Reasoning about Concurrent Systems Using Types

Davide Sangiorgi

INRIA – Sophia Antipolis, France

The purpose of these notes is to discuss some examples of the importance of types for reasoning about concurrent systems, and to list some relevant references. The list is surely not meant to be exhaustive, as the area is broad and very active. The examples are presented in the *π-calculus* [29], a paradigmatical process calculus for message-passing concurrency. We will not describe the proof techniques based on types with which the equalities in the examples are actually proved; for this, the interested reader can follow the references.

Acknowledgements. I would like to thank Benjamin Pierce, for our collaborations and the numerous discussions on the topic of these notes.

The π-calculus. As the λ-calculus, so the π-calculus language consists of a small set of primitive constructs. In the λ-calculus, they are constructs for building functions. In the π-calculus, they are constructs for building processes, notably: composition $P \mid Q$ to run two processes in parallel; restriction $\boldsymbol{\nu} x\, P$ to localise the scope of name x to process P (name is a synonymous for channels); input $x(y).\, P$ to receive a name z at x and then to continue as $P\{^z/_y\}$; output $\overline{x}\langle y\rangle.\, P$ to emit name y at x and then to continue as P; replication $!P$ to express processes with an infinite behaviour ($!P$ stands for a countable infinite number of copies of P in parallel); the inactive process $\mathbf{0}$. In the pure (i.e., untyped) calculus, all values transmitted are names.

We will find it convenient to present some of the examples on the *polyadic π-calculus*, an extension of the pure calculus in which tuples of names may be transmitted. A polyadic input process $x(y_1, \ldots, y_n).\, P$ waits for an n-uple of names $z_1, \ldots, z_n$ at x and then continues as $P\{^{z_1, \ldots, z_n}/_{y_1, \ldots, y_n}\}$ (that is, P with the y_i's replaced by the z_i's); a polyadic ouput process $\overline{x}\langle y_1, \ldots, y_n\rangle.\, P$ emits names $y_1, \ldots, y_n$ at x and then continues as P. We will abbreviate processes of the form $\overline{x}\langle y_1, \ldots, y_n\rangle.\, \mathbf{0}$ as $\overline{x}\langle y_1, \ldots, y_n\rangle$.

The most important predecessor of the π-calculus is CCS. The main novelty of the π-calculus over CCS is that names themselves may be communicated. This gives π-calculus a much greater expressiveness. We can encode, for instance: data values, the λ-calculus, higher-order process calculi (i.e., calculi where terms of the language can be exchanged) [27, 28, 42], which indicates that the π-calculus can be a model of languages incorporating functional and concurrent features, and that it may be a foundation for the design of new programming languages; the spatial dependencies among processes [39], which indicates that the π-calculus can be a model of languages for distributed computing; (some) object-oriented languages [21, 52, 22, 20].

Types. A type system is, roughly, a mechanism for classifying the expressions of a program. Type systems are useful for several reasons: to perform optimisa-

W. Thomas (Ed.): FOSSACS'99, LNCS 1578, pp. 31-40, 1999.

tions in compilers; to detect simple kinds of programming errors at compilation time; to aid the structure and design of systems; to extract behavioral information that can be used for *reasoning* about programs. In sequential programming languages, type systems are widely used and generally well-understood. In concurrent programming languages, by contrast, the tradition of type systems is much less established.

In the π-calculus world, types have quickly emerged as an important part of its theory and of its applications, and as one of the most important differences with respect to CCS-like languages. The types that have been proposed for the π-calculus are often inspired by well-known type systems of sequential languages, especially λ-calculi. Also type systems specific to processes have been (and are being) investigated, for instance for preventing certain forms of interferences among processes or certain forms of deadlocks.

One of the main reasons for which types are important for reasoning on π-calculus processes is the following. Although well-developed, the theory of the pure π-calculus is often insufficient to prove "expected" properties of processes. This because a π-calculus programmer normally uses names according to some precise logical discipline (the same happens for the λ-calculus, which is hardly-ever used untyped since each variable has usually an 'intended' functionality). This discipline on names does not appear anywhere in the terms of the pure calculus, and therefore cannot be taken into account in proofs. *Types* can bring this structure back into light. Below we illustrate this point with two examples that have to do with *encapsulation*.

Encapsulation. Desirable features in both sequential and concurrent languages are facilities for encapsulation, that is for constraining the access to components such as data and resources. The need of encapsulation has led to the deveolpment of abstract data types and is a key feature of objects in object-oriented languages.

