Formal Models of Process Components

Peter Henderson

Department of Electronics and Computer Science
University of Southampton
SO17 1BJ, UK
Tel: (+44) 1703 593440
Fax: (+44) 1703 593045
Email: peter@ecs.soton.ac.uk

August 1997

Abstract

The way we have come to expect computer systems to behave is that we can simply add a new component to a running system and then this new component will begin to interwork with the running system without interuption of service. Describing and validating such systems of dynamic, reconfigurable components presents a challenge for contemporary methods of formal description. Milner's pi-calculus goes some way towards addressing this issue.

In this short paper we show that the pi-calculus is particularly good at describing the behaviour of components of a distributed system. We give a pragmatic introduction to the pi-calculus and illustrate this conjecture, using an example of clients and servers collaborating on the Web. The formalisation gives us the capability to define distributable components and to formulate properties of systems built from such components. The formalisation is different from, and probably complementary to, object-oriented formulations of such components. We describe how an elementary form of model-checking, actually a curtailed state-space search, can then be used to check these properties.

My position is that, formal methods (such as the pi-calculus) are mature enough to be used in the design and validation of components of large distributed systems and that the use of such methods will lead to the better design of components and of component-oriented architectures for dynamically reconfigurable systems.

Keywords: process components, system architecture, system evolution, client-server systems, reconfigurable systems, dynamically-loaded components, pi-calculus, model checking, state-space search

1 Background

This short paper attempts to bridge the gap between contemporary methods of formal description and contemporary methods of system design. Modern computer systems are large and necessarily built from components. In particular, we have come to expect that new components can be added to an existing system without undue disturbance to the services which the existing system provides. This is particularly true when it comes to systems deployed on the Web, where technologies such as Java have begun to make commonplace the development of dynamically reconfigurable distributed systems. Such systems present a challenge for contemporary methods of formal description. One answer to (at least part of) this problem is offered by the pi-calculus, a process algebra developed by Robin Milner [Mil91]. In this short paper we will investigate application of this method of formal description to the design of dynamically reconfigurable distributed systems.

My view of formal methods is that their use at the design stage is made more effective if a tool is available to check properties of specifications as they are produced. I personally favour executable formal models [Hen86, Gra96] but this position is a matter of taste. We will however discuss how we have used executable formal models based on the pi-calculus to check properties of components described in that language.

An earlier method of component description which we produced [Hen91], where components were viewed as providers of services (which should in some sense match when components are plugged together) lacked the formal basis which the pi-calculus can now supply. More recently, we have developed an informal method of describing complex systems built from components [Hen95]. This method has acquired some popularity, particularly in the organisation which sponsored the research (ICL). But again, that method lacked a formal underpinning which I now hope the pi-calculus will supply. This short paper goes some way towards realising this ambition.

2 Position

The problem we shall address is, can process algebras such as the pi-calculus help us to make formal models of reconfigurable components and of their composition into distributed systems? This problem is of significance partly because contemporary distributed systems are usually built from dynamically reconfigurable components and partly because the desire to have well-designed and validated systems might be more readily realised if suitable formal methods are used. The insights we are looking for to solve this problem are: do processes provide suitable components for the design of distributed systems, how do they compare with objects, is the pi-calculus mature enough to be used to design and validate such components and does this approach lead to a versatile (simple but powerful) architecture for components in a distributed environment?

2.1 Systems built from Process Components

Processes form an alternative to objects as components of systems when concurrency or distribution is of concern. They do not address all the concerns and there are many aspects of system design which are perhaps better addressed by object-oriented methods. But when it comes to issues of behaviour of systems whose components are acting cooperatively and concurrently, viewing at least some of those components as processes has distinct advantages.

The Web has re-educated us all into expecting systems to be more dynamic. We expect to be able to load new components into an already running system and have the system reconfigure itself to accomodate the new services which these objects supply. Network objects (see for example [Bir95]) have become a reality in Java and in languages such as Obliq [Car95], in particlar with the invention of Oblets [Bro96] which are distributed, active objects which can be used in conjunction with contemporary browsers to support cooperative working across the Web. These developments have demonstrated the strength of objects as components, and of the object-oriented approach, when it comes to designing dynamic systems even when those systems are to be distributed.

