π-Calculus Semantics for the Concurrent Configuration Language Darwin*

Susan Eisenbach Ross Paterson

Department of Computing
Imperial College of Science, Technology and Medicine
London SW7 2BZ, United Kingdom

Abstract

DARWIN *is a configuration language for distributed and parallel programs, providing a hierarchical structure of components with dynamic binding. In order to specify precisely the behaviour of* DARWIN *programs, we sketch a translation of the features of the language into the π-calculus, a formalism for modelling concurrent processes. The match between underlying models for* DARWIN *and π-calculus is good. Examples done in the calculus are clean abstractions of the same solutions in other concurrent languages.*

1 Introduction

Without a formal specification a language is defined by its compiler and even with the best intentions several compilers for a language will lead to several variants. For any concurrent language designed to be implemented on very different architectures and the importance of a formal language specification becomes paramount. A good specification language is one that enables a clear simple specification to be written. This is most likely if the underlying model that the specification language supports matches that of the programming language.

DARWIN [4, 7] is a programming language for the configuration of programs across processors; ports can be linked to each other; messages are explicitly sent and received at ports. Implementations exist for parallel and distributed systems. An important characteristic of DARWIN is that it supports mobile processes by making the addresses of ports first class objects. This enables systems to be configured dynamically. DARWIN is a language[12] that grew out of Conic which has been used in large scale industrial problems[6].

Several languages such as CSP and CCS that have been devised to specify communicating computational systems. These languages should be suitable for specifying concurrent programming languages[9]. However, the modelling of mobile processes is not straightforward using these languages. Milner's recent system the π-calculus [11, 10] is designed to model concurrent computation consisting of processes which interact and whose configuration is changing. It does this by viewing a system as a collection of independent processes which may share communication links with other processes. Links have names. These names (or addresses) are the fundamental building blocks of the π-calculus. DARWIN's port addresses are like these names, helping to make the π-calculus a good system for describing DARWIN. To date though both the examples undertaken in the π-calculus and the languages specified in it have not been substantial[14]. So being able to define DARWIN in π-calculus is also a demonstration that the calculus has the expressive power required for solving real problems.

In this paper brief descriptions of DARWIN and the π-calculus are given. An example to demonstrate each language is developed. This is followed by a formal semantics of DARWIN in the calculus. The paper concludes with a discussion of the value of specifying a concurrent configuration language and what might be deducible about the behaviour of DARWIN programs.

2 Darwin

The DARWIN language[4, 8, 7] is intended for the description of both static and dynamic configurations of processes. Components may be written in any language, or in DARWIN itself; the existing implementation supports C, C++, Modula-2 and MPP (a message passing Pascal[5]). A grammar of DARWIN is given in figure 1.

The basic unit of DARWIN is a process. The interface of a process with its environment is a vector of names that are **provide**d by the process, and another vec-

*To appear in *Hawaii International Conference on System Sciences*, Koloa, Hawaii, January 1993.

$$
\begin{array}{rcl}
\textit{program} & = & \textit{component}^* \\
\textit{component} & = & \textbf{component}\ \textit{id}(\textit{parameters}) : \textit{instance-type}\ \textit{body} \\
\textit{body} & = & \{\ \textit{decl}^*\ \textit{action}^*\ \} \\
 & | & \textit{primitive-component} \\
\textit{type} & = & \textit{instance-type} \\
 & | & \textit{component-type} \\
 & | & \textbf{array}\ [\textit{num}_1..\textit{num}_2]\ \textbf{of}\ \textit{type} \\
 & | & \textit{primitive-type} \\
\textit{instance-type} & = & [\ \textbf{require}\ \textit{decl}^*; \textbf{provide}\ \textit{decl}^*\ ] \\
\textit{component-type} & = & \textbf{component}\ (\textit{parameters})\ \textit{instance-type} \\
\textit{decl} & = & \textit{id} : \textit{type} \\
\textit{action} & = & \textbf{inst}\ \textit{instance-id}\ \texttt{:=}\ \textit{component-id}(\textit{arguments}) \\
 & | & \textbf{bind}\ \textit{service-name}_1\ \texttt{--}\ \textit{service-name}_2 \\
 & | & \textbf{when}\ \textit{expr}\ \{\ \textit{action}^*\ \} \\
 & | & \textbf{forall}\ \textit{id} : \textit{expr}_1\ ..\ \textit{expr}_2\ \{\ \textit{action}^*\ \} \\
\textit{service-name} & = & \textit{id} \\
 & | & \textit{instance-id.id}
\end{array}
$$

