The Polyadic π-Calculus: a Tutorial

Robin Milner

Laboratory for Foundations of Computer Science,
Computer Science Department, University of Edinburgh,
The King's Buildings, Edinburgh EH9 3JZ, UK

October 1991

Abstract

The π-calculus is a model of concurrent computation based upon the notion of *naming*. It is first presented in its simplest and original form, with the help of several illustrative applications. Then it is generalized from *monadic* to *polyadic* form. Semantics is done in terms of both a reduction system and a version of labelled transitions called *commitment*; the known algebraic axiomatization of strong bisimilarity is given in the new setting, and so also is a characterization in modal logic. Some theorems about the *replication* operator are proved.

Justification for the polyadic form is provided by the concepts of *sort* and *sorting* which it supports. Several illustrations of different sortings are given. One example is the presentation of data structures as processes which respect a particular sorting; another is the sorting for a known translation of the λ-calculus into π-calculus. For this translation, the equational validity of β-conversion is proved with the help of replication theorems. The paper ends with an extension of the π-calculus to ω-order processes, and a brief account of the demonstration by Davide Sangiorgi that higher-order processes may be faithfully encoded at first-order. This extends and strengthens the original result of this kind given by Bent Thomsen for second-order processes.

This work was done with the support of a Senior Fellowship from the Science and Engineering Research Council, UK.

1 Introduction

The π-calculus is a way of describing and analysing systems consisting of agents which interact among each other, and whose configuration or neighbourhood is continually changing. Since its first presentation [19] it has developed, and continues to do so; but the development has a main stream. In this tutorial paper I give an introduction to the central ideas of the calculus, which can be read by people who have never seen it before; I also show some of the current developments which seem most important – not all of which have been reported elsewhere.

Any model of the world, or of computation (which is part of the world), makes some ontological commitment; I mean this in the loose sense of a commitment as to which phenomena it will try to capture, and which mental constructions are seen to fit these phenomena best. This is obvious for the "denotational" models of computing; for example, the set-theoretic notion of *function* is chosen as the essence or abstract content of the deterministic sequential process by which a result is computed from arguments. But mathematical operations – adding, taking square-roots – existed long before set theory; and it seems that Church in creating the λ-calculus had "algorithm" more in mind than "function" in the abstract sense of the word.

Nevertheless, the λ-calculus makes *some* ontological commitment about computation. It emphasizes the view of computation as taking arguments and yielding results. By contrast, it gives no direct representation of a heterarchical family of agents, each with its changing state and an identity which persists from one computation to another. One may say that the λ-calculus owes its very success to its quite special focus upon argument-result computations.

Concurrent computation, and in particular the power of concurrently active agents to influence each other's activity on the fly, cannot be forced into the "function" mould (set-theoretic or not) without severe distortion. Of course concurrent agents can be assumed (or constrained) to interact in all sorts of different ways. One way would be to treat each other precisely as "function computers"; such an agent's interaction with its environment would consist of receiving arguments and giving results and expecting its sub-agents, computing auxiliary functions, to behave in a similar way. Thus functional computation is a special case of concurrent computation, and we should expect to find the λ-calculus exactly represented within a general enough model of concurrency.

In looking for basic notions for a model of concurrency it is therefore probably wrong to extrapolate from λ-calculus, except to follow its example in seeking something small and powerful. (Here is an analogy: Music is an art form, but it would be wrong to look for an aesthetic theory to cover all art forms by extrapolation from musical theory.) So where else do we look? From one point of view, there is an embarrassingly wide range of idea-sources to choose from; for concurrent computation in the broadest sense is about any co-operative activity among independent agents – even human organizations as well as distributed computing systems. One may even hope that a model of concurrency may attain a breadth of application comparable to physics; Petri expressed such hopes in his seminal work on concurrency [25], and was guided by this analogy.

Because the field is indeed so large, we may doubt whether a single *unified* theory of concurrency is possible; or, even if possible, whether it is good research strategy to seek it so early. Another more modest strategy is to seize upon some single notion which seems to be pervasive, make it the focus of a model, and then submit that model to various tests: Is its intrinsic theory tractable and appealing? Does it apply to enough real situations to be useful in building systems, or in understanding those in existence?