In CCS, encapsulation is given by the *restriction* operator. Restricting a channel x on a process P, written (using π-calculus notation) $\boldsymbol{\nu} x\, P$, guarantees that interactions along x between subcomponents of P occur without interference from outside. For instance, suppose we have two 1-place buffers, `Buf1` and `Buf2`, the first of which receives values along a channel x and resends them along y, whereas the second receives at y and resends at z. They can be composed into a 2-place buffer which receives at x and resends at z thus: $\boldsymbol{\nu} y\, (\texttt{Buf1} \mid \texttt{Buf2})$. Here, the restriction ensures us that actions at y from `Buf1` and `Buf2` are not stolen by processes in the external environment. With the formal definitions of `Buf1` and `Buf2` at hand, one can indeed prove that the system $\boldsymbol{\nu} y\, (\texttt{Buf1} \mid \texttt{Buf2})$ is behaviourally equivalent to a 2-place buffer.

The restriction operator provides quite a satisfactory level of protection in CCS, where the visibility of channels in processes is fixed. By contrast, restriction alone is often not satisfactory in the π-calculus, where the visibility of channels may change dynamically. Here are two examples.

Example 1 (A printer with mobile ownership [34]). Consider the situation in which several client processes cooperate in the use of a shared resource such as

a printer. Data are sent for printing by the client processes along a channel p. Clients may also communicate channel p so that new clients can get access to the printer. Suppose that initially there are two clients

$$\begin{aligned} \texttt{C1} &= \overline{p}\langle j_1\rangle.\,\overline{p}\langle j_2\rangle\ldots \\ \texttt{C2} &= \overline{b}\langle p\rangle \end{aligned}$$

and therefore, writing P for the printer process, the initial system is

$$\nu p\ \ (P\ \mid\ \texttt{C1}\ \mid\ \texttt{C2}).$$

One might wish to prove that C1's print jobs represented by j_1 and j_2 are eventually received and processed in that order by the printer, possibly under some fairness condition on the printer scheduling policy. Unfortunately this is false: a misbehaving new client C3 which has obtained p from C2 can disrupt the protocol expected by P and C1 just by reading print requests from p and throwing them away:

$$\texttt{C3} = p(j).\,p(j').\,\mathbf{0}.$$

□

Example 2 (A boolean package implementation [35]). For an even more dramatic example, consider a π-calculus representation of a simple boolean package:

$$\begin{aligned} \texttt{BoolPack1} = (\nu t, f, \texttt{if}\,)\Big(& \\ & \overline{\texttt{getBool}}\langle t, f, if\rangle \\ & \mid\ !t(x,y).\,\overline{x}\langle\rangle \\ & \mid\ !f(x,y).\,\overline{y}\langle\rangle \\ & \mid\ !\texttt{if}(b,x,y).\,\overline{b}\langle x,y\rangle\Big) \end{aligned}$$

The package provides implementation of the `true` and `false` values and of an `if-true` function. In the π-calculus, a boolean value is implemented as a process located at a certain name; above the name is t for the value `true` and f for the value `false`. This process receives two return channels, above called x and y, and produces an answer at the first or at the second depending on whether the value `true` or `false` is implemented. The `if-true` function is located at `if`, where it receives three arguments: the location b of a boolean value and two return channels x and y; the function interacts with the boolean located at b and, depending on whether this is `true` or `false`, an answer at x or y is produced. Both the boolean values and the `if-true` function are replicated so that they may be used more than once. Other functionalities, like `and`, `or` and `not` functions, can be added to the package in a similar way.

A client can use the package by reading at `getBool` the channels t, f and `if`. After this, what remains of the package is

$$\begin{aligned} & !t(x,y).\,\overline{x}\langle\rangle \\ \mid\ & !f(x,y).\,\overline{y}\langle\rangle \\ \mid\ & !\texttt{if}(b,x,y).\,\overline{b}\langle x,y\rangle \end{aligned}$$

But now the implementation of the package is completely uncovered! A misbehaving client has free access to the internal representation of the components. It may interfere with these components, by attempting to read from t, f or `if`. It may also send at `if` a tuple of names the first of which is not the location of a boolean value. If multiple processes get to know the access channels t, f and `if` (which may happen because these channel may be communicated), then a client has no guarantee about the correctness of the answers obtained from querying the package. □

Using types to obtain encapsulation. In the two examples, the protection of a resource fails if the access to the resource is transmitted, because no assumptions on the use of that access by a recipient can be made. Simple and powerful encapsulation barriers against the mobility of names can be created using *type* concepts familiar from the literature of typed λ-calculi. We discuss the two examples above.

