Review of **The π-calculus: A Theory of Mobile Processes**[*]

Riccardo Pucella

Department of Computer Science
Cornell University

July 8, 2001

Introduction

With the rise of computer networks in the past decades, the spread of distributed applications with components across multiple machines, and with new notions such as mobile code, there has been a need for formal methods to model and reason about concurrency and mobility. The study of sequential computations has been based on notions such as Turing machines, recursive functions, the λ-calculus, all equivalent formalisms capturing the essence of sequential computations. Unfortunately, for concurrent programs, theories for sequential computation are not enough. Many programs are not simply programs that compute a result and return it to the user, but rather interact with other programs, and even move from machine to machine.

Process calculi are an attempt at getting a formal foundation based on such ideas. They emerged from the work of Hoare [4] and Milner [6] on models of concurrency. These calculi are meant to model systems made up of processes communicating by exchanging values across channels. They allow for the dynamic creation and removal of processes, allowing the modelling of dynamic systems. A typical process calculus in that vein is CCS [6, 7]. The π-calculus extends CCS with the ability to create and remove communication links between processes, a new form of dynamic behaviour. By allowing links to be created and deleted, it is possible to model a form of *mobility*, by identifying the position of a process by its communication links.

This book, “The π-calculus: A Theory of Mobile Processes”, by Davide Sangiorgi and David Walker, is a in-depth study of the properties of the π-calculus and its variants. In a sense, it is the logical followup to the recent introduction to concurrency and the π-calculus by Milner [8], reviewed in SIGACT News, 31(4), December 2000.

What follows is a whirlwind introduction to CCS and the π-calculus. It is meant as a way to introduce the notions discussed in much more depth by the book under review. Let us start with the basics. CCS provides a syntax for writing processes. The syntax is minimalist, in the grand tradition of foundational calculi such as the λ-calculus. Processes perform actions, which can be of three forms: the sending of a message over channel x (written $\overline{x}$), the receiving of a message over channel x (written x), and internal actions (written τ), the details of which are unobservable. Send and receive actions are called *synchronization* actions, since communication occurs when the corresponding processes synchronize. Let α stand for actions, including the internal action τ, while we reserve $\lambda, \mu, \ldots$ for synchronization actions.[1] Processes are written using the following syntax:

$$P \quad ::= \quad A\langle x_1, \ldots, x_k\rangle \mid \sum_{i \in I} \alpha_i.P_i \mid P_1 | P_2 \mid \nu x.P$$

We write 0 for the empty summation (when $I = \emptyset$). The idea behind process expressions is simple. The process 0 represents the process that does nothing and simply terminates. A process of the form $\lambda.P$ awaits to synchronize

[*] D. Sangiorgi, D. Walker, *The π-calculus: A Theory of Mobile Processes*, Cambridge University Press, 2001, 580pp, ISBN 0521781779.

[1] In the literature, the actions of CCS are often given a much more abstract interpretation, as simply names and co-names. The send/receive interpretation is useful when one moves to the π-calculus.

with a process of the form $\overline{\lambda}.Q$, after which the processes continue as process P and Q respectively. A generalization of such processes is $\sum_{i \in I} \alpha_i.P_i$, which nondeterministically synchronizes via one of its α_i only. We will write $\sum_{i \in I} \alpha_i.P_i$ as $\alpha_1.P_1 + \cdots + \alpha_n.P_n$ when the set I is $\{1, \ldots, n\}$. To combine processes, the parallel composition $P_1|P_2$ is used. Note the difference between summation and parallel composition: a summation offers a choice, so only one of the summands can synchronize and proceed, while a parallel composition allows all its component processes to proceed. (This will be made clear when we get to the transition rules describing how processes execute.) The process expression $\nu x.P$ defines a local channel name x to be used within process P. This name is guaranteed to be unique to P (possibly through consistent renaming). Finally, we allow process definitions, where we assume that every identifier A is associated with a process definition of the form $A(x_1, \ldots, x_k) = P_A$ where $x_1, \ldots, x_k$ are free in P_A. To instantiate the process A to values $y_1, \ldots, y_k$, you write $A\langle y_1, \ldots, y_k \rangle$.