This strategy, at least with a little hindsight, is what led to the π-calculus. The pervasive notion we seize upon is *naming*. One reason for doing so is that naming strongly presupposes *independence*; one naturally assumes that the *namer* and the *named* are co-existing (concurrent) entities. Another reason is that the act of using a name, or address, is inextricably confused with the act of communication. Indeed, thinking about names seems to bring into focus many aspects of computing: problems, if not solutions. If naming is involved in communicating, and is also (as all would agree) involved in locating and modifying data, then we look for a way of treating data-access and communication as the same thing; this leads to viewing data as a special kind of process, and we shall see that this treatment of data arises naturally in the π-calculus.

Another topic which we can hope to understand better through naming is *object-oriented programming*; one of the cornerstones of this topic (which is still treated mostly informally) is the way in which objects provide access to one another by naming. In [17] I used the term *object paradigm* to describe models such as the π-calculus in which agents (objects) are assumed to persist and retain independent identity. David Walker [28] has had initial success in giving formal semantics to simple object-oriented languages in the π-calculus. A challenging problem is to reconcile the assumption, quite common in the world of object-oriented programming, that each object should possess a unique name with the view expressed below (Chapter 1) that naming of *channels*, but not of *agents*, should be primitive in the π-calculus.

By focussing upon naming, we should not give the impression that we expect every aspect of concurrency to be thereby explained. Other focal notions are likely to yield a different and complementary view. Yet naming has a strong attraction (at least for me); it is a notion distilled directly from computing practice. It remains to be seen which intuitions for understanding concurrency will arise from practice in this way, and which will arise directly from logic – which in turn is a distillation of a kind of computational experience, namely inference. Both sources should be heeded. An example of a logical intuition for concurrency is the light cast upon *resource use* by Girard's linear logic [9]. I believe it quite reasonable to view these two sources of intuition as ultimately the same source; then the understanding of computation via naming (say) is just as much a logical activity as is the use of modal logics (say) in computer science.

Background and related work The work on π-calculus really began with a failure, at the time that I wrote about CCS, the Calculus of Communicating Systems [15]. This was the failure, in discussion with Mogens Nielsen at Aarhus in 1979, to see how full mobility among processes could be handled algebraically. The wish to do this was motivated partly by Hewitt's actor systems, which he introduced much earlier [12]. Several years later Engberg and Nielsen [8] succeeded in giving an algebraic formulation. The π-calculus [19] is a simplification and strengthening of their work.

Meanwhile other authors had invented and applied formalisms for processes without the restriction of a finite fixed initial connectivity. Two prominent examples are the DyNe language of Kennaway and Sleep [14], and the work on parametric channels by Astesiano and Zucca [3]. These works are comparable to the π-calculus because they achieve mobility by enriching the handling of *channels*.

By contrast, one can also achieve mobility by the powerful means of transmitting *processes* as messages; this is the *higher-order* approach. It is well exemplified by the work Astesiano and Reggio [2] in the context of general algebraic specification, F. Nielson [22] with emphasis upon type structure, Boudol [6] in the context of λ-calculus, and Thomsen [27]. It has been a deliberate intention in the π-calculus to avoid higher order initially, since the goal was to demonstrate that in some sense it is sufficiently powerful to allow only *names* or *channels* to be the content of communications. Indeed Thomsen's work supports this conjecture, and the present work strengthens his results comparing the approaches. See Milner [17] for a discussion contrasting the approaches.

Outline There are six short chapters following this introduction.

Chapter 2 reviews the formalism of the monadic π-calculus, essentially as it was presented in [19]; it also defines the notion of structural congruence and the reduction relation as first given in [17].

Chapter 3 is entirely devoted to applications; the first defines a simple mobile telephone protocol, the second encodes arithmetic in π-calculus, and the third presents two useful disciplines of name-use (such as may be obeyed in an operating system) in the form of properties invariant under reduction.

Chapter 4 generalizes π-calculus to polyadic communications, introduces the notions of *abstraction* and *concretion* which enhance the power of expression of the calculus (illustrated by a simple treatment of truth values), and affirms that the reduction relation remains essentially unchanged.

Chapter 5 and Chapter 6 provide the technical basis of the work. In Chapter 5, first *reduction congruence* is defined; this is a natural congruence based upon reduction and observability. Next, the standard operational semantics of [19] is reformulated in terms of a new notion, *commitment*; this, together with the flexibility which abstractions and concretions provide, yields a very succinct presentation. Then the (late) bisimilarity of [19] is restated in the polyadic setting, with its axiomatization. Its slightly weaker variant *early* bisimilarity, discussed in

Part II of [19], is shown to induce a congruence identical with reduction congruence. Some theorems about replication are given. Finally, the modal logic of [20], which provides characterizations of both late and early bisimilarity, is formulated in a new way – again taking advantage of the new setting.