The misbehaving printer client `C3` of Example 1 can be prevented by separating between the input and the output capabilities of a channel. It suffices to assign the input capability on channel p to the printer and the output capability to the initial clients `C1` and `C2`. In this way, new clients which receive p from existing clients will only receive the output capability on p. The misbehaving `C3` is thus ruled out as ill-typed, as it uses p in input. This idea of "directionality in channels" was introduced in [34] and formalised by means of type constructs, the i/o *types*. They give rise to a natural subtyping relation, similar to those used for reference types in imperative languages like Forsythe (cf: Reynolds [38]). In the case of the π-calculus encodings of the λ-calculus [27], this subtyping validates the standard subtyping rules for function types [42]. This subtyping is also important when modeling object-oriented languages, whose type systems usually incorporate some powerful form of subtyping.

A common concept in typed λ-calculi is *polymorphism*. It is rather straightforward to add it onto a π-calculus type system by allowing channels to carry a tuple of both types and values. Forms of polymorphic type systems for the π-calculus are presented in [12, 50, 48, 47, 35, 11]. Polymorphic types can be used in Example 2 of the boolean package `BoolPack1` to hide the implementation details of the package components, in a way similar to Mitchell and Plotkin's representation of abstract data types in the λ-calculus [30]. We can make channel `getBool` polymorphic by abstracting away the type of the boolean channels t and f. This forces a well-typed observer to use t and f only as arguments of the `if-true` function. Indeed, using polymorphism this way the package `BoolPack1` is undistinguishable from the package

$$\begin{aligned}\texttt{BoolPack2} = (\nu t, f, \texttt{if}\,)\Big(\;&\overline{\texttt{getBool}}\langle t, f, if\rangle\\ &\mid\; !t(x,y).\,\overline{y}\langle\rangle\\ &\mid\; !f(x,y).\,\overline{x}\langle\rangle\\ &\mid\; !\texttt{if}(b,x,y).\,\overline{b}\langle y,x\rangle\Big)\end{aligned}$$

The latter has a different internal representations of the boolean values (a value `true` responds on the second of the two return channels, rather than on the first, and similarly for the value `false`) and of the `if-true` function. By "undistinguishable", we mean that no well-typed observer can tell the difference between the two packages by interacting with them.

The packages `BoolPack1` and `BoolPack2` are not behavioural equivalent in the standard theories of behavioural equivalences for process calculi. Trace equivalence is considered the coarsest behavioural equivalence; the packages are not trace equivalent because they have several different traces of actions, e.g.,

$$\overline{getBools}\langle t, f, \texttt{if}\rangle \texttt{if}(t, x, y).\, \overline{t}\langle x, y\rangle$$

is a trace of `BoolPack1` but not of `BoolPack2`.

Similarly, suppose we have, as in some versions of the π-calculus, a mismatch construct $[x \neq y]P$ that behaves as P if names x and y are different, as $\mathbf{0}$ if they are equal. With polymorphism we can make `BoolPack1` equivalent to the package `BoolPack3` obtained from `BoolPack1` by replacing the line implementing the conditional test with:

$$\texttt{if}(b, x, y).\ (\overline{b}\langle x, y\rangle \mid [b \neq t][b \neq f]\ \texttt{BAD}).$$

where `BAD` can be any process. The new package is equivalent to `BoolPack1` because the value received at `if` for b is always either t or f. This example shows that a client of the boolean package is not authorized to make up new values of the same type as the boolean channels t and f, since the client knows nothing about this type. Again, the equivalence betweeb `BoolPack1` and `BoolPack3` is not valid in the standard theories of behavioural equivalences for process calculi.

Types for reasoning. Types are important for reasoning on π-calculus processes. First, types reduce the number of legal contexts in which a given process may be tested. The consequence is that *more behavioural equalities between processes are true than in the untyped calculus.* Examples of this have been given above. The equalities considered in these examples fail in the untyped π-calculus, even with respect to the very coarse notion of trace equivalence. That is, there are contexts of the untyped π-calculus that are able to detect the difference between the processes of the equalities. By imposing type systems, these contexts are ruled out as ill-typed. On the remaining legal contexts the processes being compared are undistinguishable. Useful algebraic laws, such as laws for copying or distributing resources whose effect is to localise computation or laws for enhancing the parallelism in a process, can thus become valid.