Processes form an interesting alternative to objects as components for such systems. Java has recently introduced Servlets which act as additional, dynamically loaded, active components on the server side of a client-server system. Servlets are more like processes than objects. They communicate through streams as you would expect with Java concurrency. Time will tell whether Servlets become a major feature of future distributed systems. For me, they reinforce the view that something more than objects is required.

The key issue is, how does a system evolve? Once a system is installed and running, ideally we never want to take it out of service. We want to simply drop new components into the system and expect old ones to graciously retire when their services are no longer required. This view of evolution of systems is one we hope to illustate in the simple examples of this short paper.

2.2 Why make Formal Models

It is not common practice to make formal models of systems especially if these systems are large. The reason is that formal methods are expensive to apply and usually an engineering project doesn't have the necessary resources. But there are good reasons for making formal models of large systems. These include the prospects for validation of the design, the ability it gives us to devise a resilient architecture for the system and also the ability it gives us to consider alternative designs and architectures.

The apparent mismatch between formal models and large scale systems may be more of an illusion than a reality. Large systems are built from components. Formal descriptions of components and of the methods for combining components into systems need not be large, just because the components and systems they describe are large. The trick is to apply these methods at a system level.

It seems that process algebras have the necessary concepts to be able to describe systems at a system level. Especially when those systems have concurrent behaviour and where they are intended to be distributed across a network. To support this conjecture we will introduce the pi-calculus (in a pragmatic way) and use it to describe systems built from clients and servers. The intention is that these descriptions are formal models of the behaviour of such components in a distributed environment such as that provided by the Web.

At the end of this section we will briefly discuss the methods we have used to validate the models we are about to present. These methods include execution and model checking. Finally, we will discuss what research needs to be done next in order to further develop these ideas.

2.3 Overview of the pi-calculus

The pi-calculus [Mil91, Mil93, Tur95] is a process algebra developed by Milner as a successor to his very successful CCS language [Mil89]. It is very similar to Hoare's CSP language [Hoare86], differing largely in the way that communications are synchronised. Our use of (the polyadic) pi-calculus in this short paper is going to be rather pragmatic and hence we will give a users' description of it, referring the reader to [Mil91] for a full semantics of the language. In fact, we will introduce the pi-calculus through a series of examples, showing how processes are specified and how to understand their possible communication behaviours.

There are two basic actions: an output action denoted $c!x$ and an input action denoted $c?x$. These respectively mean send value x on channel c and receive a value on channel c, binding that value to the name (variable) x. Sequences of actions can be composed so that the expression

$$c![3,d]\ .\ d?[x,y,z]\ .\ c!(x+y+z)$$

is the process whose behaviour is: send the tuple $[3,d]$ on channel c, then receive a 3-tuple on channel d whose components are bound to the variables x, y and z, finally send the sum $x+y+z$ on channel c.

For such a process to communicate, it must be running concurrently with other processes which do the matching half of the communication. For example if the above process runs in parallel with

$$c?[n,u]\ .\ u![n, n*n, n*n*n]\ .\ c?s$$

the net effect will be for the value $n+n*n+n*n*n$ to be bound to s at the end of the second process expression. [1]

Parallel composition of process expressions is denoted by $P|Q$. The essential semantic concept of communication is that two processes running in parallel may communicate with each other by sending values across channels. The interesting semantic aspect is when more than one communication is possible, which would arise for example in the context

$$(c!99\ .\ P1)|(c?x\ .\ P2(x))|(c?y\ .\ P3(y))$$

where $P1$, $P2(x)$ and $P3(y)$ are themselves process expressions. The first of the three parallel processes can communicate with either the second or the third. In this case either, but not both, communications could happen and thus the possible successor states to this process state are $P1|P2(99)$ and $P1|P3(99)$

Strictly, in pi-calculus, the values which are sent on channels are quite restricted, in order to be able to give a satisfactory mathematical definition. We will not restrict ourselves quite so rigidly in this short paper, nor will we justify the easing of the restriction. Rather, we will appeal to the readers' intuition about the rigour with which processes must be designed so as to not lead to stupid behaviour.