Chapter 6 introduces the notions of *sort* and *sorting*, which are somewhat analogous to the simple type hierarchy in λ-calculus, but with significant differences. Data structures are shown to be represented as a particularly well-behaved class of processes, which moreover respect a distinctive *sorting* discipline. Finally, with the help of sorts, new light is cast upon the encoding of λ-calculus into π-calculus first presented in [17]; a simple proof is given of the validity of β-conversion in this interpretation of λ-calculus, using theorems from Chapter 5.

Chapter 7 explores higher-order processes, extending the work of Thomsen [27]. It is shown how sorts and sorting extend naturally not only to second-order (processes-as-data), but even to ω-order; a key rôle is played here by abstractions. A theorem of Sangiorgi [26] is given which asserts that these ω-order processes can be faithfully encoded in the first-order π-calculus (i.e. the calculus of Chapter 4). Some details of this encoding are given.

Acknowledgements I thank Joachim Parrow and David Walker for the insights which came from our original work together on π-calculus, and which have deeply informed the present development. I also thank Davide Sangiorgi and Bent Thomsen for useful discussions, particularly about higher-order processes. I am most grateful to Dorothy McKie for her help and skill in preparing this manuscript.

The work was carried out under a Senior Fellowship funded by the Science and Engineering Research Council, UK.

2 The Monadic π-calculus

2.1 Basic ideas

The most primitive entity in π-calculus is a *name*. Names, infinitely many, are $x, y, \ldots \in \mathcal{X}$; they have no structure. In the basic version of π-calculus which we begin with, there is only one other kind of entity; a *process*. Processes are $P, Q, \ldots \in \mathcal{P}$ and are built from names by this syntax

$$P \quad ::= \quad \Sigma_{i \in I} \pi_i . P_i \;\;\big|\;\; P \mid Q \;\;\big|\;\; !P \;\;\big|\;\; (\nu x)P$$

Here I is a finite indexing set; in the case $I = \emptyset$ we write the sum as $\mathbf{0}$. In a summand $\pi.P$ the prefix π represents an *atomic action*, the first action performed by $\pi.P$. There are two basic forms of prefix:

$x(y)$, which binds y in the prefixed process, means "input some name – call it y – along the link named x",

$\overline{x}y$, which does not bind y, means "output the name y along the link named x".

In each case we call x the *subject* and y the *object* of the action; the subject is *positive* for input, *negative* for output.

A name refers to a link or a channel. It can sometimes be thought of as naming a process at "the other end" of a channel; there is a polarity of names, and $\overline{x}$ – the *co-name* of x – is used for output, while x itself is used for input. But there are two reasons why "naming a process" is not a good elementary notion. The first is that a process may be referred to by many names; it may satisfy different demands, along different channels, for many clients. The second is that a name may access many processes; I may request a resource or a service – e.g. I may cry for help – from any agent able to supply it. In fact, if we had names for processes we would have to have (a different kind of) names for channels too! This would oppose the parsimony which is essential in a basic model.

Of course in human communities it is often convenient, and a convention, that a certain name is borne uniquely by a certain member (as the name "Robin" is borne uniquely by me in my family, but not in a larger community). So, in process communities it will sometimes be a convention that a name x is borne uniquely by a certain process, in the sense that only this member will use the name x as a (positive) subject; then those addressing the process will use the co-name $\overline{x}$ as a (negative) subject. But conventions are not maintained automatically; they require discipline! In fact, that a name is uniquely borne is an invariant which is useful to prove about certain process communities, such as distributed operating systems.

We dwelt at length on this point about naming, because it illustrates so well the point made in the introduction about ontological commitment. We now return to describing the calculus.

The summation form $\Sigma\pi_i.P_i$ represents a process able to take part in one – but only one – of several alternatives for communication. The choice is not made by the process; it can never commit to one alternative until it occurs, and this occurrence precludes the other alternatives. Processes in this form are called *normal processes* (because as we see later, all processes can be converted to this *normal form*). For normal processes $M, N, \ldots \in \mathcal{N}$ we shall use the following syntax:

$$N \ ::= \ \pi.P \ \big| \ \mathbf{0} \ \big| \ M+N$$

In this version of π-calculus we confine summation to normal processes, though previously we have allowed the form $P+Q$ for arbitrary processes. One reason is that the reduction rules in Section 2.4 are simpler with this constraint; another is that forms such as $(P|Q)+R$ have very little significance. However, everything in this paper can be adjusted to allow for the more general use of summation.

