π-calculus

In theoretical computer science, the **π-calculus** (or **pi-calculus**) is a process calculus originally developed by Robin Milner, Joachim Parrow [1] and David Walker, based on ideas by Uffe Engberg and Mogens Nielsen. It can be seen as a continuation of Milner's work on the process calculus CCS (Calculus of Communicating Systems). The π-calculus allows channel names to be communicated along the channels themselves, and in this way it is able to describe concurrent computations whose network configuration may change during the computation.

The π-calculus is elegantly simple yet very expressive. Functional programs can be encoded into the π-calculus, and the encoding emphasises the dialogue nature of computation, drawing connections with game semantics. Extensions of the π-calculus, such as the spi calculus and applied π, have been successful in reasoning about cryptographic protocols. Beside the original use in describing concurrent systems, the π-calculus has also been used to reason about business processes and molecular biology.

Informal definition

The π-calculus belongs to the family of process calculi, mathematical formalisms for describing and analyzing properties of concurrent computation. In fact, the π-calculus, like the λ-calculus, is so minimal that it does not contain primitives such as numbers, booleans, data structures, variables, functions, or even the usual flow control statements (such as `if-then-else`, `while`).

Process constructs

Central to the π-calculus is the notion of *name*. The simplicity of the calculus lies in the dual role that names play as *communication channels* and *variables*.

The process constructs available in the calculus are the following (a precise definition is given in the following section):

- *concurrency*, written $P \mid Q$, where P and Q are two processes or threads executed concurrently.
- *communication*, where
  - *input prefixing* $c(x).P$ is a process waiting for a message that was sent on a communication channel named c before proceeding as P, binding the name received to the name x. Typically, this models either a process expecting a communication from the network or a label `c` usable only once by a `goto c` operation.
  - *output prefixing* $\overline{c}\langle y \rangle.P$ describes that the name y is emitted on channel c before proceeding as P. Typically, this models either sending a message on the network or a `goto c` operation.
- *replication*, written $!P$, which may be seen as a process which can always create a new copy of P. Typically, this models either a network service or a label `c` waiting for any number of `goto c` operations.
- *creation of a new name*, written $(\nu x)\,P$, which may be seen as a process allocating a new constant x within P. The constants of π-calculus are defined by their names only and are always communication channels. Creation of a new name in a process is also called *restriction*.
- the nil process, written *0*, is a process whose execution is complete and has stopped.

Although the minimalism of the π-calculus prevents us from writing programs in the normal sense, it is easy to extend the calculus. In particular, it is easy to define both control structures such as recursion, loops and sequential composition and datatypes such as first-order functions, truth values, lists and integers. Moreover, extensions of the π-calculus have been proposed which take into account distribution or public-key cryptography. The *applied π-calculus* due to Abadi and Fournet [2] put these various extensions on a formal footing by extending the π-calculus with arbitrary datatypes.

A small example

Below is a tiny example of a process which consists of three parallel components. The channel name x is only known by the first two components.

$$\begin{aligned}&(\nu x)\ (\ \overline{x}\langle z\rangle.\ 0\\&\qquad\quad |\ x(y).\ \overline{y}\langle x\rangle.\ x(y).\ 0\ )\\&|\ z(v).\ \overline{v}\langle v\rangle.0\end{aligned}$$

The first two components are able to communicate on the channel x, and the name y becomes bound to z. The next step in the process is therefore

$$\begin{aligned}&(\nu x)\ (\ 0\\&\qquad\quad |\ \overline{z}\langle x\rangle.\ x(y).\ 0\ )\\&|\ z(v).\ \overline{v}\langle v\rangle.\ 0\end{aligned}$$

Note that the remaining y is not affected because it is defined in an inner scope. The second and third parallel components can now communicate on the channel name z, and the name v becomes bound to x. The next step in the process is now

$$\begin{aligned}&(\nu x)(\ 0\\&\qquad |\ x(y).\ 0\\&\qquad |\ \overline{x}\langle x\rangle.\ 0\ )\end{aligned}$$

Note that since the local name x has been output, the scope of x is extended to cover the third component as well. Finally, the channel x can be used for sending the name x.

