FAQ on π-Calculus

Jeannette M. Wing
Visiting Researcher, Microsoft Research
Professor of Computer Science, Carnegie Mellon University

27 December 2002

1. What is π-calculus?

π -calculus is a model of computation for concurrent systems.

The syntax of π-calculus lets you represent processes, parallel composition of processes, synchronous communication between processes through channels, creation of fresh channels, replication of processes, and nondeterminism. That's it!

2. What do you mean by process? By channel?

A *process* is an abstraction of an independent thread of control. A *channel* is an abstraction of the communication link between two processes. Processes interact with each other by sending and receiving messages over channels.

3. Could you be a little more concrete?

Let P and Q denote processes. Then

- P | Q denotes a process composed of P and Q running in parallel.
- a(x).P denotes a process that waits to read a value x from the channel a and then, having received it, behaves like P.
- $\bar{a}\langle x\rangle$.P denotes a process that first waits to send the value x along the channel a and then, after x has been accepted by some input process, behaves like P.
- (νa)P ensures that a is a fresh channel in P. (Read the Greek letter "nu" as "new.")
- !P denotes an infinite number of copies of P, all running in parallel.
- P + Q denotes a process that behaves like either P or Q.
- **0** denotes the inert process that does nothing.

All concurrent behavior that you can imagine would have to be written in terms of just the above constructs.

4. How about an example? (Impatient readers should feel free to skip this question.)

Suppose you want to model a remote procedure call between a client and a server.

Consider the following function, `incr`, running on the server. `incr` returns the integer one greater than its argument, x:

```
int incr(int x) { return x+1; }
```

First, we model the "incr" server as a process in π-calculus as follows:

$$!\text{incr}(a, x).\bar{a}\langle x+1\rangle$$

Ignoring the ! for now, this process expression says that the incr channel accepts two inputs: one is the name of the channel, a, which we will use to return the result of calling incr, and the other is the argument, x, which will be instantiated with an integer value upon a client call. After the call, the process will send back the result of incrementing its argument, x, on the channel a. The use of the replication operator, !, in the above process expression means that the "incr" server will happily make multiple copies of itself, one for each client interaction.

Now let's model a client call to the "incr" server. In the following assignment statement, the result of calling `incr` with 17 gets bound to the integer variable y:

```
y := incr(17)
```

and would look like this in π-calculus:

$$(\nu a)(\ \overline{\text{incr}}\langle a, 17\rangle \mid a(y)\ )$$

which says in parallel: (1) send on the incr channel both the channel a (for passing back the result value) and the integer value 17, and (2) receive on the channel a the result y. The use of the ν operator guarantees that a private channel of communication is set up for each client interaction with the "incr" server.

Putting the client and server processes in parallel together we get the final process expression:

$$!\text{incr}(a, x).\bar{a}\langle x+1\rangle \mid\ (\nu a)(\ \overline{\text{incr}}\langle a, 17\rangle \mid a(y)\ )$$

which expresses the client call to the "incr" server with the argument 17 and the assignment of the returned value 18 to y.

5. What is the analogy between π-calculus and λ-calculus?

λ-calculus is to sequential programs as π-calculus is to concurrent programs.

More precisely, λ-calculus is the core language of functional computation, in which "everything is a function" and all computation proceeds by function application; π-calculus is the core calculus of message-based concurrency, in which "everything is a process" and all computation proceeds by communication on channels. λ-calculus can claim to be a *canonical* model of functional computation; however, π-calculus cannot make such a claim for concurrent computation [Pierce95].