What do the last three forms of process mean? $P|Q$ – "P par Q" – simply means that P and Q are concurrently active, so they can act independently – but can also communicate. $!P$ – "bang P" – means $P|P|\ldots$; as many copies as you wish. There is no risk of infinite concurrent activity; our reduction rules will see to that. The operator "!" is called *replication*. A common instance of replication is $!\pi.P$ – a resource which can only be replicated when a requester communicates via π.

Finally, $(\nu x)P$ – "new x in P" – restricts the use of the name x to P. Another way of describing it is that it declares a new unique name x, distinct from all external names, for use in P. The behaviour of (νx) is subtle. In fact, the character of the π-calculus derives from the interplay between its two binding operators: $x(y)$ which binds y somewhat as λy binds y in the λ-calculus, and (νx) which has no exact correlate in other calculi (but is the restriction operator of CCS promoted to a more influential rôle).

Before looking at examples, we introduce a convenient abbreviation. Processes like $x(y).\mathbf{0}$ and $\overline{x}y.\mathbf{0}$ are so common that we prefer to omit the trailing "$.\mathbf{0}$" and write just $x(y)$ and $\overline{x}y$.

2.2 Some simple examples

Consider the process

$$\overline{x}y.\mathbf{0} \mid x(u).\overline{u}v.\mathbf{0} \mid \overline{x}z.\mathbf{0}$$

which we now abbreviate to

$$\overline{x}y \mid x(u).\overline{u}v \mid \overline{x}z$$

Call it $P \mid Q \mid R$. One of two communications (but not both) can occur along the channel x; P can send y to Q, or R can send z to Q. The two alternatives for the result are

$$\mathbf{0} \mid \overline{y}v \mid \overline{x}z \quad \text{or} \quad \overline{x}y \mid \overline{z}v \mid \mathbf{0}$$

Note that R has become $\overline{y}v$ or $\overline{z}v$; thus, the communication has determined which channel R can next use for output, y or z.

Now consider a variant

$$(\nu x)(\overline{x}y \mid x(u).\overline{u}v) \mid \overline{x}z$$

In this case, the (free) x in R is quite different from the (bound) x in P and Q, so only one communication can happen, yielding

$$\mathbf{0} \mid \overline{y}v \mid \overline{x}z$$

(The restriction (νx) has vanished; it has no work left to do, since the x which it restricted has been used up by the communication.)

Third, consider

$$\overline{x}y \mid !x(u).\overline{u}v \mid \overline{x}z$$

This differs from the first case, because Q is now replicated. So $!Q$ can first spin off one copy to communicate with P, and the system becomes

$$\mathbf{0} \mid \overline{y}v \mid !Q \mid \overline{x}z$$

Then $!Q$ can spin off another copy to communicate with R, and the system becomes

$$\mathbf{0} \mid \overline{y}v \mid !Q \mid \overline{z}v \mid \mathbf{0}$$

We have just seen several examples of *reduction*, i.e. the transformation of a process corresponding to a single communication. We now present the π-calculus reduction rules; the analogy with reduction in the λ-calculus is striking but so are the differences.

2.3 Structural Congruence

We have already said that there are two binding operators; the input prefix $x(y)$ (which binds y) and the restriction (νx). So we can define the *free names* $\mathrm{fn}(P)$, and the *bound names* $\mathrm{bn}(P)$ of a process P in the usual way. We extend these to prefixes; note

$$\begin{aligned} \mathrm{bn}(x(y)) = \{y\} \quad &, \quad \mathrm{fn}(x(y)) = \{x\} \\ \mathrm{bn}(\overline{x}y) = \emptyset \quad &, \quad \mathrm{fn}(\overline{x}y) = \{x, y\} \end{aligned}$$

Also, the *names* of a process P are $\mathrm{n}(P) \stackrel{\mathrm{def}}{=} \mathrm{bn}(P) \cup \mathrm{fn}(P)$.