When we want to make a channel c local to a process expression P we use the construction **new** $c\ .\ P$, for example in

$$\textbf{new}\ c\ .\ ((c!99\ .\ P1)|(c?x\ .\ P2(x))|(c?y\ .\ P3(y)))$$

The only other pi-calculus construction we will make use of is non-deterministic (external) choice, denoted $P+Q$, which is a process which will behave as either P or as Q but not

[1] This use of ? and ! for input and output in the pi-calculus is after Turner [Tur95].

as both, the choice being determined by whichever of P or Q is able to communicate. For example, the expression

$$(c?x\ .\ P1(x)) + (d?y\ .\ P2(y))$$

is waiting for communication on channels c or d. If a concurrent process communicates with $c!99$ then the above process moves to the state $P1(99)$, whereas if it communicates with the $d!77$ then it moves to $P2(77)$. If however the process is

$$(c?x\ .\ P1(x)) + (c?y\ .\ P2(y))$$

then the communication $c!99$ will cause an arbitrary choice between $P1(99)$ and $P2(99)$ as the successor state. The language does not prescribe the behaviour in this case. The user of the language must be prepared to cope with either choice. We will define more elaborate process behaviours using recursion equations.

Consider for example the definition

$$sink(c) = c?x\ .\ sink(c)$$

This defines a process $sink(c)$ which will consume all values transmitted to it on channel c. Here c is a bound variable in a function definition which will be instantiated to an explicit channel for each actual use of the function. Consider also the following process

$$source(c) = (c!1\ .\ c!2\ .\ source(c)) + (c!3\ .\ c!4\ .\ source(c))$$

Readers should be able to convince themselves that

$$source(c1)|sink(c1)$$

runs forever, transmitting a sequence of integers on channel $c1$. An example of a sequence of values seen on $c1$ might begin 1 2 1 2 3 4 1 2 1 2 1 2 3 4 3 4 1 2 3 4 . . . . Obviously, every 1 is followed immediately by a 2, and every 3 is followed immediately by a 4. But 1s and 3s can interleave arbitrarily. Similarly

$$source(c1)|source(c1)|sink(c1)$$

also runs forever, transmitting a sequence of integers on channel $c1$. Since there are two processes behaving as $source(c1)$, either of them can transmit on $c1$ and in fact they will eventually both do so in an interleaved manner. An example of a sequence of values seen on $c1$ might begin $1\,\underline{1}\,2\,1\,\underline{2}\,\underline{1}\,\underline{2}\,2\,\underline{3}\,3\,4\,1\,\underline{4}\,\underline{1}\,2\,\underline{2}\ldots$ where the communications with one of the subprocesses have been underlined for clarity.

Suppose that we wished to define a source which was able to serve up values from a set $\{v1, v2, ..., vn\}$. What we want to write is

$$source(c) = (c!v1\ .\ source(c)) + (c!v2\ .\ source(c)) + \ldots + (c!vn\ .\ source(c))$$

We introduce a shorthand for this, giving ourselves a comprehension over sets. We write

$$\sum v : V\ .\ P(v)$$

to mean the sum over all values v in V of the processes $P(v)$. Our source now becomes

$$source(c) = \sum v : \{v1, v2, ..., vn\}\ .\ (c!v\ .\ source(c))$$

With this short preparation, we are now in a position to show how such process components can be used to describe the behaviour of existing or proposed client-server systems.

2.4 Models of Components on the Web

Our first model of behaviour on the Web is going to follow reasonably closely the simple example of the previous section. The components we will model will be clients and servers. A Web server is going to be modelled as a component which is a source of pages which can be requested over a unique channel. The Server is defined as follows

$$Server(db, s) = s?(p, c)\ .\ c!db[p]\ .\ Server(db, s)$$

The server listens on channel s for a communication of the form (p, c), which is a request for page p to be transmitted on channel c. We assume db is a database of pages and that $db[p]$ looks up the page named p and returns it (or some default page, perhaps with an error message in it). This page is then returned as requested on channel c and the behaviour repeats itself.