Benjamin Pierce puts it best:

> The lambda-calculus holds an enviable position: it is recognized as embodying, in miniature, all of the essential features of functional computation. Moreover, other foundations for functional computation, such as Turing machines, have exactly the same expressive power. The "inevitability" of the lambda-calculus arises from the fact that the only way to observe a functional computation is to watch which output values it yields when presented with different input values.
>
> Unfortunately, the world of concurrent computation is not so orderly. Different notions of what can be observed may be appropriate for different circumstances, giving rise to different definitions of when two concurrent systems have "the same behavior": for example, we may wish to observe or ignore the degree of inherent parallelism of a system, the circumstances under which it may deadlock, the distribution of its processes among physical processors, or its resilience to various kinds of failures. Moreover, concurrent systems can be described in terms of many different constructs for creating processes (fork/wait, cobegin/coend, futures, data parallelism, etc.), exchanging information between them (shared memory, rendezvous, message-passing, dataflow, etc.), and managing their use of shared resources (semaphores, monitors, transactions, etc.).
>
> This variability has given rise to a large class of formal systems called **process calculi** (sometimes **process algebras**), each embodying the essence of a particular concurrent or distributed programming paradigm [Pierce 95].

π-calculus is just one of many such process calculi.

An interesting aside: λ-calculus can be encoded in π-calculus.

6. Why is the term "process algebra" sometimes used? What is a process algebra?

An algebra is a mathematical structure with a set of values and a set of operations on the values. These operations enjoy algebraic properties such as commutativity, associativity, idempotency, and distributivity. In a typical process algebra, processes are values and parallel composition is defined to be a commutative and associative operation on processes.

7. What's the difference between π-calculus and its predecessor process calculi?

What distinguishes π–calculus from earlier process calculi—in particular Robin Milner's own work on Calculus of Communicating Systems (CCS) [Milner80] and Tony Hoare's similar work on Communicating Sequential Processes (CSP) [Hoare85]—is the ability to pass channels as data along other channels. This feature allows you to express process

mobility, which in turn allows you to express changes in process structure. For example, suppose you're talking on your cell phone while driving in your car; the ability to model process mobility is useful for describing how your phone communicates with different base stations along the way.

8. Can you program in π-calculus?

Yes, but you wouldn't want to.

Just as λ-calculus can be viewed as an assembly language for sequential programs, so can π-calculus for concurrent programs. Both are simple, flexible, efficiently implementable, and suitable targets for compilation of higher-level language features [Pierce95]. But both would be impractical to use as a source-level programming language. For example, if your concurrent programming model is based on shared memory, as is the case with all Threads libraries, then encoding each globally shared variable in terms of channels would be cumbersome.

Rather, π-calculus is best viewed as a formal framework for providing the underlying semantics for a high-level concurrent or distributed programming language. For example, Jim Larus, Sriram Rajamani, and Jakob Rehof give the semantics of their new Sharpie language, for asynchronous programming, in terms of π-calculus [LRR02].

Older languages that define their semantics in terms of π-calculus or other process calculi include Pict [PierceTurner97], Amber [Cardelli86], Concurrent ML [Reppy91], and Facile [Amadio94]. Notably, even though these languages are higher-level than π-calculus, none of them are used in practice. (In contrast, in the sequential programming world, Scheme, ML, and Haskell are three popular functional programming languages whose semantics build from λ-calculus—all three are used widely in teaching and in research.)

9. What good is π-calculus as a practical tool?

π-calculus, or any process calculus for that matter, is good as a modeling language.

In particular, people have successfully used process calculi to model protocols. Protocols, after all, describe ways in which processes should communicate with each other. People have used process calculi to model and verify telecommunications protocols, often with support from model checking tools such as the Concurrency Workbench [CPS93] and FDR [FDR93]. Others have used a π-calculus variant, called Spi-calculus, to reason about cryptographic protocols [AbadiGordon97].

As with using any modeling language, there still remains a huge gap between the model and the code, i.e., between the specification of desired behavior and the program that implements it. Either you refine your model till you spit out "correct" code or you write code and prove its "correctness" against your model. Both approaches are used, with

varying degrees of automated support, to varying degrees of success. Attacking this gap is still an active area of research.

To illustrate how large this gap can be, consider again the modeling of remote procedure call in Question 4. An implementation of RPC would have to consider details such as marshalling and unmarshalling arguments and results, handling failures such as links or servers going down, synchronizing clocks, locating the server in the first place, and so on. Clearly, at a level of understanding a high-level protocol, where you are interested in only the interactions between various clients and servers (e.g., users making reservations through an on-line travel agency), you want to ignore that level of detail. Here, π-calculus as a modeling language is a win. But, eventually you have to implement not just RPC, but also the application-specific protocol (e.g., travelers should not double book). How to guarantee you have correctly implemented your communications protocol such as RPC as well as your application-specific protocol is still a hard problem.

