence" of the π-calculus. We show that our example operations of broadcasting and interrupting can be programmed conveniently in Meta-π rules, though they cannot be simulated up to bisimulation in the π-calculus.

1.2 Summary of the π-calculus

The Meta-π rule format resembles the *polyadic π-calculus* of [13, 16], and is based on many similar intuitions. We sketch the essential features of π calculi; for didactic clarity we omit a great many details, and merge notation and concepts from several π-calculus papers. We make no formal claims about the mishmosh of π-calculi presented here; it is purely for exposition. We *do* give formal relations between our system and one from [13].

The basic components of the π-calculus is a set of *Names*, ranged over by x, y, z. Names serve two purposes: (1) they are the names of *communication channels*, and one may send or receive along them, and (2) they are variables ranging over communication channels. Here are the basic constructs for manipulating names:

$\bar{x}y.P$ transmits the name y along the channel called x.

$\lambda x.P$**:** is a *function* or *abstraction* (not a process at all), which, when applied to a channel name y, produces the process $P[x := y]$. This binds x in P. Abstractions are used to accept input, using the following construct.

$x\lambda y.P$ which reads a name from channel x, and then applies the body $\lambda y.P$ to that input name. This is written $x(y).P$ in many formulations of the π-calculus.

$\nu x.P$**:** Behaves like P, except that the name x is made *local* in it. It may also be thought of as a construct which *creates* a new name, which is called x in P.

For example, consider $\bar{x}y.\mathbf{0}|x\lambda u.\bar{u}v.\mathbf{0}|\bar{x}z.\mathbf{0}$, where the process $\mathbf{0}$ is stopped, and $|$ is parallel composition. Two communications are possible; either send on $\bar{x}$ could happen, resulting in $\mathbf{0}|\bar{y}v.\mathbf{0}|\bar{x}z.\mathbf{0}$ or $\bar{x}y|\bar{z}v|\mathbf{0}$.

One of characteristic features of the π calculi is the flexible behavior of scoping. Consider $P = \nu x.\bar{y}x.\bar{x}a.\mathbf{0}$, a process which generates a new channel x and then transmits it on some (externally visible) channel y. A process $Q = y\lambda z.z\lambda w.Q'$ can read this newly generated channel. This is recorded in the operational semantics by *expanding* the scope of x:

$$P|Q = (\nu x.\bar{y}x.\bar{x}a.\mathbf{0})|(y\lambda z.z\lambda w.Q') \rightarrow \nu x.(\bar{x}a.\mathbf{0}|x\lambda w.Q')$$

The usual issues of renaming bound variables arise:

$$(\nu x.\bar{y}x.x\lambda w.\mathbf{0})|(y\lambda z.\bar{z}x.\mathbf{0}) \rightarrow \nu v.(v\lambda w.\mathbf{0}|\bar{v}x.\mathbf{0})$$

The bound name x gets renamed to v, as it conflicts with the free name x. The right process must not be $\bar{x}x.\mathbf{0}$, as it does not send *any* name along itself.

To simplify the reduction relation, Milner in [13] introduces *structural equivalence*, which we denote $\equiv_M$. It includes facts about the behavior of free and bound variables, and facts about the operations of the language.

1. $p \equiv_M q$ if p and q differ only by a change of bound variables.

2. $\nu x.\mathbf{0} \equiv_M \mathbf{0}$ and $\nu x.\nu y.p \equiv_M \nu y.\nu x.p$.

3. If x is not a free name of p, then $\nu x.(p|q) \equiv_M (\nu x.p)|q$.

4. The operations of $+$ (choice) and $|$ are commutative and associative, and $\mathbf{0}$ is an identity for both.

5. $!p = p|!p$, where $!p$ is a *replicator* operation which turns p into a reentrant server, running arbitrarily many copies of it in parallel.

Note that 1 and 2 are general properties of free and bound variables, and 3-5 are special-purpose rules explaining some properties of some operations.