Formal definition

Syntax

Let X be a set of objects called *names*. The abstract syntax for the π-calculus is built from the following BNF grammar (where *x* and *y* are any names from X):[3]

$$\begin{aligned}P ::= &\ x(y).P\\ |\ &\ \overline{x}\langle y\rangle.P\\ |\ &\ P|P\\ |\ &\ (\nu x)P\\ |\ &\ !P\\ |\ &\ 0\end{aligned}$$

In the concrete syntax below, the prefixes bind more tightly than the parallel composition (|), and parentheses are used to disambiguate.

Names are bound by the restriction and input prefix constructs. Formally, the sets of free and bound names of a process in π–calculus are defined inductively as follows.

- The 0 process has no free names and no bound names.
- The free names of $\overline{a}\langle x\rangle.P$ are a, x, and the free names of P. The bound names of $\overline{a}\langle x\rangle.P$ are the bound names of P.
- The free names of $a(x).P$ are a and the free names of P, except for x. The bound names of $a(x).P$ are x and the bound names of P.
- The free names of $P|Q$ are those of P together with those of Q. The bound names of $P|Q$ are those of P together with those of Q.

- The free names of $(\nu x).P$ are those of P, except for x. The bound names of $(\nu x).P$ are x and the bound names of P.
- The free names of $!P$ are those of P. The bound names of $!P$ are those of P.

Structural congruence

Central to both the reduction semantics and the labelled transition semantics is the notion of **structural congruence**. Two processes are structurally congruent, if they are identical up to structure. In particular, parallel composition is commutative and associative.

More precisely, structural congruence is defined as the least equivalence relation preserved by the process constructs and satisfying:

Alpha-conversion:

- $P \equiv Q$ if Q can be obtained from P by renaming one or more bound names in P.

Axioms for parallel composition:

- $P|Q \equiv Q|P$
- $(P|Q)|R \equiv P|(Q|R)$
- $P|0 \equiv P$

Axioms for restriction:

- $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$
- $(\nu x)0 \equiv 0$

Axiom for replication:

- $!P \equiv P|!P$

Axiom relating restriction and parallel:

- $(\nu x)(P|Q) \equiv (\nu x)P|Q$ if x is not a free name of Q.

This last axiom is known as the "scope extension" axiom. This axiom is central, since it describes how a bound name x may be extruded by an output action, causing the scope of x to be extended. In cases where x is a free name of Q, alpha-conversion may be used to allow extension to proceed.

Reduction semantics

We write $P \rightarrow P'$ if P can perform a computation step, following which it is now P'. This *reduction relation* $\rightarrow$ is defined as the least relation closed under a set of reduction rules.

The main reduction rule which captures the ability of processes to communicate through channels is the following:

- $\overline{x}\langle z\rangle.P|x(y).Q \rightarrow P|Q[z/y]$

  where $Q[z/y]$ denotes the process Q in which the free name z has been *substituted* for the free occurrences of y. If a free occurrence of y occurs in a location where z would not be free, alpha-conversion may be required.

There are three additional rules:

- If $P \rightarrow Q$ then also $P|R \rightarrow Q|R$.

  This rule says that parallel composition does not inhibit computation.

- If $P \rightarrow Q$, then also $(\nu x)P \rightarrow (\nu x)Q$.

  This rule ensures that computation can proceed underneath a restriction.

- If $P \equiv P'$ and $P' \rightarrow Q'$ where $Q' \equiv Q$, then also $P \rightarrow Q$.

The latter rule states that processes that are structurally congruent have the same reductions.

The example revisited

Consider again the process

$$(\nu x)(\overline{x}\langle z\rangle.0|x(y).\overline{y}\langle x\rangle.x(y).0)|z(v).\overline{v}\langle v\rangle.0$$

Applying the definition of the reduction semantics, we get the reduction

$$(\nu x)(\overline{x}\langle z\rangle.0|x(y).\overline{y}\langle x\rangle.x(y).0)|z(v).\overline{v}\langle v\rangle.0 \rightarrow (\nu x)(0|\overline{z}\langle x\rangle.x(y).0)|z(v).\overline{v}\langle v\rangle.0$$

Note how, applying the reduction substitution axiom, occurrences of y are now labeled as z.