Now, to make our reduction system simple, we wish to identify several expressions. A typical case is that we want $+$ and $\mid$ to be commutative and associative. We therefore define *structural congruence* $\equiv$ to be the smallest congruence relation over $\mathcal{P}$ such that the following laws hold:

1. Agents (processes) are identified if they only differ by a change of bound names

2. $(\mathcal{N}/\equiv, +, \mathbf{0})$ is a symmetric monoid

3. $(\mathcal{P}/\equiv, |, \mathbf{0})$ is a symmetric monoid

4. $!P \equiv P \mid !P$

5. $(\nu x)\mathbf{0} \equiv \mathbf{0}, (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$

6. If $x \notin \mathrm{fn}(P)$ then $(\nu x)(P|Q) \equiv P \mid (\nu x)Q$

Exercise Use 3, 5 and 6 to show that $(\nu x)P \equiv P$ when $x \notin \mathrm{fn}(P)$. ∎

Note that laws 1, 4 and 6 allow any restriction not inside a normal process to be pulled into outermost position; for example, if $P \equiv (\nu y)\overline{x}y$ then

$$\begin{aligned} x(z).\overline{y}z \mid !P &\equiv x(z).\overline{y}z \mid (\nu y)\overline{x}y \mid !P \\ &\equiv x(z).\overline{y}z \mid (\nu y')\overline{x}y' \mid !P \\ &\equiv (\nu y')(x(z).\overline{y}z \mid \overline{x}y') \mid !P \end{aligned}$$

This transformation has brought about the juxtaposition $x(z).\cdots \mid \overline{x}y'.\cdots$, which is reducible by the rules which follow below. The use of structural laws such as the above, to bring communicands into juxtaposition, was suggested by the Chemical Abstract Machine of Berry and Boudol [5].

2.4 Reduction rules

This section is devoted to defining the *reduction relation* $\rightarrow$ over processes; $P \rightarrow P'$ means that P can be transformed into P' by a single computational step. Now every computation step consists of the interaction between two normal terms. So our first reduction rule is *communication*:

$$\text{COMM}: \quad (\cdots + x(y).P) \mid (\cdots + \overline{x}z.Q) \rightarrow P\{z/y\} \mid Q$$

There are two ingredients here. The first is how communication occurs between two atomic normal processes $\pi.P$ which are complementary (i.e. whose subjects are complementary). The second is the discard of alternatives; either instance of "$\cdots$" can be $\mathbf{0}$ of course, but if not then the communication pre-empts other possible communications.

COMM is the only axiom for $\rightarrow$; otherwise we only have inference rules, and they are three in number. The first two say that reduction can occur underneath composition and restriction, while the third simply says that structurally congruent terms have the same reductions.

$$\text{PAR}: \frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \qquad \text{RES}: \frac{P \rightarrow P'}{(\nu x)P \rightarrow (\nu x)P'}$$

$$\text{STRUCT}: \frac{Q \equiv P \quad P \rightarrow P' \quad P' \equiv Q'}{Q \rightarrow Q'}$$

Exercise In Section 2.2 and the previous exercise several reductions were given informally. Check that they have all been inferred from the four rules for $\rightarrow$. ■

It is important to see what the rules do *not* allow. First, they do not allow reductions underneath prefix, or sum; for example we have

$$u(v).(x(y) \mid \overline{x}z) \not\rightarrow$$

Thus prefixing imposes an order upon reduction. This constraint is not necessary. However, the calculus changes non-trivially if we relax it, and we shall not consider the possibility further in this paper.

Second, the rules do not allow reduction beneath replication. In some sense, this does not reduce the computational power; for if we have $P \rightarrow P'$ then, instead of inferring $!P \rightarrow !P'$, which is equivalent to allowing unboundedly many coexisting copies of P to reduce, we can always infer

$$!P \equiv \underbrace{P \mid P \mid \cdots \mid P}_{n \text{ times}} \mid !P \rightarrow^n P' \mid P' \mid \cdots \mid P' \mid !P$$

thus (in n reductions) reducing as many copies of P as we require – and for finite work we can only require finitely many !

Third, the rules tell us nothing about *potential* communication of a process P with other processes. From the reduction behaviour alone of P and Q separately, we cannot infer the whole reduction behaviour of, say, $P|Q$. (This is just as in the λ-calculus, where λxx and λxxx have the *same* reduction behaviour – they have no reductions – but applying them to the same term λyy gives us two terms $(\lambda xx)(\lambda yy)$ and $(\lambda xxx)(\lambda yy)$ with *different* reduction behaviour.)