Consider how a client will identify a resource on the Web. It needs to know the channel to the server and the resource name (page name) on that server. So, the client knows a pair (s, p) and uses this to send a message to the server of the form $s!(p, c)$ which is a request for page p to be returned on channel c, exactly as the server is expecting. We can think of (s, p) as being a URL.

Suppose now we model the client as having a set of bookmarks, all of which are such URLs. For example the client's bookmarks might be the set

$$bm1 = \{(s1, p1), (s1, p2), (s2, p1)\}$$

where the client knows of two servers and of one or two pages on each. The behaviour of a suitable client might be

$$Client(bm, c) = \sum(s, p) : bm\ .\ (s!(p, c)\ .\ c?q\ .\ Client(bm, c))$$

Now if $db1$ and $db2$ are the databases served by the servers on channels $s1$ and $s2$ respectively then the following process exhibits a typical browsing behaviour.

$$Client(bm1, c1)|Server(db1, s1)|Server(db2, s2)$$

The client arbitrarily asks for one of the pages it knows of from either server. This it receives back on its private channel. The model doesn't describe what then happens to the received page q. Effectively it is discarded.

This model is unsatisfactory in many ways which we will proceed to fix. Firstly, the client can only browse pages it knows of at the outset. To overcome this we assume that when a page q is returned it may contain other URLs which the client can then select. Suppose the function $URLs(q)$ returns a set of URLs obtained from the page q. For example, we might have

$$\begin{array}{rcl} URLs(db1[p1]) & = & \{(s1, p2)\} \\ URLs(db1[p2]) & = & \{(s1, p1), (s2, p1)\} \\ URLs(db2[p1]) & = & \{(s1, p2), (s3, p1)\} \end{array}$$

Now we can modify the client to behave as follows

$$Client(bm, c) = \sum(s, p) : bm\ .\ (s!(p, c)\ .\ c?q\ .\ Client(bm \cup URLs(q),\ c))$$

where the client collects all the URLs from each page it visits and adds them to its set of bookmarks (it's now a kind of Web crawler).

Readers should be able to convince themselves that if the client starts in the state $Client(\{(s1, p1)\}, c1)$ then it will eventually be able to collect all the URLs on the pages on each of the servers which is running.

Each of the processes $Client(bm, c)$ and $Server(db, s)$ can be seen as components from which we can build various configurations of client-server systems. In practice, much more elaborate processes than these will be required to model realistic systems. But, for this short paper, our objective is to show how the pi-calculus can model dynamic systems, which are reconfigured as they are executing. To this end let us elaborate the client just one more time, this time to download a new application.

Suppose that we have a predicate $isAppl(q)$ which can determine if a resource q is a simple page or an executable (client-side) application. If $isAppl(q)$ returns *false* then q is a simple page. If it returns *true* then q is an application and $compile(q)$ is the loadable process which the client can execute.

We modify the client to behave as follows

$$
\begin{array}{rcl}
Client(bm, c) & = & \sum(s,p) : bm \ . \\
 & & (s!(p,c) \ . \ c?q \ . \\
 & & \quad \textbf{if } isAppl(q) \\
 & & \quad \textbf{then } Client(bm, c) | compile(q) \\
 & & \quad \textbf{else } Client(bm \ \cup \ URLs(q), \ c) \\
 & &)
\end{array}
$$

Now the client can continue to browse but a new application is running locally which presumably provides some useful service.

2.5 Componentology

So what has all this to do with building distributed systems from components? It seems reasonable to suppose that many important distributed systems in the future will be built using a client-server paradigm. Especially where the behaviour of the system is to evolve dynamically over time. Having methods to describe the behaviour of such systems formally gives us the ability both to explore alternative designs and to validate chosen designs to ensure that they have the behaviour which we expect. The pi-calculus affords one such method of formal description.