Now, Milner defines a transition relation. In [13], the transition relation is unlabeled, $P \rightarrow P'$; this variant of the π-calculus is intended for closed systems. We write this as $\xrightarrow{\tau}$, for consistency with most other systems. Much of the computational behavior is covered by $\equiv_M$. The remaining behavior is that $(A + x\lambda y.P)|(\bar{x}z.Q + B) \xrightarrow{\tau} (P[y := z])|Q$, the term $P|Q$ can execute either P or Q separately, $\nu x.P$ can execute if P can, and that $P \xrightarrow{\tau} Q$ if there exists P' and Q' such that $P \equiv_M P', Q' \equiv_M Q$ and $P' \xrightarrow{\tau} Q'$.

The calculus of [14] uses $\xrightarrow{\tau}$, and introduces three other kinds of transitions. First is $\xrightarrow{\bar{x}y}$, indicating a send of the name y on the channel called x. This has the basic rule $\bar{x}y.P \xrightarrow{\bar{x}y} P$.

The second notion is used for input. As from a suggestion of Parrow mentioned in [13], we write this $x.\lambda y.P \xrightarrow{x} \lambda y.P$.

The third form is somewhat more perplexing, and indeed makes the technical details of [14] rather tricky. Consider a term of the form $\nu x.\bar{y}x.P$, which creates a new name x, and transmits it on y. The action $\xrightarrow{\bar{y}x}$ is not appropriate, as it has lost track of the fact that this x is different from all other x's in existence. [14] uses the notation $\xrightarrow{\bar{y}(x)}$ for this transmission of a *bound* x. We prefer the notation $\xrightarrow{\nu x.\bar{y}x}$, which extends better to Meta-π.

$$\frac{p \xrightarrow{\bar{y}x} p', y \neq x}{\nu x.p \xrightarrow{\nu x.\bar{y}x} p'} \qquad \frac{p \xrightarrow{\nu x.\bar{y}x} p', \quad q \xrightarrow{y} \lambda z.q'}{p|q \xrightarrow{\tau} \nu x.(p'|(q'[z := x]))} \tag{3}$$

The left-hand rule notes when names that are being transmitted require scope extrusion. The right-hand rule marks that the message has reached its destination, and introduces a scope operator covering the sender p' (which had x before and may still have it), and the receiver q' (which has just been given it).

There are many studies showing that the π-calculi are sensible. There are notions of *bisimulation*, based on the CCS concept, which provide adequate semantics for the calculi. There are plenty of examples showing that the calculus can sensibly model a variety of situations. We will present our own version of bisimulation in Section 3.

2 Fundamentals of Meta-π

In this section, we present the Meta-π rule format.

1. We describe the (fairly rudimentary) system of types and terms.
2. We define a notion of *local equivalence*, $\equiv$, which captures (a) the delivery of messages to their recipients, and (b) the scoping of names.
3. We describe the Meta-π rule format.
4. We define a notion of bisimulation — the Meta-π equivalent of the strong equivalence of [13] — and show that all Meta-π operations respect it.

We use the π-calculus extended with broadcast and interrupt as a running example.

2.1 Types and Terms

All Meta-π calculi have the same type system, with the following types:

C: The type of channels and channel names.

P: The type of processes.

$\mathbf{A}_n$: The type of n-ary abstractions, for $n \geq 0$; that is, functions which accept n arguments of type **C** and return a **P**. $\mathbf{A}_0$ is identified with **P**.

We use ξ to range over types.

A Meta-π language defines a number of operations. We divide these operations into two kinds: *functions* and *replicators*. This division is prompted by the observation that all of the operations in the π-calculus except for replication, and most of the operations found in the literature share the property that if f is an operation , $f(x_1, \ldots, \nu\{a\}.x_i, \ldots, x_n)$ is equivalent to $\nu\{a\}.f(x_1, \ldots, x_i, \ldots, x_n)$ as long as none of the other x_j have a free. In other words, one can move variable binding in and out of such functions as long as it doesn't result in capturing any unbound channel names. This is a very useful property, since it allows great freedom in re-arranging the bindings in a term. The operations which have this property we will call functions – intuitively, they do not make copies of their arguments either directly or indirectly. The other operations are replicators, which can copy their arguments.