Next, we get the reduction

$$(\nu x)(0|\overline{z}\langle x\rangle.x(y).0)|z(v).\overline{v}\langle v\rangle.0 \rightarrow (\nu x)(0|x(y).0|\overline{x}\langle x\rangle.0)$$

Note that since the local name x has been output, the scope of x is extended to cover the third component as well. This was captured using the scope extension axiom.

Labelled semantics

Alternatively, one may give the pi-calculus a labelled transition semantics (as has been done with the Calculus of Communicating Systems). Transitions in this semantics are of the form:

$$P \xrightarrow{\alpha} P'$$

This notation signifies that P after the action α becomes P'. α can be an *input action* $a(x)$, an *output action* $\overline{a}\langle x\rangle$, or a tau-action τ corresponding to an internal communication.
A standard result about the labelled semantics is that it agrees with the reduction semantics in the sense that $P \rightarrow P'$ if and only if $P \xrightarrow{\tau} P'$ for some action τ.

Extensions and variants

The syntax given above is a minimal one. However, the syntax may be modified in various ways.

A *nondeterministic choice operator* $P + Q$ can be added to the syntax.

A test for *name equality* $[x = y]P$ can be added to the syntax. This *match operator* can proceed as P if and only if x and y are the same name. Similarly, one may add a *mismatch operator* for **name inequality**. Practical programs which can pass names (URLs or pointers) often use such functionality: for directly modelling such functionality inside the calculus, this and related extensions are often useful.
The *asynchronous π-calculus* allows only outputs with no continuation, i.e. output atoms of the form $\overline{x}\langle y\rangle$, yielding a smaller calculus. However, any process in the original calculus can be represented by the smaller asynchronous π-calculus using an extra channel to simulate explicit acknowledgement from the receiving process. Since a continuation-free output can model a message-in-transit, this fragment shows that the original π-calculus, which is intuitively based on synchronous communication, has an expressive asynchronous communication model inside its syntax.
The *polyadic π-calculus* allows communicating more than one name in a single action: $\overline{x} < z_1, ...z_n > .P$ *(polyadic output)* and $x(z_1, ...z_n)$ *(polyadic input)*. This polyadic extension, which is useful especially when studying types for name passing processes, can be encoded in the monadic calculus by passing the name of a private channel through which the multiple arguments are then passed in sequence. The encoding is defined recursively by the clauses
$\overline{x}\langle y_1, \cdots, y_n\rangle.P$ is encoded as $(\nu w)\overline{x}\langle w\rangle.\overline{w}\langle y_1\rangle.\cdots.\overline{w}\langle y_n\rangle.[P]$
$x(y_1, \cdots, y_n).P$ is encoded as $x(w).w(y_1).\cdots.w(y_n).[P]$
All other process constructs are left unchanged by the encoding.

In the above, $[P]$ denotes the encoding of all prefixes in the continuation P in the same way.

The full power of replication $!P$ is not needed. Often, one only considers *replicated input* $!x(y).P$, whose structural congruence axiom is $!x(y).P \equiv x(y).P|!x(y).P$.
Replicated input process such as $!x(y).P$ can be understood as servers, waiting on channel x to be invoked by clients. Invocation of a server spawns a new copy of the process $P[a/y]$, where a is the name passed by the client to the server, during the latter's invocation.
A *higher order π-calculus* can be defined where not only names but processes are sent through channels. The key reduction rule for the higher order case is

$$\overline{x}\langle R\rangle.P|x(Y).Q \rightarrow P|Q[R/Y]$$

Here, Y denotes a *process variable* which can be instantiated by a process term. Sangiorgi established the surprising result that the ability to pass processes does not increase the expressivity of the π-calculus: passing a process *P* can be simulated by just passing a name that points to *P* instead.