Describing components as processes, and studying how combinations of components behave, provides a model which is complementary to the object-oriented paradigm. Like the object-oriented approach we can specify the individual behaviour of components and then, abstracting from that, describe the composite behaviour of systems built from those components. Such a scheme of components and rules of combination we refer to as a componentology. Within the componentology we can design components and systems. We can formulate descriptions and ask questions. And by various means we can furnish answers to those questions.

2.6 Validation of Formal Models

We have build a simple tool based on the operational semantics for the pi-calculus given by Milner in [Mil91]. This tool does nothing more than code up the reduction rules given in that paper, but it does it in the context of a language which can support recursion equations. The tool can be used to code up the models of the earlier sections and to then navigate through them a step at a time. A menu is provided on which each of the possible next events is listed. The experimenter chooses an event (always of the form: send a particular message on a particular channel) and the executing model then changes state, which is reflected in a change in the choices available on the menu. The experimenter can always step backwards through the execution and make an alternative choice of event, as often as necessary to conduct a manual search of the state-space.

We have also automated this state-space search where the tool constructs a finite part of the state-space and presents it in the form of a finite state machine (ie state-event-state triples). The state-space search is pruned in various ways and we are experimenting with methods of pruning such as those pioneered in the model checking tools described by Clarke [Cla91], Ip and Dill [Ip96] and Jackson [Jac96].

2.7 Future Work

Our interest in the pi-calculus arose out of a problem we posed in [Ber94], where we described process support systems built from communicating process components. We needed a language to formalise the models we were describing there so that we could validate them and develop an architecture or a componentology suited to the design of distributed process support systems. This remains our goal and so the next step is to design a set of process components suited to this task. The components described in this short note go some way towards that objective, but we still have lots of interesting investigations to pursue.

Another objective is to realise an implementation (in Java) of a process support system which has been designed and validated formally. To this end we already have developed Java classes which implement channels whose behaviour is exactly modelled by the pi-calculus. In a related project we have captured requirements for business process support for a particular organisation. Mapping these requirements onto a real implementation will be done informally within that project. Our plan is to show that this mapping can also be done formally, by using our formally specified process components and by validating their composition into a system using state-space search. There are, of course, major semantic issues which we need to deal with in relation to how accurately the pi-calculus models we have given (or will give) match the actual behaviour of real Web components. These issues will be dealt with as a significant part of our proposed componentology.

3 Comparison

The ideas we have presented in this short paper will perhaps go some way towards convincing the reader that processes are complementary to, if not an alternative to, objects for the design and description of behaviour in distributed systems. The description technique given here is perhaps not as powerful as some object-oriented techniques (for example [Bre96]) but it does seem to have some advangages when it comes to description of concurrent behaviour. Perhaps, for distributed systems built from components, we need object-oriented techniques for the description of passive objects and techniques similar to those described here, based on process algebras, for description of active objects and concurrent combination of components.

An interesting question for formal methods remains: when we should use process algebras and when we should use state based methods of the Z, VDM, Object-Oriented kind to describe components and their composition into systems? For validation of system level descriptions it does seem that state-space search and model checking [Cla91, Ip96, Jac96] could play a major role. In particular as a practical way to remove design faults early in the system design process. We have shown how certain kinds of components can be described using pi-calculus and briefly described our method of validating these components. The trend in the component supply business seems to be towards versatile, dynamically-reconfigurable components such as applets, oblets, servlets and generally downloadable run-time add-ons.

This trend presents significant challenges for the formal methods community. The componentology provided by technologies such as Java (but also Corba, and OLE, and ActiveX) are state-of-the-art as far as realistically usable components go. In this short paper we have attempted to show that formal methods such as the pi-calculus are mature enough to describe such componentology. Perhaps we have succeeded in some small way. But, I am happy to conclude, there are still a significant number of major challenges ahead.