The *signature* of a Meta-π language comprises disjoint sets of *Channel Constants*, *Function Constants*, *Replicator Constants*, and *Action Constructors*. Function and replicator constants have type signatures of the form $\xi_{1f} \times \cdots \times \xi_{nf} \to \mathbf{P}$. Action constructors have nonnegative numbers as arities.

The terms generated by the signature are the obvious ones. Note that we use a finite set of actions rather than a single action in the ν construct; this allows us to bind two channel names without being forced to state which one is bound first, which corresponds to our intuition that order of binding shouldn't matter. Also notice that there are non-trivial terms of type **C**. For instance, $\nu\{a\}.a$ is a term of type channel which represents a new channel unlike any others.

$$\frac{}{X^\xi : \xi} \qquad \frac{}{a : \mathbf{C}} \qquad \frac{t_i : \xi_{if}, f \in \mathsf{FunConst} \cup \mathsf{ReplConst}}{f(t_1, \ldots, t_{n_f}) : \mathbf{P}}$$

$$\frac{t : \mathbf{P}}{\lambda X_1^{\mathbf{C}}, \ldots X_n^{\mathbf{C}}.t : \mathbf{A}_n} \qquad \frac{t : \xi, s \subset \mathsf{ChanConst}, |s| < \infty}{\nu s.t : \xi} \qquad \frac{t : \mathbf{A}_n, u_1, u_2, \ldots, u_n : \mathbf{C}}{t u_1 u_2 \ldots u_n : \mathbf{P}}$$

We define the *free variables* $\mathsf{fv}(t)$ and *free channels* $\mathsf{fc}(t)$ of a term t in the obvious way. λ binds variables; ν binds all the actions in a set. A *process* is a term of type **P** with no free variables. Processes are the entities of our prime concern: they are able to perform actions.

2.2 Semantics

2.2.1 α-Conversion and Equality

We view α-conversion (equality up to changes of name of bound variables) as an intrinsic property of any Meta-π language. Predictably, we define $\nu(\{a\} \cup s).t =_\alpha \nu(\{b\} \cup s).(t[a := b])$, where b is a fresh variable. However, we also let $=_\alpha$ eliminate unused channel names:

$$\nu(\{a\} \cup s).t =_\alpha \nu s.t \text{ if } a \notin \mathsf{fc}(t) \tag{4}$$

In our theory, we use $=_\alpha$, which we call trivial equality, as our notion of equality. We never want to be able to distinguish terms that are trivially equal and the theory is simplified by using an equality that is more suitable than lexical equality.

2.2.2 Local Equivalence

We define a notion of *local equivalence* $p \equiv q$. Local equivalence plays a similar role to Milner's structural equivalence. However, structural equivalence mixes universal properties of channel names (such as elimination of unused channel names, and distribution of ν over suitable compositions) with specific properties of the operations of the calculus (associativity and commutativity of $+$ and $|$). In the more general setting of, Meta-π, the specific properties are not available.

Local equivalence only concerns universal matters of how binding operations behave. The familiar β-reduction holds universally. The distribution of binding over functions, as discussed above, is represented in the local equivalence. We use local equivalence to transform composite processes into forms in which subprocesses can interact.

1. $t_1 \equiv t_2$ if $t_1 =_\alpha t_2$.
2. $(\lambda \vec{X}.q)\vec{u} \equiv q[\vec{X} := \vec{u}]$. ($\beta$-reduction)[1]
3. $\nu\{a\}.(t_0 t_1 t_2 \ldots t_n) \equiv (\nu\{a\}.t_0)(\nu\{a\}.t_1)\ldots(\nu\{a\}.t_n)$ if $a \in \mathsf{fc}(t_i)$ for at most one i
4. $(\nu s_1.(\nu s_2.t)) \equiv \nu(s_1 \cup s_2).t$
5. $\nu\{a\}.f(\vec{t}) \equiv f(\nu\{a\}.t_1, \ldots, \nu\{a\}.t_n)$ if $f \in \mathsf{FunConst}$ and $a \in \mathsf{fc}(t_i)$ for at most one i.
6. $\nu s.\lambda X.p \equiv \lambda X.\nu s.p$
7. For any context $C[\cdot]$ of the correct type, $t \equiv t' \Rightarrow C[t] \equiv C[t']$.