Figure 1: A grammar for DARWIN

tor of names that are **require**d by the process. These names may be of any type, depending on the underlying language, but in the examples used here they will refer to communication ports. Both kinds of name may be referred to in the program defining the process. In a primitive process, written in the underlying language, the provided names are supplied with values, while required names are unresolved references. Any reference to an undefined name by a process will cause the process to block until a value is assigned.

To illustrate the language, consider a model of a group practice of doctors[8]. There are three kinds of primitive process, coded in some implementation language, namely doctors, a receptionist, and patients. The receptionist provides communication channels to talk to doctors and patients, each of which require corresponding channels. The interface types of these primitive components are therefore as follows:

$Recept = [\textbf{provide}\ doc: port(A),\ pat: port(B)]$
$Doctor = [\textbf{require}\ next: port(A)]$
$Patient = [\textbf{require}\ talk: port(B)]$

Primitive components of these types may be declared as follows:

component *receptionist*: *Recept*;
component *doctor*(*did*: *name*): *Doctor*;
component *patient*(*pid*: *name*): *Patient*;

Processes may also be written in DARWIN. Such a process may create instances of other processes, whether written in DARWIN or the underlying language. The core statement of DARWIN is

bind $id_1.requirement$ -- $id_2.provision$

which assigns the provided value of one instance to the required name of another. The **bind** statement may also refer to the provisions and requirements of the process being defined. The following combinations are permitted:

bind *provision* -- *instance.provision*

bind *instance.requirement* -- *requirement*

bind *provision* -- *requirement*

However, no variable may be bound twice.

In DARWIN, all actions may be performed concurrently. Thus the name on the right side may not have a value when the **bind** statement is performed. The semantics must be defined in such a way that the order in which the bindings are performed is immaterial.

Continuing with our group practice example, we can first define a component to generate the patients, and another to set up the doctors' surgery:

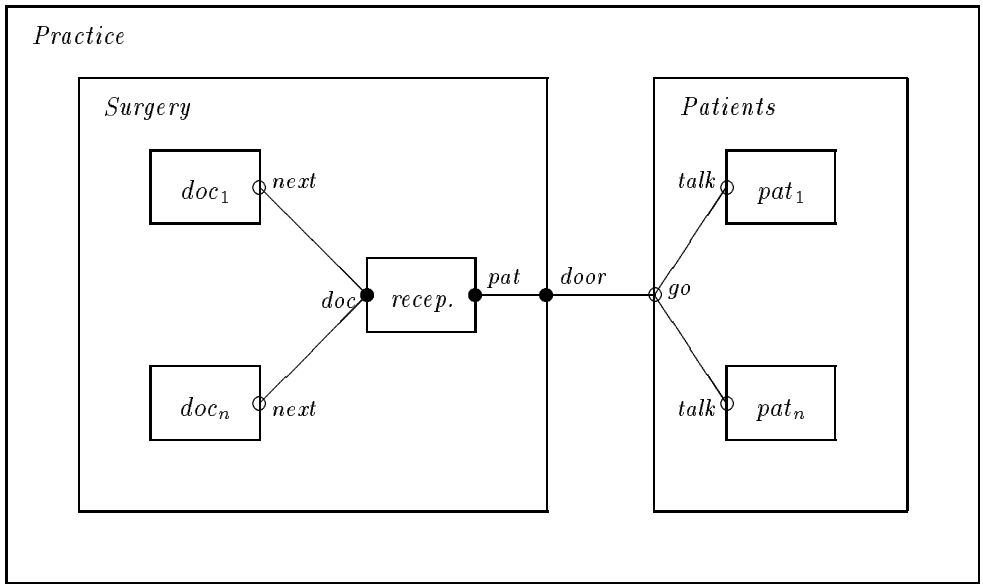

Figure 2: DARWIN example: a group medical practice

```
component patients(npat: int):
    [require go: port(B)]
{
    i: int;
    pat: array[1..npat] of Patient;

    forall i: 1..npat
    {
        inst pat[i]: patient(i);
        bind pat[i].talk -- go;
    }
}

component surgery(ndoc: int):
    [provide door: port(A)]
{
    i: int
    r: Recept;
    doc: array[1..ndoc] of Doctor;

    inst r := receptionist;
    bind door -- r.pat;
    forall i: 1..ndoc
    {
        inst doc[i] := doctor(i);
        bind doc[i].next -- r.doc;
    }
}
```