10. What does π-calculus, or more generally process calculi, have to do with software engineering?

One place where process calculi have recently played a role in software engineering research is in the field of software architecture. A software architecture describes a system's *components* (e.g., clients and servers, peer processes, distributed agents) and *connectors* (e.g., remote procedure call, event broadcast, publish-subscribe, Unix pipes). A connector describes the protocol by which components communicate. Rob Allen and David Garlan define a software architecture language called Wright [AllenGarlan97], suitable for defining different kinds of connectors, i.e., different ways in which components can interact. They give the semantics of Wright in terms of CSP [Hoare95], a process algebra in the same family as CCS, the predecessor of π-calculus. Moreover, they use the FDR model checker [FDR93], whose semantics are also given in terms of CSP, to find bugs in architecture-level descriptions of software systems.

11. What good is π-calculus as a formalism?

π-calculus, as with any formalism, can help answer fundamental questions about seemingly disparate issues, all within a single framework. For example, consider synchronous versus asynchronous communication. Making a telephone call is an example of synchronous communication; sending e-mail is an example of asynchronous communication.

Synchronous π-calculus requires that senders and receivers rendezvous when communicating. Asynchrony is inherent in distributed systems, because it takes time for a message to travel from one machine to another. Thus, many researchers use asynchronous π-calculus, which is a fragment of synchronous π-calculus, as their model of concurrency, especially for modeling distributed computation.

A question that naturally arises is whether these two mechanisms are equivalent, i.e., can one implement the other? It is well known how to implement asynchronous

communication in terms of synchronous. But what about the other direction? Catuscia Palamidessi uses the π-calculus as a uniform framework to answer this question in the negative: synchronous communication is more powerful [Palamidessi02].

12. Why are there so many formalisms such as π-calculus and Abstract State Machines (ASM)?

There are many classes of formalisms. They differ in the types of systems they are suitable for describing (e.g., hardware, real-time, programs, protocols), in the types of properties they can be used to prove (e.g., correctness, deadlock freedom, timing, liveness), in their underlying logical power (e.g., first-order, higher-order), in their underlying semantic model (e.g., type of algebraic structure), in their usability (e.g., expressive power, tool support, scalability), etc.

π-calculus is a natural choice for describing concurrent processes that communicate through message passing. It is not a natural choice for describing abstract data types. It is not a natural choice for describing states with rich or complex data structures.

ASM is a natural choice for describing abstract states and single-step transitions that change state. Its model of concurrency is based on abstract shared global state. See Yuri Gurevich's web site http://research.microsoft.com/~gurevich/annotated.html for ASM-related papers, including the Lipari 1993 guide (#103) that defines sequential, parallel, and distributed ASMs in terms of *evolving algebras* [Gurevich93].

See http://archive.comlab.ox.ac.uk/formal-methods.html for a wealth of information about other formalisms: their notations, methods, and tools. As of today, π-calculus and ASM are two of the 92 listed.

13. Where can I read more about π-calculus?

Standard references on π-calculus are Milner's tutorial "The Polyadic π-calculus" [Milner91]; Milner, Parrow, and Walker's two-part article "A Calculus of Mobile Processes" [MPW92]; and Milner's book, *Communicating and Mobile Systems: the π-calculus* [Milner99]. Sangiorgi and Walker's graduate-level textbook, *The π-calculus: A Theory of Mobile Processes* [SangiorgiWalker01], gives a detailed encoding of λ-calculus in π-calculus and shows how to do object-oriented programming in π-calculus.

Benjamin Pierce gives an excellent introduction [Pierce95] to both λ-calculus and π-calculus.

See http://liinwww.ira.uka.de/bibliography/Theory/pi.html for a bibliography on calculi for mobile processes.

Acknowledgments

I thank Jim Larus, Dan Ling, and Jim Kajiya for their general comments. Thanks to Jim Larus for his specific suggestion to add Question 2, and to Jakob Rehof and Sriram Rajamani for their help with the example for Question 4. Thanks to Yuri Gurevich for clarifying my understanding of ASMs and for his helpful pointers. I took most of the answer in Question 5 from Benjamin Pierce's *CRC Handbook* article [Pierce95].

