In Cabernet Radicals Workshop, October 2002
http://www.wischik.com/lu/research/

New directions in implementing the pi calculus

Lucian Wischik, University of Bologna

30th August 2002

Do you know what the *pi calculus* is? It is a language invented ten years ago for describing concurrent and distributed systems. It has come to dominate theoretical research into concurrency and distribution, and now its time has come to be used in practice.

The author, along with Laneve and Gardner, has recently developed a distributed virtual machine [10, 12, 2] for the pi calculus. This is new territory – ours is actually the calculus' first true distributed implementation. We depart in several ways from mainstream ideas in the research community. Indeed, our implementation has more in common with the commercial product Microsoft Biztalk [5] (a recent tool used to integrate business systems and which itself is based partly on the pi calculus). In our future plans we have been partly inspired by the practical concerns faced by Biztalk; in turn, the designers of Biztalk are taking some of our ideas for their next version.

The goal of this paper is to present the practical lessons learned from our ongoing implementation experience and also from Biztalk. We hope to challenge some existing ideas, and to draw attention to fresh areas needing investigation. The basic motivation is that the pi calculus seems an easier way to write a wide range of concurrent and distributed programs.

1 The pi calculus and its market

This paper is about implementing the pi calculus language. We therefore start by describing it. To call it a language is in fact a little grandiose – it is really just a minimal core of commands and declarations, which will presumably be embedded in some richer language. In our current implementation we provide the commands as C functions, and we are working on the richer language; in Biztalk the commands are part of its internal XLANG. We will describe the language by example; for a full reference see *Communicating and Mobile Systems* by Milner [6].

The pi calculus describes programs which run in parallel and which interact over channels. Consider the example program

$$\overline{u}x.P \mid u(y).Q.$$

It contains one program $\overline{u}x.P$ ready to send the data x over the channel u, and afterwards to continue doing P. In parallel, a second program $u(y).Q$ is ready to receive the data y over the channel, and afterwards continue doing Q. The name y is a formal argument and is local to Q. The two programs interact as follows:

$$\overline{u}x.P \mid u(y).Q \;\;\rightarrow\;\; P \mid Q\{x/y\}.$$

Note how convenient it is to write parallel threads and synchronisation – easier than with fork, shared data and condition variables, and more flexible than remote procedure calls.

(The calculus' non-ASCII notation betrays its mathematical roots. Note also that mathematicians like to use the same language for states as well as programs, and hence they write substitution on terms $Q\{x/y\}$. In practice we actually compile terms into byte-code and use a local stack for formal arguments.)

A term may also be *replicated*, indicated with $!u(y).Q$. This is a service that runs at channel u and that spawns a fresh copy of Q every time it is invoked.

The *restriction* operator $(x)P$ declares a name x to be local in P. Names are more like heap variables than stack variables: they can be used outside the sub-program in which they were first created, and can persist after the sub-program ends; whereas stack variables can do neither.

A distinguishing feature of the pi calculus is *input mobility*. We explain with an example. When the program $u(x).x(y).Q$ has performed a rendezvous at u, then it will wait to perform a second rendezvous at y – but the actual channel y will only be determined at runtime. 'Input mobility' refers to this late binding of the input channel. It is absent from Biztalk and other distributed calculi such as the join calculus [1], where input channels are known at compile-time. We suspect it will prove to be a convenient feature. For instance, a cell-phone receives the name of a new base station, and then waits for messages from that new station.

Note: in the literature, when two programs have essentially the same "physical structure", they are treated as equivalent. But as implementors we have a different idea of physical structure from mathematicians. In this paper we write $\rightarrow$ for execution steps that are also deemed execution steps in the literature, and we write $\rightharpoonup$ for execution steps that are commonly deemed equivalences.

Selling points. Our instinct is that the pi calculus is a natural way to think about concurrency and to write concurrent programs; it certainly seems easier than the current paradigm of forks, pipes and condition variables. Moreover, since most areas of programming are concurrent – from high-level systems integration to low-level device drivers – the pi calculus seems natural for all levels of programming. No current programming paradigm spans such a wide range. This selling point was suggested to us by Meredith, designer of Biztalk.

So far, we have evidence (from the success of Biztalk) that the pi calculus is indeed natural and appropriate for high-level systems integration. We also have evidence (from the collective groans of programmers all over the world) that current paradigms are not working for concurrent middle-level programs such as word processors and Windows Explorer. Hopefully, when our implementation is complete, we will gain experience at this level of programming. A particular promise at this level is that behavioural type systems seem possible – so the compiler can ensure that one's code obeys a particular protocol or is deadlock-free.