The π-calculus equivalents to all of these are valid up to strong congruence in the π-calculus, though not all are part of structural equivalence. Also, special-case facts, such as the fact that $p|q$ and $q|p$ have identical behavior, will be theorems rather than part of the definition of the calculus.

2.2.3 Actions

Actions in the π-calculus denote events occurring: messages being sent along channels. Messages, and hence actions, may contain channel names. Thus, when we generalize the concept, we parameterize our actions by a sequence of channel names.

We assume a set of action constructors, symbols $\sharp$ with associated arities. Actions have the form

$$\alpha = \langle \sharp a_1 a_2 \ldots a_n \rangle \tag{5}$$

where $\sharp$ is an n-ary action constructor.

For example, CCS-style actions are 0-ary constructors. The π-calculus action $\bar{x}y$ carries two channel names; we represent it with a binary action constructor $-$, as the action $\langle -xy \rangle$. The action representing receiving on channel x will be $\langle ?x \rangle$, and the silent action will be $\langle \tau \rangle$.

[1] For technical reasons, each u_i must either be a channel constant or a variable. This isn't a restriction, because otherwise rule 3 can be used to simplify it.

2.2.4 Transitions and Built-In Rules

As with many other process calculi, the main judgement of interest is a process p performing an action a and thereafter behaving like some other entity p': written $p \xrightarrow{a} p'$. p must be a process; but it is convenient to allow p' to be a term of any type. Transitions to processes are interpreted as usual in process algebra. Transitions to non-processes are used to code input, output, and other interesting things. The type of p' is effectively part of the transition; that is, transitions are more properly written $p \xrightarrow{a,\xi} p'$ where p' has type ξ.

This allows us to specify processes which accept input. Following [17, 9, 14, 5], a process q will signal that it is capable of accepting one channel value as input by taking a transition to $q' = \lambda x.r'$ of type $\mathbf{A}_1$. When the actual input value v is provided by some other process in the environment, q' will be given v as an argument.

We have two rules, showing how the built-in constructs ν and $\equiv$ interact with transitions. The ν-rule says that $\nu s.P$ can do what P can do, except that the names hidden by ν are not allowed to escape from it. The $\equiv$-rule says that processes which are equivalent have the same transitions.

$$\frac{P \xrightarrow{\alpha} P', \quad s \cap \mathsf{fc}(\alpha) = \varnothing}{\nu s.P \xrightarrow{\alpha} \nu s.P'} \qquad \frac{P \equiv Q \xrightarrow{\alpha} Q' \equiv P'}{P \xrightarrow{\alpha} P'}$$

2.2.5 Transitions and Rules

The precise format for the rules for operations is somewhat subtle. Rules for functions and replicators are as follows:

$$\frac{hyp_1, \ldots, hyp_n}{f(\vec{X}) \xrightarrow{\langle \sharp \vec{A} \rangle} t}$$

where:

- Each hypothesis has one of the following forms:
  1. $X_i \xrightarrow{\langle \sharp \vec{Z} \rangle} Y_i$; testing to see if a process can take a transition with constructor $\sharp$. Y_i and $\vec{Z}$ will be substituted with the appropriate values from the transition. The variables Y and Z must be distinct in all hypotheses.
  2. $X_j \stackrel{\sharp}{\not\longrightarrow}$; checking to see that X_j cannot take a transition with constructor $\sharp$.
  3. $C_k = D_k$ or $C_k \neq D_k$, where C_k and D_k are channel variables. This tests to see if two channels are equal or unequal.
- $\vec{A} \cup \vec{C} \cup \vec{D} \subseteq \vec{X} \cup \bigcup \vec{Z}$. That is, every action appearing in an equality test or in the transition of $f(\vec{X})$ must either appear as an argument to f, or from some transition of an argument.
- The variables of t can only be X's, Y's, or Z's, and $\mathsf{fc}(t) = \varnothing$.