If we wish to identify *every* potential communication of a process, so as to distinguish say $\overline{x}y$ from $\overline{x}z$, then we would indeed become involved with the familiar labelled transition systems used in process algebra (and introduced later in this paper). We do not want to do this yet. But for technical reasons we want to do a little of it. To be precise, we only want to distinguish processes which can perform an external communication at some location α – a name or co-name – from those which cannot. So we give a few simple definitions.

First, we say that Q occurs *unguarded* in P if it occurs in P but not under a prefix. Thus, for example, Q is unguarded in $Q|R$ and in $(\nu x)Q$ but not in $x(y).Q$. Then we say P is *observable at* α – and write $P \downarrow_\alpha$ – if some $\pi.Q$ occurs unguarded in P, where α is the subject of π and is unrestricted. Thus $x(y) \downarrow_x$ and $(\nu z)\overline{x}z \downarrow_{\overline{x}}$, but $(\nu x)\overline{x}z \not\downarrow_{\overline{x}}$; also $(\nu x)(x(y) \mid \overline{x}z) \not\downarrow_x$ even though it has a reduction.

It turns out that we get an interesting congruence over $\mathcal{P}$ in terms of $\rightarrow$ and $\downarrow_\alpha$. This will be set out in Chapter 4; first we digress in Chapter 3 to look at several applications.

3 Applications

In this section, we give some simple illustrations of the π-calculus. We begin by introducing a few convenient derived forms and abbreviations.

3.1 Some derived forms

In applications, we often want forms which are less primitive than the basic constructions of monadic π-calculus. One of the first things we find useful is multiple inputs and outputs along the same channel. A natural abbreviation could be to write e.g. $x(yz)$ for $x(y).x(z)$ and $\overline{x}yz$ for $\overline{x}y.\overline{x}z$. But this would give a misleading impression about the indivisibility of the pair of actions in each case. Consider

$$x(yz) \mid \overline{x}y_1z_1 \mid \overline{x}y_2z_2$$

for example; the intention is that y, z should get bound to either y_1, z_1 or y_2, z_2. But if we adopt the above abbreviations there is a third possibility, which is a mix-up; y, z can get bound to y_1, y_2. To avoid this mix-up, a way is needed of making a *single* commitment to any multiple communication, and this can be done using private (i.e. restricted) names. So we introduce abbreviations

$$\begin{array}{rcl} x(y_1 \cdots y_n) & \text{for} & x(w).w(y_1).\cdots.w(y_n) \\ \overline{x}y_1 \cdots y_n & \text{for} & (\nu w)\overline{x}w.\overline{w}y_1.\cdots.\overline{w}y_n \end{array}$$

– writing just x for $x()$ when $n = 0$. You can check that the mix-up in the example is no longer possible. The abbreviation has introduced an extra communication, even in the case $n = 1$, but this will cause no problem.

Next, we often wish to define parametric processes recursively. For example, we may like to define A and B, of arity 1 and 2 respectively, by

$$A(x) \stackrel{\mathrm{def}}{=} x(yz).B(y, z) \quad , \quad B(y, z) \stackrel{\mathrm{def}}{=} \overline{y}z.A(z)$$

If we wish to allow such parametric process definitions of the general form $K(\vec{x}) \stackrel{\mathrm{def}}{=} P_K$, we add

$$P \ ::= \ \cdots \ \mid \ K(\vec{y})$$

to the syntax of processes, where K ranges over process identifiers; for each definition we also add a new structural congruence law $K(\vec{y}) \equiv P_K\{\vec{y}/\vec{x}\}$ to those given in Section 2.3.

However, it is easier to develop a theory if "definition-making" does not have to be taken as primitive. In fact, provided the number of such recursive definitions is finite, we can encode them by replication; then the introduction of new constants, with definitions, is just a matter of convenience. We shall content ourselves with showing how to encode a single recursive definition with a single parameter. Thus, suppose we have

$$A(x) \stackrel{\mathrm{def}}{=} P$$

where we assume that $\mathrm{fn}(P) \subseteq \{x\}$, and that P may contain occurrences of A (perhaps with different parameters). The idea is, first, to replace every recursive call $A(y)$ within P by a little process $\overline{a}y$ which excites a new copy of P. (Here a is a new name.) Let us denote by $\widehat{P}$ the result of doing these replacements in P. Then the replication

$$!a(x).\widehat{P}$$

corresponds to the parametric process $A(x)$. We now have to take care of the outermost calls of A. So let $A(z)$ occur in some system S; then we replace it by

$$(\nu a)(\overline{a}z \mid !a(x).\widehat{P})$$

Note that this places a separate copy of the replication at each call $A(z)$ in S. Alternatively one can make do with a single copy; transform S to $\widehat{S}$ by replacing each call $A(z)$ just by $\overline{a}z$, and then replace S by