References

[AbadiGordon97] Martin Abadi and Andrew Gordon, Reasoning about Cryptographic Protocols in the Spi Calculus, *CONCUR'97: Concurrency Theory*, Lecture Notes in Computer Science, volume 1243, Springer-Verlag, July 1997, pp. 59-73. http://research.microsoft.com/~adg/Publications/details.htm

[AllenGarlan97] Robert Allen and David Garlan, A Formal Basis for Architectural Connection, *ACM Trans. on Soft. Eng. and Methodology*, July 1997. http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/able/www/paper_abstracts/wright-tosem97.html

[Amadio94] Roberto M. Amadio, Translating core Facile, Technical Report ECRC-TR-3-94, European Computer-Industry Research Center, GmbH, Munich, 1994.

[Cardelli86] Luca Cardelli, Amber, *Combinators and Functional Programming Languages*, Lecture Notes in Computer Science, volume 242, Springer-Verlag, 1986, pp. 21-47.

[CPS93] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen, The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems, *ACM Trans. on Prog. Lang. and Systems*, 15(1): 36-72, January 1993. http://www.cs.sunysb.edu/~rance/publications/./1993.html

[FDR93] Formal Systems (Europe) Ltd., *Failures-Divergences-Refinement*, User Manual and Tutorial, 1993.

[Gurevich93] Yuri Gurevich, Evolving Algebras 1993: Lipari Guide, in *Specification and Validation Methods*, ed. E. Börger, Oxford University Press, 1995, pp. 9-36. http://research.microsoft.com/~gurevich/Opera/103.pdf

[Hoare85] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[LLR02] Jim Larus, Sriram Rajamani, and Jakob Rehof, Behavioral Types of Asynchronous Programming, Microsoft Research Technical Report, November 2002, submitted to PLDI. http://www.research.microsoft.com/behave/sharpie-report-abs.html

[Milner80] Robin Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, volume 92, Springer-Verlag, 1980.

[Milner91] Robin Milner, The Polyadic π-calculus, Technical Report ECS-LFCS-91-180, Lab. for Foundations of Computer Science, Dept. of Computer Science, University of Edinburgh, UK, October 1991.
http://www.lfcs.informatics.ed.ac.uk/reports/91/ECS-LFCS-91-180/

[Milner99] Robin Milner, *Communicating and mobile systems: the π-calculus*, Cambridge University Press, 1999.

[MPW92] Robin Milner, Joachim Parrow, and David Walker, A Calculus of Mobile Processes, Parts I and II, *Information and Computation*, 100(1): 1-40 and 41-77.
http://www.lfcs.informatics.ed.ac.uk/reports/89/ECS-LFCS-89-85/
http://www.lfcs.informatics.ed.ac.uk/reports/89/ECS-LFCS-89-86/

[Palamidessi02] Catuscia Palamidessi, Comparing the Expressive Power of the Synchronous and the Asynchronous π-calculi, to appear in *Math. Struct. in Comp. Sci.*, 2002.
http://www.cse.psu.edu/~catuscia/papers/pi_calc/mscs.ps

[Pierce95] Benjamin C. Pierce, Foundational Calculi for Programming Languages, *CRC Handbook of Computer Science and Engineering*, Chapter 136, CRC Press, 1996.
http://www.cis.upenn.edu/~bcpierce/papers/

[PierceTurner97] Benjamin C. Pierce and David N. Turner, Pict: A Programming Language Based on the Pi-Calculus, Indiana University CSCI Technical Report #476, 1997.
http://www.cis.upenn.edu/~bcpierce/papers/

[Reppy91] John Reppy, CML: A higher-order concurrent language, *Proc. of Programming Language Design and Implementation*, ACM SIGPLAN, June 1991, pp. 293-259.

[SangiorgiWalker01] Davide Sangiorgi and David Walker, *The π-calculus: a Theory of Mobile Processes*, Cambridge University Press, 2001.