As an example, consider the process $(x.y.0 + \overline{x}.z.0)|\overline{x}.0|\overline{y}.0$. Intuitively, it consists of three processes running in parallel: the first offers of choice of either receiving over channel x, or sending over channel x, the second sends over channel x, and the third sends over channel y. Depending on which choice the first process performs (as we will see, this depends on the actions the other process can perform), it can continue in one of two ways: if it chooses to receive on channel x (i.e., the $x.y.0$ summand is chosen), it can then receive on channel y, while if it chooses to send on x (i.e., the $\overline{x}.z.0$ summand is chosen), it can then receive on channel z.

To represent the execution of a process expression, we define the notion of a transition. Intuitively, the transition relation tells us how to perform one step of execution of the process. Note that since there can be many ways in which a process executes, the transition is fundamentally nondeterministic. The transition of a process P into a process Q by performing an action α is indicated $P \xrightarrow{\alpha} Q$. The action α is the observation of the transition. (We will sometimes simply use $\longrightarrow$ when the observation is unimportant.) The transition relation is defined by the following inference rules:

$$\frac{}{\sum_{i \in I} \alpha_i.P_i \xrightarrow{\alpha_j} P_j} \text{ for } j \in I \qquad \frac{P \xrightarrow{\lambda} P' \quad Q \xrightarrow{\overline{\lambda}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

$$\frac{P \xrightarrow{\alpha} P'}{\nu x.P \xrightarrow{\alpha} \nu x.P'} \text{ if } \alpha \notin \{x, \overline{x}\} \qquad \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q}$$

$$\frac{\{\vec{y}/\vec{x}\}P_A \xrightarrow{\alpha} P'}{A\langle \vec{y} \rangle \xrightarrow{\alpha} P'} \text{ if } A(\vec{x}) = P_A \qquad \frac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P|Q'}$$

For example, consider the transitions of the example process above, $(x.y.0 + \overline{x}.z.0)|\overline{x}.0|\overline{y}.0$. A possible first transition (the first step of the execution, if you wish), can be derived as follows:

$$\frac{\dfrac{x.y.0 + \overline{x}.z.0 \xrightarrow{x} y.0 \quad \overline{x}.0 \xrightarrow{\overline{x}} 0}{(x.y.0 + \overline{x}.z.0)|\overline{x}.0 \xrightarrow{\tau} y.0|0}}{(x.y.0 + \overline{x}.z.0)|\overline{x}.0|\overline{y}.0 \xrightarrow{\tau} y.0|0|\overline{y}.0}$$

That is, the process reduces to $y.0|0|\overline{y}.0$ in one step that does not provide outside information, since it appears as an internal action. Note that the 0 can be removed from the resulting process, as it does not contribute further to the execution of the process. The resulting process $y.0|\overline{y}.0$ can then perform a further transition, derived as follows:

$$\frac{y.0 \xrightarrow{y} 0 \quad \overline{y}.0 \xrightarrow{\overline{y}} 0}{y.0|\overline{y}.0 \xrightarrow{\tau} 0|0}$$

In summary, a possible sequence of transitions for the original process is the two-step sequence

$$(x.y.0 + \overline{x}.z.0)|\overline{x}.0|\overline{y}.0 \xrightarrow{\tau} y.0|\overline{y}.0 \xrightarrow{\tau} 0.$$

A central concept in the study of processes is that of equivalence of processes. We have in fact implicitly used a notion of equivalence in the example above, when we removed processes of the form 0 from parallel processes.

Many notions of equivalence can be defined, capturing the various intuitions that lead us to think of two processes as equivalent. A standard notion of equivalence is strong bisimulation. A strong bisimulation is a relation $\mathcal{R}$ such that whenever $P\mathcal{R}Q$, if $P \xrightarrow{\alpha} P'$, then there exists Q' such that $Q \xrightarrow{\alpha} Q'$ and $P'\mathcal{R}Q'$, and if $Q \xrightarrow{\alpha} Q'$, then there exists P' such that $P \xrightarrow{\alpha} P'$ and $P'\mathcal{R}Q'$. We say P and Q are strongly bisimilar if there exists a strong bisimulation $\mathcal{R}$ such that $P\mathcal{R}Q$. In other words, if P and Q are strongly bisimilar, then whatever transition P can take, Q can match it with one of its own that retains all of P's options, and vice versa.