Secondly, types facilitate the reasoning, by allowing the use of some proof techniques or simplifying their application. For instance type system for linearity, confluence, and receptiveness (see below) guarantee that certain communications are not preemptive. This is a partial confluence property, in the presence of which only parts of process behaviours need to be explored. Types can also allow more efficient implementations of communications between channels, or optimisations in compilers such as tail-call optimisation.

Another situation where types are useful is in limiting the explosion of the number of the derivatives of a process. To see why this can be a problem, consider a process $a(x).P$. In the untyped π-calculus, its behaviour is determined by the set of all derivatives $P\{^b/_x\}$, where b ranges over the free names of P plus a fresh one. In case of a cascade of inputs, this gives rise to state explosion, which at present is a serious obstacle to the development of tools for mechanical analysis of processes. The number of legal derivatives of processes can be reduced using types. For instance, in the example of the boolean package `BoolPack1`, having polymorphic types we know that the only possible names that can be received for the parameter b of the `if-true` function are t and f.

Types in π-calculi: some references. In the λ-calculus, where functions are the unit of interaction, the key type construct is arrow type. In the π-calculus names are the unit of interaction and therefore the key type construct is the *channel* (or *name*) type $\sharp\, T$. A type assignment $a : \sharp\, T$ means that a can be used as a channel to carry values of type T. As names can carry names, T itself can be a channel type. If we add a set of *basic types*, such as integer or boolean types, we obtain the analogous of simply-typed λ-calculus, which we may therefore call the *simply-typed π-calculus*. Type constructs familiar from sequential languages, such as those for products, unions, records, variants, recursive types, polymorphism, subtyping, linearity, can be adapted to the π-calculus [12, 50, 51, 18, 48, 47, 24, 19, 32, 11, 35, 3].

If we have recursive types, then we may avoid basic types as initial elements for defining types. The calculus with channel, product and recursive types is the *polyadic π-calculus*, mentioned at the beginning of these notes. Its type system is, essentially, Milner's *sorting systems* [27], historically the first form of type system for the π-calculus (in the sorting system type equality is syntactic, i.e., 'by-name'; more flexible notions of type equality are adopted in later systems).

The following type systems are development of those above but go beyond traditional type systems for sequential languages. Sewell [43] and Hennessy and Riely [17, 16] extend the i/o type system with richer sets of capabilities for distributed versions of the π-calculus (also [9] extends i/o types, on a Linda-based distributed language). Steffen and Nestmann [45] use types to obtain confluent processes. *Receptive types* [40] guarantee that the input end of a name is "functional", in the sense that it is always available (hence messages sent along that names can be immediately processed) and with the same continuation. Yoshida [53], Boudol [7] and Kobayashi and Sumii [23, 46], Ravara and Vasconcelos [37] put forward type systems that prevent certain forms of deadlocks. Abadi [1] uses types for guaranteeing secrecy properties in security protocols. The typing rules guarantee that a protocol that typechecks does not leak its secret information. Typing rules and protocols are presented on the spi-calculus, an extension of the π-calculus with shared-key cryptographic primitives. Honda [19] proposes a general framework for the the above-mentioned types, as well as other type systems.

Experimental typed programming languages, or proposals for typed programming languages, inspired by the π-calculus include Pict [36], Join [10], Blue [6], and Tyco [49].

Reasoning techniques for typed behavioural equivalences are presented in [34, 5, 3, 26, 17] for i/o or related types, in [35] for polymorphic types, in [24] for linear types, in [40] for receptive types. One of the most important application areas for the π-calculus is object-oriented languages. The reason is that naming is a central notion both for these languages and for the π-calculus. Proof techniques based on types have been used to prove the validity of algebraic laws and programm transformations on object-oriented languages [22, 41, 8, 20].

Other type sytems for concurrent calculi. Type systems can be used to guarantee safety properties, such as the absence of run-time errors. Examples 1 and 2 above show more refined properties, in which types prevent undesirable interactions among processes (even if these interactions would not produce run-time errors) thus guaranteeing that certain security constraints are not violated. In the printer Example 1, i/o types prevent malicious adversary from stealing jobs sent to the printer. In the boolean package Example 2, polymorphism prevents free access to the implementation details of the package.