As for low-level device drivers, it is worth noting that a computer's internal architecture is a kind of distributed system. In particular, there is high latency between the various devices (CPU, memory, peripherals); and communication between them is by message-passing. Moreover, the next generation of games consoles look set to use a much higher number of "distributed" special-purpose chips with a much higher comparative latency between them than has yet been seen. It is widely believed that some new programming paradigm will be needed to tame their complexity, but no one yet

knows which paradigm.

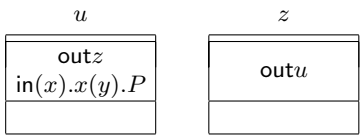
The *vertical market* for the pi calculus, from high level to low level, requires a more diverse research programme than is currently the case. For instance, efficiency is important for device-drivers but has not been widely addressed. And no single model of failures spans from Biztalk to device drivers.

To illustrate, we briefly outline the Biztalk failure model. This failure model has also not yet been addressed in the research community, but it should be. Biztalk is used in a commercial setting where it is essential that no data (e.g. accounting records) are ever lost. Reliable messaging protocols are used between parts of the system. Every event is written to a reliable log on disk. If one part of the system should crash then it can be rebooted, or the disk taken to a fresh machine. The challenge is to properly and completely recover the system's state after a reboot. (By contrast, existing work on failures has been focused on middle-level applications that have to reroute around failure, and on the low-level algorithms for reliable messaging.)

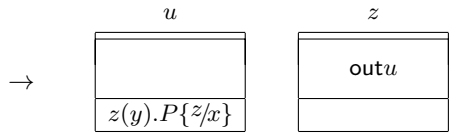
2 Channel-based implementation

Our distributed implementation of the pi calculus is *channel-based*. This means that the only things to exist at runtime are channels, each at its own location (or co-located with some other channels). In Biztalk the channels may belong to different companies. We split a program up into fragments, each fragment ready to rendezvous at some channel, and we deploy each fragment directly to this channel. Thus, synchronous rendezvous is local. We illustrate our implementation with a small example, and then compare with other work.

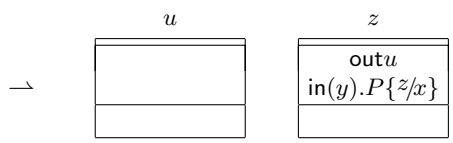
Consider the program $\overline{u}z \mid u(x).x(y).P \mid \overline{z}u$. After compilation the program is split into three fragments, two deployed at u and the third at z:



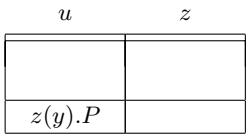
In the diagrams the letters u and z at the top represent names of channels. Each channel has two areas: the top area contains those fragments ready to rendezvous on the channel, and the bottom area (currently empty) contains those fragments yet to be deployed. In this example, a rendezvous at u is immediately possible:



As a result of the rendezvous, the *continuation* $z(y).P\{z/x\}$ has been placed in bottom area of u, indicating that it is now ready to be deployed. Deployment involves sending it across the network to z:



As mentioned, we suppose that channels may be co-located in the same address space. We draw such co-located channels as physically adjacent:



If two channels are co-located, then a program fragment can be sent from one to the other in constant time. We have used this property to prove efficiency results.

There are two noteworthy features in this channel-based implementation of synchronous rendezvous. First: it is local, and handshake-free. Second: if the continuation fragment $x(y).P$ had happened to be large, then the double cost of sending it (first to u in the initial deployment and then to z afterwards) would be prohibitive. The central challenge is to avoid this cost. The two main solutions are as follows:

Biztalk/Join solution. Biztalk and the join calculus [1] disallow input mobility, and disallow continuations after the send command $\overline{u}z$. This means that continuations need never be sent around the system (for the only continuations are those after an input command; and their ultimate location is known at compile-time). The join calculus additionally requires that every input command be replicated. This yields the property that every send message is guaranteed to find a ready receiver.

Fragments solution. Our solution is to break programs down into smaller fragments, and *pre-deploy* them to their ultimate location. For instance, given $u(x).v(y).w(z).P$, we might break it into $u(x).v(y)$ deployed to u, and $w(z).P$ deployed to w. The result is that the large continuations are already placed at their correct locations; all that is needed is a small message to trigger them. Technically, we have shown elsewhere [2] how the triggers can be encoded in a version of the pi calculus with *fusions* [3, 7]. (A fusion makes two channels become equivalent, in the sense that a message sent to either will have the same effect). Our machine therefore implements this fusion version of the calculus, rather than the pure pi calculus, and we leave pre-deployment as a compile-time optimisation. (Note that in the case of input mobility $u(x).x(y).P$, the ultimate location of $x(y)$ cannot be known until after $u(x)$, but even so we can still pre-deploy P.) Other authors have proposed weaker forms of fragmentation [8, 4] but in a theoretical setting, not for implementation.