Strong bisimulation is a very fine equivalence relation—not many processes end up being equivalent. More worryingly, strong bisimulation does not handle internal actions very well. Intuitively, process equivalence should really only involve observable actions. Two processes that only perform internal actions should be considered equivalent. For instance, the processes $\tau.\tau.0$ and $\tau.0$ should really be considered equivalent, as they really do nothing after performing some internal (and hence really unobservable) actions. Unfortunately, it is easy to check that these two processes are not strongly bisimilar. To capture this intuition, we define a weaker notion of equivalence, aptly called weak bisimulation.

Let $P \Longrightarrow Q$ denote that P can take any number of transitions before turning into Q. In other words, $P \Longrightarrow Q$ holds if $P \longrightarrow \cdots \longrightarrow Q$, i.e., $\Longrightarrow$ is the reflexive transitive closure of $\longrightarrow$. We write $P \overset{\alpha}{\Longrightarrow} Q$ if $P \Longrightarrow P' \xrightarrow{\alpha} Q' \Longrightarrow Q$, i.e., if P can take any number of transitions before and after doing an α-transition. A weak bisimulation is a relation $\mathcal{R}$ such that whenever $P\mathcal{R}Q$, if $P \xrightarrow{\tau} P'$ then there exists Q' such that $Q \Longrightarrow Q'$, while if $P \xrightarrow{\lambda} P'$ then there exists Q' such that $Q \overset{\lambda}{\Longrightarrow} Q'$, and vice versa for Q. We say that P and Q are weakly bisimilar if there exists a weak bisimulation $\mathcal{R}$ such that $P\mathcal{R}Q$. If P and Q are weakly bisimilar, an internal transition by P can be matched by zero or more transitions by Q, while an α-transition by P can be matched by one or more transition by Q as long as one such is an α-transition, and vice versa. One can check that a strong bisimulation is a weak bisimulation, but the converse does not hold: weak bisimilarity is a coarser equivalence relation.

In the framework above, we cannot communicate any value at synchronization time. It is not difficult to extend the calculus to allow for the exchange of values such as integers over the channel during a synchronization. Doing so does not fundamentally change the character of the calculus. A variation on this, however, does change the calculus in a highly nontrivial way, and that is to allow for the communication of channel names during synchronization. This yields the π-calculus. To see why such an extension might be useful, consider the following scenario. It shows that passing channel names around can be used to model process mobility. Intuitively, a process is characterized by the channels it exposes to the world, that can be used to communicate with it. These channels act as an interface to the process. Hence, the process $P = (x.y.0|\overline{x}.z.0)$ provides the channel x as an interface. A process that sends x to another process in some sense sends the capability to access P to that process. This captures the mobility of process P, although more accurately it captures the mobility of the capability to access P. It was Sangiorgi's original contribution to the theory of the π-calculus to show that capability mobility could indeed express in a precise sense process mobility [9]. (This particular topic is covered in Part V of the book.)

The necessary modifications to the calculus that capture the above intuition are straightforward. Syntactically, we need to change the kind of guards that can appear in summands. Sends now must carry a value, and receives must accept an identifier to be bound to the received value. The only values that can be exchanged are channel names. We define a prefix π to be of the following form:

$$\pi \quad ::= \quad \overline{x}\langle y\rangle \mid x(y) \mid \tau$$

Here, x and y are channel names, and τ is the internal action. Processes are described by the following syntax, where the only difference from CCS is in the summations:

$$P \quad ::= \quad A\langle x_1,\ldots,x_k\rangle \mid \sum_{i\in I}\pi_i.P_i \mid P_1|P_2 \mid \nu x.P$$

The intuition behind the new prefixes should be rather clear: as before, a process of the form $\tau.P$ performs an internal action τ before becoming P; a process of the form $\overline{x}\langle y\rangle.P$ is ready to send the channel name y onto channel x, and when this process synchronizes it behaves as P; a process of the form $x(y).P$ is ready to receive a channel name from channel x, and when this process synchronizes, it binds the received channel to the identifier y in P before continuing as the (modified) P. Note that the calculus in Sangiorgi and Walker's book is slightly different than the one presented here, which has been kept simple for reasons of exposition.