Here are other works that apply types to security, on calculi or languages that are not based on the π-calculus. Smith and Volpano [44] use type systems to control information flow and to guarantee that private information is not improperly disclosed. Program variables are separated into high security and low security variables; the type system prevents information from flowing from high variables to low variables, so that the final values of the low variables are independent of the initial values of the high variables. On the use of type systems for controlling the flow of secure information, see also Heintze and Riecke [15]. Leroy and Rouaix [25] show how types can guarantee certain security properties on applets. Necula and Lee's proof-carrying code [31] is an elegant technique for ensuring safety of mobile code; mobile code is equipped with a proof attesting the conformity of the code to some safety policy. Defining and checking the validity of proofs exploits the type theory of the Edinburgh Logical Framework.

Applications of type theories to process reasoning include the use of theorem provers to verify the correctness of process protocols and process transformations [4, 33, 14].

We conclude mentioning a denotational approach to types for reasoning on processes. Abramsky, Gay and Nagarajan [2] have proposed *Interaction Categories* as a semantic foundation for typed concurrent languages, based on category theory and linear logic. In Interaction Categories, objects are types, morphisms are processes respecting those types, and composition is process interaction. Interaction Categories have been used to give the semantics to data-flow languages such as Lustre and Signal, and to define classes of processes that are deadlock-free in a compositional way. [13] presents a typed process calculus whose design follows the structure of Interaction Categories. It is not clear at present how Interaction Categories can handle process mobility and distribution.

References

1. M. Abadi. Secrecy by typing in security protocols. In M. Abadi and T. Ito, editors, *Proc. TACS '97*, volume 1281 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.
2. S. Abramsky, S. J. Gay, and R. Nagarajan. Interaction categories and the foundations of typed concurrent programming. In *Deductive Program Design: Proceedings of the 1994 Marktoberdorf Summer School*, NATO ASI Series F, 1995.
3. R. Amadio. An asynchronous model of locality, failure, and process mobility. In *Proc. Coordination'97*, volume 1282 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.
4. D. Bolignano and V. Ménissier-Morain. Formal verification of cryptographic protocols using coq. Submitted to a journal, 1997.
5. M. Boreale and D. Sangiorgi. Bisimulation in name-passing calculi without matching. In *13th LICS Conf.* IEEE Computer Society Press, 1998.
6. G. Boudol. The pi-calculus in direct style. In *Proc. 24th POPL*. ACM Press, 1997.
7. G. Boudol. Typing the use of resources in a concurrent calculus. In *Proc. ASIAN '97*, volume 1345 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.
8. S. Dal-Zilio. Concurrent objects in the blue calculus. Submitted, 1998.
9. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
10. C. Fournet and G. Gonthier. The Reflexive Chemical Abstract Machine and the Join calculus. In *Proc. 23th POPL*. ACM Press, 1996.
11. C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Implict typing à la ML for the join-calculus. In *CONCUR'97*. Springer Verlag, 1997.
12. S. Gay. A sort inference algorithm for the polyadic π-calculus. In *Proc. 20th POPL*. ACM, 1993.
13. S. J. Gay and R. R. Nagarajan. A typed calculus of synchronous processes. In *10th LICS Conf.* IEEE Computer Society Press, 1995.
14. J. F. Groote, F. Monin, and J. C. Van de Pol. Checking verifications of protocols and distributed systems by computer. In *Proc. CONCUR '98*, volume 1466 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.
15. N. Heintze and Riecke J.G. The SLam calculus: Programming with security and integrity. In *Proc. 25th POPL*. ACM Press, 1998.
16. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. In U. Nestmann and B. C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages*, volume 16.3 of *ENTCS*. Elsevier Science Publishers, 1998.
17. M. Hennessy and J. Riely. A typed language for distributed mobile processes. In *Proc. 25th POPL*. ACM Press, 1998.
18. K. Honda. Types for dyadic interaction. In E. Best, editor, *Proc. CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer Verlag, 1993.
19. K. Honda. Composing processes. In *Proc. 23th POPL*. ACM Press, 1996.
20. H. Hüttel, J. Kleist, M. Merro, and U. Nestmann. Migration = cloning ; aliasing. To be presented at Sixth Workshop on Foundations of Object-Oriented Languages (FOOL 6), 1999.
21. C.B. Jones. A π-calculus semantics for an object-based design notation. In E. Best, editor, *Proc. CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*, pages 158–172. Springer Verlag, 1993.