Properties

Turing completeness

The π-calculus is a universal model of computation. This was first observed by Milner in his paper "Functions as Processes",[4] in which he presents two encodings of the lambda-calculus in the π-calculus. One encoding simulates the eager (call-by-value) evaluation strategy, the other encoding simulates the normal-order (call-by-name) strategy. In both of these, the crucial insight is the modeling of environment bindings – for instance, " x is bound to term M " – as replicating agents that respond to requests for their bindings by sending back a connection to the term M.

The features of the π-calculus that make these encodings possible are name-passing and replication (or, equivalently, recursively defined agents). In the absence of replication/recursion, the π-calculus ceases to be Turing-powerful. This can be seen by the fact that bisimulation equivalence becomes decidable for the recursion-free calculus and even for the finite-control π-calculus where the number of parallel components in any process is bounded by a constant.[5]

Bisimulations in the π-calculus

As for process calculi, the π-calculus allows for a definition of bisimulation equivalence. In the π-calculus, the definition of bisimulation equivalence (also known as bisimilarity) may be based on either the reduction semantics or on the labelled transition semantics.

There are (at least) three different ways of defining *labelled bisimulation equivalence* in the π-calculus: Early, late and open bisimilarity. This stems from the fact that the π-calculus is a value-passing process calculus.

In the remainder of this section, we let p and q denote processes and R denote binary relations over processes.

Early and late bisimilarity

Early and late bisimilarity were both discovered by Milner, Parrow and Walker in their original paper on the π-calculus.[6]

A binary relation R over processes is an *early bisimulation* if for every pair of processes $(p, q) \in R$,

- whenever $p \xrightarrow{a(x)} p'$ then for every name y there exists some q' such that $q \xrightarrow{a(x)} q'$ and $(p'[y/x], q'[y/x]) \in R$;
- for any non-input action α, if $p \xrightarrow{\alpha} p'$ then there exists some q' such that $q \xrightarrow{\alpha} q'$ and $(p', q') \in R$;
- and symmetric requirements with p and q interchanged.

Processes p and q are said to be early bisimilar, written $p \sim_e q$ if the pair $(p, q) \in R$ for some early bisimulation R.

In late bisimilarity, the transition match must be independent of the name being transmitted. A binary relation R over processes is a *late bisimulation* if for every pair of processes $(p, q) \in R$,

- whenever $p \xrightarrow{a(x)} p'$ then for some q' it holds that $q \xrightarrow{a(x)} q'$ and $(p'[y/x], q'[y/x]) \in R$ *for every name y*;
- for any non-input action α, if $p \xrightarrow{\alpha} p'$ implies that there exists some q' such that $q \xrightarrow{\alpha} q'$ and $(p', q') \in R$;
- and symmetric requirements with p and q interchanged.

Processes p and q are said to be late bisimilar, written $p \sim_l q$ if the pair $(p, q) \in R$ for some late bisimulation R.

Both $\sim_e$ and $\sim_l$ suffer from the problem that they are not *congruence relations* in the sense that they are not preserved by all process constructs. More precisely, there exist processes p and q such that $p \sim_e q$ but $a(x).p \not\sim_e a(x).q$. One may remedy this problem by considering the maximal congruence relations included in $\sim_e$ and $\sim_l$, known as *early congruence* and *late congruence*, respectively.

Open bisimilarity

Fortunately, a third definition is possible, which avoids this problem, namely that of *open bisimilarity*, due to Sangiorgi.[7]

A binary relation R over processes is an *open bisimulation* if for every pair of elements $(p, q) \in R$ and for every name substitution σ and every action α, whenever $p\sigma \xrightarrow{\alpha} p'$ then there exists some q' such that $q\sigma \xrightarrow{\alpha} q'$ and $(p', q') \in R$.

Processes p and q are said to be open bisimilar, written $p \sim_o q$ if the pair $(p, q) \in R$ for some open bisimulation R.

Early, late and open bisimilarity are in fact all distinct. The containments are proper, so $\sim_o \subsetneq \sim_l \subsetneq \sim_e$.

In certain subcalculi such as the asynchronous pi-calculus, late, early and open bisimilarity are known to coincide. However, in this setting a more appropriate notion is that of *asynchronous bisimilarity*.

The reader should note that, in the literature, the term *open bisimulation* usually refers to a more sophisticated notion, where processes and relations are indexed by distinction relations; details are in Sangiorgi's paper cited above.