As an example of a process in the π-calculus, consider the process $x(y).\overline{y}\langle z\rangle.0|\overline{x}\langle w\rangle$. Intuitively, the second process sends channel name w via channel x to the first process, who binds it to name y. The first process then sends channel name z over this channel. Hence, the above process "reduces" to the process $\overline{w}\langle z\rangle$. Although the intuition underlying passing channel names over channels should be clear, it turns out that formalizing this in a nice way is difficult. Already describing the transition relation (in the form we gave above) is complicated by the fact that we cannot simply consider channel names, but also need to take into account the information exchanged at synchronization time. Similarly, describing the appropriate notion of bisimulation in such a setting needs to account for the information exchanged. Rather than describing these, I will defer to Sangiorgi and Walker's book, since in a sense this is exactly where the book picks up.

The book

The book splits into seven parts. Each parts comprises between two and three chapters, and deals with a particular aspect of the π-calculus.

Part I, **The π-calculus**, is made up of two chapters. Chapter 1 introduces the basic concepts of the π-calculus, starting from the syntax, and defines two notion of system behaviour. The first notion, called reduction, can be understood as a simplified account of what we described above. Essentially, reduction tells you how a term rewrites upon execution, without keeping track of the actions performed by the process. This notion of behaviour has the advantage of being simple and intuitive. The second notion of behaviour, in terms of labelled transition, follows the account I gave in the introduction. The relationship between these two notions is made explicit.

Behavioural equivalence of processes occupies a central part in the theory, and an initial take on the problem is given in Chapter 2. Many notion of equivalence can be defined for the π-calculus, and the fundamental ones are studied in this chapter. The main form of equivalence, barbed congruence, is defined naturally by specifying that no difference between equivalent processes can be observed by placing them into an arbitrary π-calculus context. This natural notion of equivalence turns out to be awkward to work with, and definitions based on ideas similar to bisimilarity given above can be introduced. It turns out that strong bisimilarity defined in the most natural way is equivalent to barbed congruence for the π-calculus. (This equivalence however fails when extensions to the π-calculus are considered in Parts IV and V, where barbed congruence remains the natural notion of equivalence.)

Part II, **Variations of the π-calculus**, is made up of three chapters. Chapter 3 studies various simple modifications to the basic π-calculus framework given in Part I. A variant of the π-calculus, the polyadic π-calculus, is introduced, where tuples of names can be passed at synchronization time. Adding tuples in such a way forces the introduction of sorts, which intuitively ensure that synchronizing terms agree as to the number of names that are being exchanged. This can be viewed as a primitive form of typing for the π-calculus. The second variation considers the addition of recursive definitions to the π-calculus. These variations are minor in the sense that they do not add expressive power: anything that can be expressed using tuples or recursive definitions can already be expressed in the basic π-calculus of Part I.

Chapter 4 returns to behavioural equivalence. New notions of equivalence are defined, such as ground, late, and open bisimilarity. Roughly speaking, these new bisimilarity definitions ease the demand on the processes to mimic one another's input actions. The main advantage of these equivalences is that they are easier to establish than the more natural notions of equivalence, a recurring concern, especially in the context of automatic tools for reasoning about processes. After studying these new equivalence relations, the question of axiomatizing these equivalences is addressed. An axiomatization for an equivalence relation on processes is a set of equational axiom on process expressions that, together with the rules of equational reasoning (i.e., reflexivity, transitivity, etc.), suffice for proving exactly the valid equations between process expressions, with respect to the equivalence under consideration.