Location-based calculi. The dominant paradigm in the research community is not channel-based but rather *location-based*. This means: assume several locations, each containing a collection of programs with their own channels. For instance, the state $m[\overline{u}x|u(y).P] \parallel n[R|u(z).Q]$ would represent two locations m and n, with the channels $m.u$ and $n.u$ distinct. This paradigm is used, for instance, in nomadic pict [9] and distributed pi and in the ambient calculus. (The ambient calculus additionally allows hierarchical locations).

However, these location-based calculi must now be augmented with new commands for interaction between locations. These new commands then face the same problem, of how to perform a distributed rendezvous. The answer has generally been to offer only a more basic form of interaction, such as process migration. (Because their distributed interaction is not pi-like, we do not call these "true distributed implementation of the pi calculus.")

Evaluation. It is widely thought that, by disallowing continuations after the send command, one obtains a calculus that is somehow more implementable. As we have

seen, this is not the case.

We believe that the channel-based paradigm is better than the location-based for reasons of simplicity. In particular, it uses just one type (located channels) rather than two (locations and channels); it uses just one set of interaction commands (over channels) rather than two sets (one for local and one for remote); and it allows the correctness of a program to be studied independently from its location behaviour, using familiar pi-calculus reasoning. Indeed, one reviewer dismissed the correctness proofs of our virtual machine as "trivial and obvious".

3 New names

In the pi calculus, the restriction operator $(x)P$ indicates that the name x is local in P. It does this through *alpha-renaming* and *scope extrusion*, as illustrated in the following program. (In the program, note that the x in the left fragment is local and hence different from the x in the right fragment.)

$$
\begin{array}{lll}
(x)(\overline{u}x) \mid u(y).\overline{x}y & & \textit{create a new channel } x \\
\quad \equiv & (x')(\overline{u}x') \mid u(y).\overline{x}y & \textit{alpha-rename } x \textit{ to } x' \\
\quad \equiv & (x')(\overline{u}x' \mid u(y).\overline{x}y) & \textit{extrude scope of } x' \\
\quad \rightarrow & (x')(\overline{x}x') & \textit{rendezvous on channel } u
\end{array}
$$

But creating a new channel is an actual physical event: it sets up a machine somewhere on the Internet which will listen for messages. This channel can be identified by its IP number and TCP port – for instance, $12.7.3.1{:}135$. However, the pi calculus does not represent the event of channel creation. Furthermore, its alpha-renaming and scope extrusion make little sense in an implementation. We therefore propose a new rule:

$$(x)P \;\rightharpoonup\; P\{^{12.7.3.1:135}/_{x}\}.$$

This means that *the restriction command, upon execution, yields a state with a particular freshly-created channel*. If we start with the original pi calculus and add this rule, and remove alpha-renaming and scope extrusion, the resulting calculus is in fact equivalent (up to full abstraction) to the original calculus [11]. And it is more implementable.

We suggested in the previous section that channels might be co-located. To program this, we proposed a modified restriction command

$$(x@y)P.$$

This indicates that the fresh channel x should be created in the same address space as y.

4 Conclusions

We believe that the pi calculus will prove a good language for writing concurrent and distributed programs: it is simpler than threads, and it looks to be applicable to the full range of programs from low-level device drivers to high-level systems integration. This wide range gives rise to new motivations and new challenges in pi calculus research.

Why has the research community been slow to develop distributed implementations of the calculus? We feel that several unhelpful paradigms have hindered development. In particular: the belief that continuation-less send commands are somehow more implementable (they are not); the use of location-based calculi (these add needless complexity); and the restriction operator (which is simply not implementable).

References

[1] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *POPL'96*, pages 372–385. ACM Press.

[2] P. Gardner, C. Laneve, and L. Wischik. The fusion machine. In *CONCUR 2002*, LNCS 2421:418–433.

[3] P. Gardner and L. Wischik. Explicit fusions. In *MFCS 2000*, LNCS 1893:373–382.

[4] C. Laneve and B. Victor. Solos in concert. In *ICALP'99*, LNCS 1644:513–523.

[5] Microsoft Corp. Biztalk Server. http:// www.microsoft.com/ biztalk.

[6] R. Milner. *Communicating and mobile systems: the Pi-calculus*. Cambridge University Press, 1999.

[7] J. Parrow and B. Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *LICS'98*, pages 176–185. Computer Society Press.

[8] J. Parrow. Trios in concert. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 621–637. MIT Press, 2000.

[9] P. Sewell, P. Wojciechowski, and B. Pierce. Location-independent communication for mobile agents: a two-level architecture. Technical report, University of Cambridge, 1999.

[10] L. Wischik. *Explicit Fusions: Theory and Implementation*. PhD thesis, Computer Laboratory, University of Cambridge, 2001.

[11] L. Wischik. Old names for nu. In progress, 2002.

[12] L. Wischik. Fusion machine prototype. http:// www.cs.unibo.it/ fusion.