Functions have a few other technical requirements. Essentially, we must avoid duplicating their arguments, either directly (by repeating variables) or indirectly (by replicators). Duplication would interfere with the ability of ν's to move over functions. (1) t must be linear in $\vec{X}$ and $\vec{Y}$: using each variable once, and not using X_i at all if there is an $X_i \longrightarrow Y_i$ hypothesis. (2) There may not be any negative antecedents. (3) No X_i or Y_i variable can appear as an argument to a replicator inside of t.

Examples: Informally, the prefixing operators have the same kind of rule that they do in CCS: *e.g.*, $\alpha X \xrightarrow{\alpha} X$. Formally, all desired prefixing operations have the form $\mathsf{prefix}_\sharp$, an operation of type $\mathbf{C}^n \times \xi \to \mathbf{P}$ for some ξ. These operations have the single rule:

$$\frac{}{\mathsf{prefix}_\sharp(X_1, \ldots, X_n, X_{n+1}) \xrightarrow{\langle \sharp X_1 \ldots X_n \rangle} X_{n+1}}$$

We use the informal notation. The π-calculus prefixing operations are $\langle -xy \rangle.P$ for sending, $\langle ?x \rangle.(\lambda Y.p)$ for recieving a value and binding it to Y, and $\langle \tau \rangle$ for hidden action.

The rules for π-calculus-style concurrency in Meta-π will include

$$\frac{p \xrightarrow{\alpha, \mathbf{P}} p'}{p|q \xrightarrow{\alpha} p'|q} \qquad \frac{p \xrightarrow{\langle -xv \rangle, \mathbf{P}} p', \quad q \xrightarrow{\langle ?x' \rangle, \mathbf{A}_1} q', \quad x = x'}{p|q \xrightarrow{\langle \tau \rangle, \mathbf{P}} p'|(q'v)} \tag{6}$$

In the left rule we use α to range over actions; the rule allows p to compute asynchronously. The right rule handles communication. One process tries to send value v on some channel x; the other signals its willingness to accept input on channel x'. If the two happen to agree on the channel they are communicating on — that is, if $x = x'$ — then the communication can happen. The sender p' simply proceeds, knowing that the communication happened. The receiver q' proceeds — but the value v that was sent to it is now available to it.

The interrupt operation can be programmed conveniently. Recall that $\mathsf{inter}(X_1, X_2)$ lets process X_1 run as it wishes, except that will be interrupted and cancelled by a signal sent to it on X_2. Its rules include

$$\frac{X_1 \xrightarrow{\langle ?Z \rangle} Y, \quad Z \neq X_2}{\mathsf{inter}(X_1, X_2) \xrightarrow{\langle ?Z \rangle} \mathsf{inter}(Y, X_2)} \qquad \frac{}{\mathsf{inter}(X_1, X_2) \xrightarrow{\langle ?X_2 \rangle} \mathbf{0}} \tag{7}$$

The left rule lets $\mathsf{inter}(X_1, X_2)$ receive along all channels other than X_2; other rules (not listed here) let it perform all its other actions as well. The right rule intercepts reading on channel X_2, killing X_1 when a signal comes in on X_2.

A Meta-π system comprises finite collections of operations, action constructors, and rules. The main technical results of this study show that Meta-π systems work properly. The first such result says that they work at all:

Theorem 2.1 *Given any Meta-π system, the transition relation $\longrightarrow$ has a unique and type-correct definition.*

2.2.6 Example of Computation

Let p equal $\nu\{y\}.\langle -xy \rangle.\mathbf{0}$ and q be $\langle ?x \rangle \lambda Y.\langle -zY \rangle.r$. p is a process which generates a new channel y, and transmits it on a fixed channel x. Note that $p \equiv \langle -x(\nu\{y\}.y) \rangle.\mathbf{0}$, a form in which it is clear that p is creating a channel and sending it. q reads a single value from channel x, re-transmits it along a fixed channel z, and proceeds with some further computation r.