While the variants of the π-calculus examined in Chapter 3 are extensions that do not change the expressive power of the calculus, Chapter 5 studies restrictions to the calculus that lend insight into various phenomena of interaction and mobility. The asynchronous π-calculus restricts terms that perform a send action to be of the form $\overline{x}\langle y\rangle.0$, that is, to become the null process after synchronization. This can be used to capture a form of asynchrony, and the resulting calculus is provably less expressive than the full π-calculus. The localized π-calculus has essentially the restriction that a name received by a process cannot be used as an input channel—it must be either used for sending, or sent to another process. Hence, all input channels are localized in the process in which they are defined. Finally, the private

π-calculus imposes the restriction on the π-calculus that local names cannot be exported, that is, they cannot be sent to a process outside of the scope of the ν where the channel is defined. For all these subcalculi, notions of equivalences are studied.

Part III, **Typed π-calculi**, explores the issue of assigning types to π-calculus expressions. This part is mostly concerned with defining type systems to detect errors statically, or to enforce properties statically. (The following part focuses on types as an aid for reasoning about the behaviour of processes.) In Chapter 6, the foundations of type systems for the π-calculus are laid. The Base-π calculus is introduced, essentially CCS extended with the capability of passing values at synchronization time. The types are associated with those values. Channels are also given a type, stating the type of value they carry. Processes themselves are also given a type, all processes getting the same type. The simply-typed π-calculus is obtained by adding channel names to the values that can be exchanged, and modifying the type system accordingly.

Chapter 7 extends the basic type system of the simply-typed π-calculus with the notion of subtyping. In order for this to be nontrivial, the calculus is refined to differentiate, in the type of channels, whether or not the channel is an input channel (used for receiving values) or an output channel (used for sending values). We can then refine the type system to account for subtyping: if we have an output channel that can send values of type S, then clearly we can also use the channel to send values of any subtype of S. Various properties of subtyping are examined.

The type systems described above are fairly standard. In Chapter 8, more advanced type systems are investigated. These are meant to capture various properties of processes that we may want to enforce. For instance, it is possible to deal with linearity constraints in the type system, for example, that a given channel name can only be used once for input or output. Another property is that of receptiveness, that is, that a given channel name is always ready to process some input, or equivalently that sending a value on that channel will never deadlock. Another property is polymorphism, namely the fact that some processes don't really care about the actual type of the value they process. An example of such a process is one that simply forwards a value from one channel to another. Type systems to capture these properties are defined and studied.

Part IV, **Reasoning about processes using types**, explores another advantage of type systems beyond static enforcement, that of helping reasoning about behavioural properties of processes. The three chapters in this part mirror those of Part III. Specifically, Chapter 9 discusses how types can help in reasoning. The example examined in the chapter is that of a security property, specifically that a given name always remains private to a given process. Behavioural equivalences on typed processes are investigated. In Chapter 10, reasoning about typed processes in the context of subtyping with input and output channels (as introduced in Chapter 7) is investigated, with an application towards the asynchronous π-calculus. Process equivalence in the presence of input and output channel types is then studied. In Chapter 11, a similar development is done for type systems capturing linearity, responsiveness, and polymorphism.

Part V, **The higher-order paradigm**, provides an in-depth study of the notion of mobility. As we saw in the introduction, mobility in the π-calculus is modeled by allowing channel names to be sent and received via channels. This is also called name-passing (or first-order). A more concrete approach to mobility is to allow the ability for processes to send and receive entire processes, an approach called process-passing (or higher-order). Mathematically, name-passing is much simpler than process-passing. On the other hand, process-passing is a more intuitive way to model mobility. It turns out that allowing process-passing does not add to the expressive power of the π-calculus, that is, anything expressible using process-passing is already expressible using name-passing. To make this formal, Chapter 12 introduces a higher-order typed π-calculus, HOπ, and develops its basic theory. The idea is to define the notion of a process abstraction that can be communicated across channels. In Chapter 13, it is shown how to translate HOπ into the π-calculus in a satisfactory way. Intuitively, the communication of a process abstraction translates into the communication of access to that abstraction. The translation is such that it reflects and preserves the equivalence of processes: two terms in the higher-order language are equivalent if and only their respective translations are equivalent, using the appropriate notions of equivalence.