References

[Bir95] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward P. Wobber. *Network Objects.* Software Practice and Experience, 25(S4):87-130, December 1995. Also appeared as SRC Research Report 115, see http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-115.html

[Ber94] Berrington, N, David DeRoure, Mark Greenwood and Peter Henderson *Distribution and Change: Investigating two challenges for process enactment systems.* in Software Process Technology, EWSPT 94, Springer Verlag LNCS 772. (1994)

[Bre96] Breu, R and E Zucca *Algebraic Specification Framework for Object-Oriented Languages with Concurrency* Formal Aspects of Computing, Vol 8, No 6. (1996)

[Bro96] Brown, Marc H and Marc A Najork *Distributed Active Objects* Computer Networks and ISDN Systems, 28:1037–1052, May 1996. (Proceedings of the Fifth International World Wide Web Conference, Paris, France, May 6-10, 1996). Digital Systems Research Center, Report 141a, http://www.research.digital.com/SRC/bibindex/tmp/30547.html

[Car95] Cardelli, Luca *A Language with Distributed Scope.* Computing Systems, Vol 8, No1, Jan 1995.

[Cla91] Clarke E.M., J.R.Burch, O Grumberg, D.E.Long and K.L.McMillan *Automatic Verification of Sequential Circuit Designs.* in Mechanised Reasoning and Hardware Design, Royal Society Discussion Meeting, October 1991.

[Gra96] Gravell, A. and P. Henderson, *Executing formal specifications need not be harmful,* Software Engineering Journal, vol. 11, num 2., IEE. (1996)

[Hen91] Henderson, P & BC Warboys *Configuration Description for Component Reuse,* First International Workshop on Software Reuse, Dortmund, 1991

[Hen95] Henderson, P & Pratten, G.D. *POSD - A Notation for Presenting Complex Systems of Processes,* in Proceedings of the First IEEE International Conference on Engineering of Complex Systems, IEEE Computer Society Press (1995)

[Hen86] Henderson, P *Functional Programming, Formal Specification and Rapid Prototyping,* IEEE Transactions on Software Engineering, Vol.12, No.2, pp.241-250, 1986

[Hoare86] Hoare C.A.R *Communicating Sequential Processes* Prentice Hall, 1986

[Ip96] Ip C, Norris and David L Dill *Verifying Systems with Replicated Components in Murphi.* International Conference on Computer Aided Verification. (1996) see http://sprout.stanford.edu/dill/murphi.html

[Jac96] Jackson, D, Somesh Jha and Craig A Damon *Faster Checking of Software Specifications By Eliminating Isomorphs.* Proc. ACM Conf. on Principles of Programming Languages , Jan 1996. see http://www.cs.cmu.edu/afs/cs/user/dnj/pub/www/home.html

[Mil89] Milner, Robin *Communication and Concurrency* Prentice Hall International (1989)

[Mil91] Milner, Robin *The Polyadic pi-calculus: a tutorial* International Summerschool on Logic and Algebra of Specification, Marktoberdorf, (1991) see http://www.dcs.ed.ac.uk/lfcsreps/EXPORT/91/ECS-LFCS-91-180/index.html

[Mil93] Milner, Robin *Elements of Interaction: Turing Award Lecture* Communications of the ACM, Vol 36, No 1, January 1993

[Tur95] Turner David, N *The Polymorphic pi-calculus: Theory and Implementation* Ph.D Thesis, University of Edinburgh, 1995 see http://www.dcs.ed.ac.uk/lfcsreps/EXPORT/96/ECS-LFCS-96-345/index.html

4 Biography

Peter Henderson is Professor of Computer Science in the Department of Electronics and Computer Science at the University of Southampton in the UK. Prior to his move to Southampton in 1987 he was Professor at the University of Stirling. Henderson is also an ICL Fellow. He is head of the Declarative Systems and Software Engineering Research Group (see http://www.dsse.ecs.soton.ac.uk/) which combines research interests in Software Engineering, Formal Methods and Programming Languages. His own research includes executable specifications, component-based systems and process modelling. Currently Henderson is national co-ordinator for an EPSRC (Engineering and Physical Sciences Research Council) research programme entitled Systems Engineering for Business Process Change which has a legacy systems, COTS, component-based theme within it.