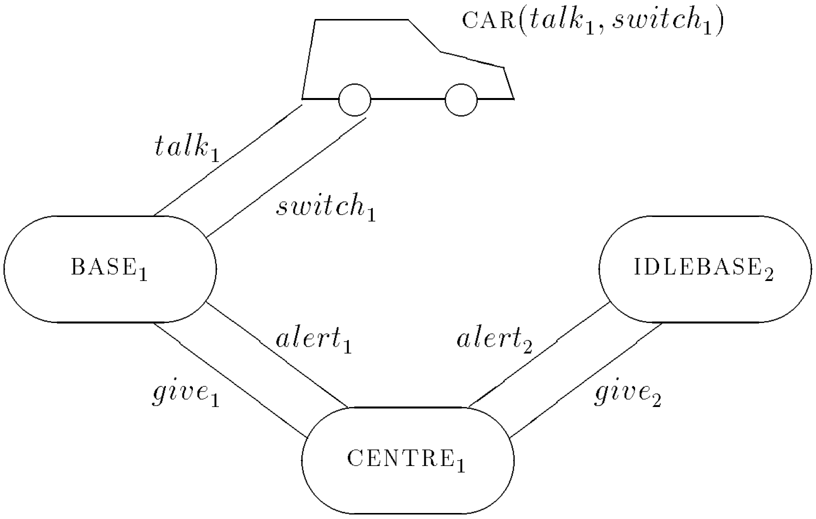
$$(\nu a)(\widehat{S} \mid !a(x).\widehat{P})$$

Of course, these translations do not behave identically with the original, because they do one more reduction for each call of A; but they are *weakly congruent* to the original (in the sense of [19]), which is all we would require in applications.

From now on, in applications we shall freely use parametric recursive definitions; but, knowing that translation is possible, in our theoretical development we shall ignore them and stick to replication.

3.2 Mobile telephones

Here is a "flowgraph" of our first application:



This is a simplified version of a system used by Orava and Parrow [23] to illustrate π-calculus. A CENTRE is in permanent contact with two BASE stations, each in a different part of the country. A CAR with a mobile telephone moves about the country; it should always be in contact with a BASE. If it gets rather far from its current BASE contact, then (in a way which we do not model) a hand-over procedure is initiated, and as a result the CAR relinquishes contact with one BASE and assumes contact with another.

The flowgraph shows the system in the state where the CAR is in contact with BASE_1; it may be written

$$\begin{aligned}\mathrm{SYSTEM}_1 \stackrel{\mathrm{def}}{=}\ & (\nu\, talk_i, switch_i, give_i, alert_i \ :\ i=1,2) \\ & \big(\mathrm{CAR}(talk_1, switch_1) \mid \mathrm{BASE}_1 \mid \mathrm{IDLEBASE}_2 \mid \mathrm{CENTRE}_1\big)\end{aligned}$$

What about the components?

A CAR is parametric upon a *talk* channel and a *switch* channel. On *talk* it can talk repeatedly; but at any time along *switch* it may receive two new channels which it must then start to use:

$$\begin{aligned}\mathrm{CAR}(talk, switch) \stackrel{\mathrm{def}}{=}\ & talk\,.\,\mathrm{CAR}(talk, switch) \\ & +\ switch(talk' switch')\,.\,\mathrm{CAR}(talk', switch')\end{aligned}$$

A BASE can talk repeatedly with the CAR; but at any time it can receive along its *give* channel two new channels which it should communicate to the CAR, and then become idle itself; we define

$$\begin{aligned}\mathrm{BASE}(t,s,g,a) \stackrel{\mathrm{def}}{=}\ & t\,.\,\mathrm{BASE}(t,s,g,a) \\ & +\ g(t's')\,.\,\overline{s}t's'\,.\,\mathrm{IDLEBASE}(t,s,g,a)\end{aligned}$$