The last two parts of the book address the relevance of the π-calculus to the theory of programming languages. Part VI, **Functions as processes**, explores the relation between the λ-calculus, a standard calculus for modeling sequential computations, and the π-calculus. It turns out that it is possible to encode the λ-calculus in the π-calculus, essentially turning the functions of the λ-calculus into processes that accept a value on a preselected input channel (the parameter channel), and returns a value on a preselected output channel (the result channel). In Chapter 14, the relevant theory of the λ-calculus is reviewed. The reader is expected to have had prior exposure to this topic, as the treatment is fast. In Chapter 15, the basic encoding is presented, which essentially amounts to a transformation of λ-terms into continuation-passing style, where each function takes an extra functional argument (the continuation) to which the result of the function call is passed. Different encodings can be given, corresponding to the different reductions strategies possible for the λ-calculus (call-by-name, call-by-value, etc...). Chapter 16 does the same for the typed λ-calculus. Chapters 17 and 18 explore properties of a particular encoding, that of the untyped call-by-name λ-calculus. The property of interest is that of the equality relation induced on λ-terms when their respective encodings are behaviourally equivalent as processes (according to different notions of process equivalences, but mostly barbed congruence).

Part VII, **Objects and π-calculus**, develops the relationship between the π-calculus and object-oriented programming. The intuitive similarities between these may not be completely clear at first glance, until one describes object-oriented systems as made up of objects that interact by invoking each other's methods, a procedure akin to sending messages. In Chapter 19, this is made manifest by introducing a simple object-oriented programming language, OOL, and by showing how to give it a semantics by translation to the π-calculus. In Chapter 20, some formal properties of OOL are examined, illustrating the use of π-calculus techniques in reasoning about objects. For example, one can check the correctness of certain program transformations, or the implementation of objects by separating code shared between instances of an object and data private to each instance.

Opinion

The basic recommendation I have is that anyone with a technical interest in the π-calculus needs this book. It brings together much of what is know about the calculus, information that for the most part can only be found in technical research articles. As such, it should remain the *de facto* reference work on the π-calculus for a great many years to come. I will even go as far as predicting that it will play the same role for the π-calculus that Barendregt's seminal book [1] plays for the λ-calculus.

It should be emphasized, however, that this is a reference book, not a textbook. It requires a good level of mathematical maturity, and furthermore, it requires prior exposure to the problems intrinsic to modeling concurrent systems, as well as the motivation underlying the process calculi approaches to those problems. Because of this, prior to reading the book, the neophyte should really first read Milner's own introduction to the π-calculus (in fact, to process calculi in general, and CCS in particular). Milner's original book on concurrency [7] is also suitable as an introduction to the material.

The need for a good reference for the basic theory of the π-calculus is clear when one looks at current work based on process calculi. Systems based on or inspired by the π-calculus are being used to study various aspects of security, for instance. The Ambient Calculus of Cardelli and Gordon [2] aims at modeling and reasoning about the notion of a locale in which computations execute, and in and out of which computations can move; the Spi-calculus of Gordon and Abadi [3] aims at modeling and reasoning about cryptographic protocols; the type system for processes developed by Honda et al. [5] aims at restricting statically the kind of information flowing from processes deemed high-security to processes deemed low-security.

References

[1] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. Studies in Logic. North-Holland, Amsterdam, 1981.

[2] L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

[3] A. D. Gordon and M. Abadi. A calculus for cryptographic protocols: The Spi calculus. In *4th ACM Conference on Computer and Communications Security*, 1997.

[4] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[5] K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 180–199. Springer-Verlag, 2000.

[6] R. Milner. *A Calculus of Communicating Systems*. Number 92 in Lecture Notes in Computer Science. Springer-Verlag, 1980.

[7] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[8] R. Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999.

[9] D. Sangiorgi. *Expressing mobility in process algebras: first-order and higher-order paradigms*. PhD thesis, Department of Computer Science, University of Edinburgh, 1993. CST-99-93, also published as ECS-LFCS-93-266.