22. J. Kleist and D. Sangiorgi. Imperative objects and mobile processes. In *Proc. IFIP Working Conference on Programming Concepts and Methods (PROCOMET'98)*. North-Holland, 1998.
23. N. Kobayashi. A partially deadlock-free typed process calculus. *Transactions on Programming Languages and Systems*, 20(2):436–482, 1998. A preliminary version in *12th Lics Conf.* IEEE Computer Society Press 128–139, 1997.
24. N. Kobayashi, B.C. Pierce, and D.N. Turner. Linearity and the pi-calculus. In *Proc. 23th POPL*. ACM Press, 1996.
25. X. Leroy and F. Rouaix. Security properties of typed applets. In *Proc. 25th POPL*. ACM Press, 1998.
26. M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. In *25th ICALP*, volume 1443 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.
27. R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, LFCS, Dept. of Comp. Sci., Edinburgh Univ., October 1991. Also in *Logic and Algebra of Specification*, ed. F.L. Bauer, W. Brauer and H. Schwichtenberg, Springer Verlag, 1993.
28. R. Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.
29. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, (Parts I and II). *Information and Computation*, 100:1–77, 1992.
30. J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Trans. Prog. Lang. and Sys.*, 10(3):470–502, 1988.
31. G.C. Necula. Proof-carrying code. In *Proc. 24th POPL*. ACM Press, 1997.
32. J. Niehren. Functional computation as concurrent computation. In *Proc. 23th POPL*. ACM Press, 1996.
33. L.C. Paulson. The inductive approach to verifying cryptographic protocols. *J. Computer Security*, 6:85–128, 1998.
34. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Journal of Mathematical Structures in Computer Science*, 6(5):409–454, 1996. An extended abstract in *Proc. LICS 93*, IEEE Computer Society Press.
35. B. Pierce and D. Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. In *24th POPL*. ACM Press, 1997.
36. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Indiana University, 1997. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, MIT Press.
37. A. Ravara and V.T. Vasconcelos. Behavioural types for a calculus of concurrent objects. In *3rd International Euro-Par Conference*, volume 1300 of *Lecture Notes in Computer Science*, pages 554–561. Springer Verlag, 1997.
38. J. C. Reynolds. Preliminary design of the programming language Forsythe. Tech. rept. CMU-CS-88-159. Carnegie Mellon University., 1988.
39. D. Sangiorgi. Locality and non-interleaving semantics in calculi for mobile processes. *Theoretical Computer Science*, 155:39–83, 1996.
40. D. Sangiorgi. The name discipline of receptiveness. In *24th ICALP*, volume 1256 of *Lecture Notes in Computer Science*. Springer Verlag, 1997. To appear in TCS.
41. D. Sangiorgi. Typed π-calculus at work: a proof of Jones's parallelisation transformation on concurrent objects. Presented at the Fourth Workshop on Foundations of Object-Oriented Languages (FOOL 4). To appear in *Theory and Practice of Object-oriented Systems*, 1997.
42. D. Sangiorgi. Interpreting functions as pi-calculus processes: a tutorial. Technical Report RR-3470, INRIA-Sophia Antipolis, 1998.

43. P. Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Proc. 25th ICALP*, volume 1443 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.
44. G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. 25th POPL*. ACM Press, 1998.
45. M. Steffen and U. Nestmann. Typing confluence. Interner Bericht IMMD7-xx/95, Informatik VII, Universität Erlangen-Nürnberg, 1995.
46. E. Sumii and N. Kobayashi. A generalized deadlock-free process calculus. In U. Nestmann and B.C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages*, volume 16.3 of *ENTCS*. Elsevier Science Publishers, 1998.
47. N.D. Turner. *The polymorphic pi-calculus: Theory and Implementation*. PhD thesis, Department of Computer Science, University of Edinburgh, 1996.
48. V.T. Vasconcelos. Predicative polymorphism in -calculus. In *Proc. 6th Parallel Architectures and Languages Europe*, volume 817 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.
49. V.T. Vasconcelos and R. Bastos. Core-TyCO, the language definition, version 0.1. DI/FCUL TR 98–3, Department of Computer Science, University of Lisbon, March 1998.
50. V.T. Vasconcelos and K. Honda. Principal typing schemes in a polyadic π-calculus. In E. Best, editor, *Proc. CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.
51. V.T. Vasconcelos and M. Tokoro. A typing system for a calculus of objects. In *Proc. Object Technologies for Advanced Software '93*, volume 742 of *Lecture Notes in Computer Science*, pages 460–474. Springer Verlag, 1993.
52. D. Walker. Objects in the π-calculus. *Information and Computation*, 116(2):253–271, 1995.
53. N. Yoshida. Graph types for monadic mobile processes. In *Proc. FST & TCS*, volume 1180 of *Lecture Notes in Computer Science*, pages 371–386. Springer Verlag, 1996.