Here is the canonical computation in which p and q coordinate:

$$p|q \equiv \nu\{y\}.(\langle -xy \rangle.\mathbf{0}|\langle ?x \rangle \lambda Y.\langle -zY \rangle.r) \xrightarrow{\langle \tau \rangle} \nu\{y\}.(\mathbf{0}|(\lambda Y.\langle -zY \rangle.r)y) \equiv \nu\{y\}.(\mathbf{0}|(\langle -zy \rangle.r))$$

The first $\equiv$ is a use of the scope-extension laws; since q does not mention y, the scope of y can be extended to cover q. Within that scope, y may be sent over channel x. After the communication has happened, q's descendant is an abstraction capable of accepting a value; it is given the value y

that was transmitted. A step of β-reduction, which is part of $\equiv$, can distribute the received value y to the places in the body of q where it was used.

So far, we have considered processes in isolation. The essence of the π-calculus, which we will keep in Meta-π, is that processes may extend channel names into their surrounding context. For example, consider a process $p_1 = \nu\{y\}.(\langle -xy\rangle.\mathbf{0})$, which creates a new channel y and transmits it on some known channel x.

In isolation, p_1 cannot do anything. p_1's body $\langle -xy\rangle.\mathbf{0}$ can take a $\langle -xy\rangle$-transition to $\mathbf{0}$. However, the presence of the bound name y in the action $\langle -xy\rangle$ prevents that signal from escaping the ν. If we tried to consider it in isolation, p_1 would be stopped and uninteresting.

In contexts, though, p_1 is not stopped. p_1 can send its private channel y to another process. Let $q_1 \xrightarrow{\langle ?x\rangle} \lambda z.q'$ be a process capable of reading from channel x. Then,

$$p_1|q_1 \equiv \nu\{y\}.((\langle -xy\rangle.\mathbf{0})|q_1) \xrightarrow{\langle\tau\rangle} \nu\{y\}.(\mathbf{0}|q'y) \tag{8}$$

which is to say, q_1's descendant now has access to the hidden name y, whose scope has been expanded.

This pattern of a scope expansion followed by the communication of a once-hidden action is paradigmatic. We define *extended transitions* to capture the concept:

$$p \xrightarrow{\nu s.\alpha} p' \quad \text{iff} \quad p \equiv \nu s.q \text{ for some } s, q \text{ such that } s \subseteq \mathsf{fc}(\alpha), q \xrightarrow{\alpha} p'$$

For example, $p_1 \xrightarrow{\nu\{x\}.\langle -xy\rangle} \mathbf{0}$.

3 Bisimulation in Meta-π

We define a notion of bisimulation on Meta-π languages. For the π-calculus, the notion called bisimulation is not an adequate semantics; a finer version called "strong congruence" is required. This is due to the π-calculus identification of channel names and channel variables. One cannot trust distinct constants to remain distinct; they might be treated as variables, and both replaced by the same value. We separate the concepts of channel names and channel variables, thus guaranteeing that distinct constants will actually stay distinct. We believe that the two concepts ought to be distinct, though reasonable scholars might disagree.

So, in our setting, bisimulation is an adequate semantics. The underlying technical realities are actually the same. Our definition of bisimulation on abstractions — saying that two functions agree if they agree on all arguments — is tantamount to the strong congruence construction.

Definition 3.1 *If $\smile$ is a binary relation between terms of the same type, then we define*

$$\nu s.\langle\beta, q\rangle \;\overset{\smile}{=}\; \nu s'.\langle\beta', q'\rangle$$

to hold if $\nu s.\langle\beta, q\rangle \equiv \nu s_*.\langle\beta_*, q_*\rangle$, $\nu s'.\langle\beta', q'\rangle \equiv \nu s_*.\langle\beta_*, q'_*\rangle$ (extending $\equiv$ to tuples in the obvious way) and $q_* \smile q'_*$.

The definition of bisimulation is familiar. Channels are simply scalars; they are bisimilar if they are equal. For processes, we use the standard notion as from CCS, except that we may rename bound channels in the action. Functions are bisimilar if they are bisimilar on all arguments.