Barbed equivalence

Alternatively, one may define bisimulation equivalence directly from the reduction semantics. We write $p \Downarrow a$ if process p immediately allows an input or an output on name a.

A binary relation R over processes is a *barbed bisimulation* if it is a symmetric relation which satisfies that for every pair of elements $(p, q) \in R$ we have that

(1) $p \Downarrow a$ if and only if $q \Downarrow a$ for every name a

and

(2) for every reduction $p \longrightarrow p'$ there exists a reduction $q \longrightarrow q'$

such that $(p', q') \in R$.

We say that p and q are *barbed bisimilar* if there exists a barbed bisimulation R where $(p, q) \in R$.

Defining a context as a π term with a hole [] we say that two processes P and Q are *barbed congruent*, written $P \sim_b Q$, if for every context $C[]$ we have that $C[P]$ and $C[Q]$ are barbed bisimilar. It turns out that barbed congruence coincides with the congruence induced by early bisimilarity.

Applications

The π-calculus has been used to describe many different kinds of concurrent systems. In fact, some of the most recent applications lie outside the realm of computer science.

In 1997, Martin Abadi and Andrew Gordon proposed an extension of the π-calculus, the Spi-calculus, as a formal notation for describing and reasoning about cryptographic protocols. The spi-calculus extends the π-calculus with primitives for encryption and decryption. In 2001, Martin Abadi and Cedric Fournet generalised the handling of cryptographic protocols to produce the applied π calculus. There is now a large body of work devoted to variants of the applied π calculus, including a number of experimental verification tools. One example is the tool ProVerif [8] due to Bruno Blanchet, based on a translation of the applied π-calculus into Blanchet's logic programming framework. Another example is Cryptyc [9], due to Andrew Gordon and Alan Jeffrey, which uses Woo and Lam's method of correspondence assertions as the basis for type systems that can check for authentication properties of cryptographic protocols.

Around 2002, Howard Smith and Peter Fingar became interested in using the π-calculus as a description tool for modelling business processes. As of July 2006, there is discussion in the community as to how useful this will be. Most recently, the π-calculus has been used as the theoretical basis of Business Process Modeling Language (BPML), and of Microsoft's XLANG.[10]

The π-calculus has also attracted interest in molecular biology. In 1999, Aviv Regev and Ehud Shapiro showed that one can describe a cellular signaling pathway (the so-called RTK/MAPK cascade) and in particular the molecular "lego" which implements these tasks of communication in an extension of the π-calculus.[11]Following this seminal paper, other authors described the whole metabolic network of a minimal cell.[12]

Implementations

The following programming languages are implementations either of the π-calculus or of its variants:

- Acute
- Business Process Modeling Language (BPML)
- Nomadic Pict
- jumel [13] : a compiler for a dialect of ML augmented with pi-calculus primitives for message-passing
- occam-π
- Pict
- JoCaml (based on the Join-calculus)
- Funnel (A JRE-compatible join calculus implementation)
- The CubeVM [14] (a stackless implementation)
- The SpiCO [15] language: a stochastic pi-calulus for concurrent objects
- BioSPI [16] and SPiM [17]: simulators for the stochastic pi-calculus
- Ateji PX [18]: a Java language extension with parallel primitives inspired from π-calculus

Notes

[1] http://user.it.uu.se/~joachim/

[2] http://citeseer.ist.psu.edu/rd/0%2C573109%2C1%2C0.25%2CDownload/http%3AqSqqSqwww.cse.ucsc.eduqSq%7EabadiqSqPapersqSqisss02.pdf

[3] A Calculus of Mobile Processes part 1 (http://www.lfcs.inf.ed.ac.uk/reports/89/ECS-LFCS-89-85/) page 10, by R. Milner, J. Parrow and D. Walker published in Information and Computation 100(1) pp.1-40, Sept 1992

[4] Milner, Robin (1992). "Functions as Processes". *Mathematical Structures in Computer Science* **2**: 119–141.