An IDLEBASE, on the other hand, may be told on its *alert* channel to become active:

$$\mathrm{IDLEBASE}(t,s,g,a) \stackrel{\mathrm{def}}{=} a\,.\,\mathrm{BASE}(t,s,g,a)$$

We define the abbreviation

$$\mathrm{BASE}_i \stackrel{\mathrm{def}}{=} \mathrm{BASE}(talk_i, switch_i, give_i, alert_i) \qquad (i=1,2)$$

and a similar abbreviation $\mathrm{IDLEBASE}_i$. Thus, for example,

$$\begin{aligned}\mathrm{BASE}_i &\equiv talk_i\,.\,\mathrm{BASE}_i + give_i(t's')\,.\,\overline{switch_i}t's'\,.\,\mathrm{IDLEBASE}_i \\ \mathrm{IDLEBASE}_i &\equiv alert_i\,.\,\mathrm{BASE}_i\end{aligned}$$

Finally the CENTRE, which initially knows that the CAR is in contact with BASE_1, can decide (according to information which we do not model) to transmit the channels $talk_2$, $switch_2$ to the CAR via BASE_1, and alert BASE_2 of this fact. So we define

$$\begin{aligned}\mathrm{CENTRE}_1 &\stackrel{\mathrm{def}}{=} \overline{give_1}talk_2 switch_2\,.\,alert_2\,.\,\mathrm{CENTRE}_2 \\ \mathrm{CENTRE}_2 &\stackrel{\mathrm{def}}{=} \overline{give_2}talk_1 switch_1\,.\,alert_1\,.\,\mathrm{CENTRE}_1\end{aligned}$$

Exercise Check carefully that indeed SYSTEM_1 reduces in three steps to SYSTEM_2, which is precisely SYSTEM_1 with the subscripts 1 and 2 interchanged. The reduction is (using $\vec{c}$ for the set of eight restricted channels):

$$\begin{aligned}
\text{SYSTEM}_1 \;\equiv\;& (\nu\vec{c})\Big(\text{CAR}(talk_1, switch_1) \mid \text{BASE}_1 \mid \text{IDLEBASE}_2 \mid \text{CENTRE}_1\Big) \\
\rightarrow\;& (\nu\vec{c})\Big(\text{CAR}(talk_1, switch_1) \mid \overline{switch_1} talk_2 switch_2 \,.\, \text{IDLEBASE}_1 \\
&\qquad \mid \text{IDLEBASE}_2 \mid alert_2 \,.\, \text{CENTRE}_2\Big) \\
\rightarrow\;& (\nu\vec{c})\Big(\text{CAR}(talk_2, switch_2) \mid \text{IDLEBASE}_1 \\
&\qquad \mid \text{IDLEBASE}_2 \mid alert_2 \,.\, \text{CENTRE}_2\Big) \\
\rightarrow\;& (\nu\vec{c})\Big(\text{CAR}(talk_2, switch_2) \mid \text{IDLEBASE}_1 \mid \text{BASE}_2 \mid \text{CENTRE}_2\Big) \\
\equiv\;& \text{SYSTEM}_2
\end{aligned}$$

■

Of course this example is highly simplified. Consider one possible refinement. There is no reason why the number of available (*talk, switch*) channel-pairs is equal to the number of BASES; nor that each base always uses the same channel-pair. The reader may like to experiment with having an arbitrary (fixed) number of BASES; at each handover the new BASE could be chosen at random, and a channel-pair picked from a store of available channel pairs maintained (say) in a queue.

3.3 Numerals and arithmetic

For our second application we show that arithmetic can be done in π-calculus in much the same way as it can in λ-calculus. Church represented the natural number n in λ-calculus by

$$\lambda f \lambda x f^n(x)$$

– i.e. the function which iterates its function argument n times.

As a first attempt in π-calculus, we may choose to represent n by the parametric process

$$\underline{n}(x) \;\stackrel{\text{def}}{=}\; \underbrace{\overline{x}.\cdots.\overline{x}}_{n \text{ times}}$$

which we abbreviate to $(\overline{x}.)^n$. But this process cannot be tested for zero, and the arithmetic operators (coded also as processes) will need a test for zero.

So we give $\underline{n}$ two parameters, one representing successor, and the other representing zero:

$$\underline{n}(xz) \;\stackrel{\text{def}}{=}\; (\overline{x}.)^n\, \overline{z}$$