Definition 3.2 *A relation $\smile$ on terms with at most channel variables free is a* bisimulation relation *if*

1. *Whenever* $p \smile p' : \mathbf{C}$ *are closed, then* $p =_\alpha p'$.

2. *Whenever* $p \smile p' : \mathbf{P}$ *are closed, and* $p \xrightarrow{\nu s.\beta} q$, *then there is a* q' *such that* $p' \xrightarrow{\nu s'.\beta'} q'$ *amd* $\nu s.\langle\beta, q\rangle \overset{\smile}{=} \nu s'.\langle\beta', q'\rangle$.

3. *Whenver* $p \smile p' : \mathbf{A}_n$ *are closed, then for all vectors* $\vec{a}$ *of length* n, $p\vec{a} \smile p\vec{a}$.

4. *Whenever* $p \smile p'$ *and* p *or* p' *is open, we have* $\theta(p) \smile \theta(p')$ *for all instantiations* θ.

Terms p *and* p' *are* strongly bisimular, $p \Leftrightarrow p'$, *if there is a bisimulation relation* $\smile$ *with* $p \smile p'$.

The main technical result of this study is the following theorem, saying that bisimulation is a usable notion of process equivalence.

Theorem 3.3 *Any Meta-*π *langage respects bisimulation. Formally, if* $p_i \Leftrightarrow p'_i$, *then* $C[\vec{p}] \Leftrightarrow C[\vec{p'}]$ *for any context* C.

The proof of this theorem is quite complex – since Meta-π subsumes the π-calculus the proof by necessity has to be more involved than the π-calculus equivalent. We sketch the major points of the proof, showing only the concepts which are of general utility.

One of the properties of any Meta-π language is that the behaviour of any term p is identical to that of any term p' which is locally equivalent to it. However, the set of terms locally equivalent to p is infinite, which makes the task of determining the complete behaviour of p appear daunting at first glance. However, we show that situation is considerably better than this.

We define the notion of the *minimal free channels* of a term p, $\mathsf{mfc}(p)$, to be the channels that appear free in all terms that are locally equivalent to p. Formally, $\mathsf{mfc}(p) = \bigcap_{p' \equiv p} \mathsf{fc}(p')$. We can then show the following lemma:

Lemma 3.4 *If* $p \xrightarrow{\alpha} q$, *then* $\mathsf{fc}(\alpha) \subseteq \mathsf{mfc}(p)$ *and* $\mathsf{mfc}(q) \subseteq \mathsf{mfc}(p)$

This implies that a term cannot communicate using any channels other than the ones that it has free, or take a transition and then be able to do such communication.

We then define a *normal form* for a term p, written $\mathsf{nf}(p)$. The normal form is defined purely syntatically, and $p \equiv \mathsf{nf}(p)$. Informally, the normal form of a term is derived by performing any applications that are possible, moving any variable bindings to be as global as possible (including moving bindings past functions and abstractions), and eliminating all extraneous bindings. More formally, we define a transition relation that computes the normal form of a term, and show that this transition relation is Church-Rosser.

The normal form of a term is unique up to our notion of equality, and $\mathsf{fc}(\mathsf{nf}(p)) = \mathsf{mfc}(p)$. We show that in a strong sense the entire behaviour of a term can be computed from its normal form without considering any other locally equivalent terms. More precisely, we define another transition relation $\xrightarrow{\cdot}_n$ which is the same as the usual one except that the use of local equivalence is disallowed. We then prove the following lemma, which shows that we can still get the entire behaviour of the term using the restricted transition and the normal form.

Lemma 3.5

$$p \xrightarrow{\alpha} q \quad \textit{iff} \quad \mathsf{nf}(p) \xrightarrow{\alpha}_n q' \tag{9}$$

where $q \equiv q'$.

This lemma allows us to consider only the behaviour of terms that are in normal form. The remainder of proof consists mainly of considering the effects of every possible rule that could be in a Meta-π language, and verifying that the necessary properties always hold.