[5] Dam, Mads (1997). "On the Decidability of Process Equivalences for the pi-Calculus". *Theoretical Computer Science* **183** (183): 215–228. doi:10.1016/S0304-3975(96)00325-8.

[6] Milner, R.; J. Parrow and D. Walker (1992). "A calculus of mobile processes". *Information and Computation* **100** (100): 1–40. doi:10.1016/0890-5401(92)90008-4.

[7] Sangiorgi, D. (1996). "A theory of bisimulation for the π-calculus". *Acta Informatica* **33**: 69–97. doi:10.1007/s002360050036.

[8] http://www.proverif.ens.fr/

[9] http://www.cryptyc.org

[10] "BPML | BPEL4WS: A Convergence Path toward a Standard BPM Stack." BPMI.org Position Paper. August 15, 2002. (http://www.bpmi.org/downloads/BPML-BPEL4WS.pdf)

[11] Regev, Aviv; William Silverman and Ehud Y. Shapiro (2001). "Representation and Simulation of Biochemical Processes Using the pi-Calculus Process Algebra". *Pacific Symposium on Biocomputing*: 459–470.

[12] Chiarugi, Davide; Pierpaolo Degano and Roberto Marangoni (2007). "A computational approach to the functional screening of genomes" (http://www.ploscompbiol.org/article/info:doi/10.1371/journal.pcbi.0030174). *PLOS Computational Biology*: 1801–1806. .

[13] http://graal.ens-lyon.fr/jumel

[14] http://www-poleia.lip6.fr/~pesch/cube/about.html

[15] http://spico.gforge.inria.fr/

[16] http://www.wisdom.weizmann.ac.il/~biospi/

[17] http://research.microsoft.com/~aphillip/spim/

[18] http://www.ateji.com/px

References

- Milner, Robin (1999). *Communicating and Mobile Systems: The π-calculus*. Cambridge, UK: Cambridge University Press. ISBN 0-521-65869-1.
- Milner, Robin (1993). "The Polyadic π-Calculus: A Tutorial" (http://www.lfcs.inf.ed.ac.uk/reports/91/ECS-LFCS-91-180/ECS-LFCS-91-180.ps). In F. L. Hamer, W. Brauer, H. Schwichtenberg. *Logic and Algebra of Specification*. Springer-Verlag.
- Sangiorgi, Davide; Walker, David (2001). *The π-calculus: A Theory of Mobile Processes*. Cambridge, UK: Cambridge University Press. ISBN 0-521-78177-9.

External links

- PiCalculus (http://c2.com/cgi/wiki?PiCalculus) on the C2 wiki
- Calculi for Mobile Processes (http://move.to/mobility)
- FAQ on π-Calculus (http://www.eecs.harvard.edu/~nr/cs257/archive/jeannette-wing/pi.pdf) by Jeannette M. Wing

Article Sources and Contributors

π-calculus *Source*: https://en.wikipedia.org/w/index.php?oldid=518687855 *Contributors*: 2001:660:3301:8061:7010:B15B:FEF2:186E, A.bit, AKappa, Adam Sampson, Adrianwn, Allan McInnes, Anthony Appleyard, Ashley Y, Bah23, CRGreathouse, CSTAR, CarlHewitt, Chabbrik, Charles Matthews, Clements, ComputScientist, Cybercobra, D. Wu, Dcoetzee, Diabloblue, Dvchiaru, E-boy, Edward, Erik Zachte, Eruionnyron, Euicho, Everyking, Fronx, Functor salad, Gareth Jones, Gregbard, Hairy Dude, HansHuttel, Headbomb, Indil, Jackie, Jhwoodyatt, Jpbowen, Jsnx, Kadellar, Kosik, Lambda kid, Lmkaff, Malcohol, MarSch, MathMartin, MeltBanana, MrReaper, Naasking, Neptunius, Oerjan, Orz, Papppfaffe, Peak, RainbowOfLight, Repton, Risacher, Rjwilmsi, Ruud Koot, Sam Staton, Sergiu.dumitriu, Siddhant, Smimram, Tagtool, Taiga234, Tettix, Tigerforce, Tom Minka, Urushiol, Xyzzy n, Yoric, 162 anonymous edits

License