4 The π-calculus and Beyond

It is straightforward to define a Meta-π calculus which imitates Milner's π-calculus under late semantics, and a translation $P \mapsto P^*$ translating between them. Each of the operations of the π-calculus translates straightforwardly to a Meta-π operation. ν, of course, translates into ν. The only oddity is that the π-calculus construct $P = x(y).p$, combines reading a channel name and binding it to y. Meta-π separates these concepts, so P is translated as $\langle ?x \rangle.\lambda y.p^*[y := Y]$. Note that we replace a bound channel name by a bound variable ranging over channel names.

We have captured the π-calculus, in the sense of the following theorem. We have to cheat slightly on which π-calculus we match. The ! operator replicates p instantly; our replicators have to take at least one step to do it. So, we take one of the alternate formulations of the calculus, from [14], in which infinite behavior is provided by *recursion equations* $X = T$, recursive definitions in the style of many other process algebras. In any event, Milner argues that the additional step doesn't matter.

Theorem 4.1 *Let P be any term of the π-calculus, using recursion equations rather than the replicator !. Then the extended transitions of P match those of P^*; that is, $P \xrightarrow{\gamma} Q$ iff $P^* \xrightarrow{\gamma} Q^*$.*

Furthermore, we may construct new operations which cannot be expressed in the π-calculus. In particular, neither $\bowtie$ nor $\mathsf{inter}(\cdot,\cdot)$ can be expressed in the π-calculus. That is, there are no π-calculus contexts $BC[\cdot,\cdot]$ or $IN[\cdot,\cdot]$ such that $BC[p,q] \leftrightarrows p \bowtie q$ or $IN[p,k] \leftrightarrows \mathsf{inter}(p,k)$ for all p, q, k.

Meta-π thus provides a generalization of the theory of the π-calculus. Any Meta-π calculus has the essential basic technical properties of the π-calculus, from computational properties such as mobile scopes and ability to compute with channels to theorems such as that a suitable bisimulation notion is a congruence.

References

[1] L. Aceto, B. Bloom, and F. Vaandrager. Turning SOS rules into equations. *Information and Computation*, 111(1):1–52, May 1994. (Special Issue for LICS '92).

[2] J. C. M. Baeten. *Applications of Process Algebra*. Cambridge Tracts in Theoretical Computer Science 17. Cambridge University Press, 1990.

[3] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[4] B. Bloom. Structural operational semantics considered as a specification language. In *Proceedings of POPL 1995*, pages 107–117.

[5] B. Bloom. CHOCOLATE: Calculi of Higher Order COmmunication and LAmbda TErms. In *Proceedings of POPL 1994*, pages 339–349. IEEE Press, 1994.

[6] B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced (preliminary report). In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 229–239, 1988. Also appears as MIT Technical Memo MIT/LCS/TM-345.

[7] R. de Simone. Higher-level synchronising devices in MEIJE-SCCS. *Theoretical Computer Sci.*, 37(3):245–267, 1985.

[8] J. F. Groote and F. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100(2):202–260, October 1992.

[9] M. Hennessy. A fully abstract denotational model for higher-order processes (extended abstract). In *Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 397–408. IEEE Computer Society Press, 1993. (Full version in Sussex TR 6/92).

[10] C. A. R. Hoare. *Communicating Sequential Processes.* Series in Computer Science. Prentice-Hall, 1985.

[11] C. B. Jones. Process-algebraic foundations for an object-based design notation. Technical Report UMCS-93-10-1, University of Manchester, 1993.

[12] R. Milner. *Communication and Concurrency.* Prentice Hall International Series in Computer Science. Prentice Hall, New York, 1989.

[13] R. Milner. The polyadic π-calculus: A tutorial. In *Proc. International Summer School on Logic an dAlgebra of Specification.* Marktoberdorf, 1991.

[14] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes I and II. *Information and Computation*, 100:1–40 and 41–77, 1992.

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

[16] D. Sangiorgi. From π-calculus to Higher-Order π-calculus — and back. In M.-C.Gaudel and J.-P.Jouannaud, editors, *TAPSOFT '93: Theory and Practice of Software Development.* Springer-Verlag, April 1993.

[17] B. Thomsen. A calculus of higher order communicating systems. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 142–154, 1989.