Finally, these subcomponents are connected together to form the whole:

```
component practice(ndoc, npat: int)
{
    inst s := surgery(ndoc);
    inst p := patients(npat);
    bind p.go -- s.door;
}
```

3 The π-calculus

The π-calculus is an elementary calculus for describing and analysing concurrent systems with evolving communications structure[11, 14, 10]. A system is a collection of independent *processes* which may be linked to other processes. Links have *names*; the name is the most primitive entity in the π-calculus; names have no structure. There are an infinite number of names, represented using lower-case letters. Processes *e.g.* $P, Q, R \ldots$ are built from names as follows:

- the parallel process $P \mid Q$ will execute *both* concurrently. The operation is commutative and associative.
- the replication $!P$ provides any number of copies of P. It satisfies the equation

$$!P = P \mid !P$$

Recursion can be recoded as replication and so need not be explicitly included as a separate

$$\begin{aligned}
\textsc{receptionist}(next, talk) &\stackrel{\text{def}}{=} talk(n, ill, reply).\\
&\quad next(answer).\overline{answer}(n, ill, reply).\\
&\quad \textsc{receptionist}(next, talk)\\
\\
\textsc{doctor}(next) &\stackrel{\text{def}}{=} (\nu\ answer)\\
&\quad \overline{next}(answer).answer(n, ill, reply).\\
&\quad \overline{reply}(diagnose(ill)).\\
&\quad \textsc{doctor}(next)\\
\\
\textsc{patient}(n, talk) &\stackrel{\text{def}}{=} (\nu\ reply)\\
&\quad \overline{talk}(n, rand(), reply).\\
&\quad reply(prescription).\mathbf{0}
\end{aligned}$$

Figure 3: The group practice processes in the π-calculus

method for building processes. Recursion will be used when it makes examples clearer.

- $(\nu\ y)P$ introduces a new name y with scope P. As usual, all free occurrences of y in P are bound by the quantifier, and can be uniformly renamed to any new name without changing the value of the process. The quantifier also satisfies the axioms

  $$(\nu\ x)(\nu\ y)P = (\nu\ y)(\nu\ x)P$$

  provided x and y are distinct names, and

  $$((\nu\ x)P) \mid Q = (\nu\ x)(P \mid Q)$$

  provided x does not occur free in Q.

- A communicating process C.

Communicating processes C are of the following kinds:

- the sum $C_1 + C_2$ will execute *either* C_1 *or* C_2. The operation is commutative and associative.
- $\overline{x}(y_1, \ldots, y_n).P$ means output the names $y_1, \ldots, y_n$ along the link x and then execute P.
- $x(z_1, \ldots, z_n).P$ receives names $y_1, \ldots, y_n$ along x and then executes P with y_i substituted for each free occurrence of z_i in P.
- $\mathbf{0}$ stops. It is an identity for both $\mid$ and $+$.

As well as being able to define processes the π-calculus defines a reduction relation between relations, written $P \rightarrow P'$, for process expressions P and P'. There is only one reduction axiom, called COMM:

$$(\cdots + x(y_1, \ldots, y_n).P \cdots) \mid (\cdots + \overline{x}(z_1, \ldots, z_n).Q \cdots) \rightarrow P\{z_1/y_1, \ldots, z_n/y_n\} \mid Q$$

Sub-processes under $\mid$ and ν, but not replication or communication, may also be reduced in this way.

For example, we can express the processes of our group practice example in this calculus, as in figure 3. The intended semantics is as follows. Patients arrive at the practice and talk to the receptionist. They tell her their name and their illness. Doctors who are capable of receiving patients inform the receptionist of their availability. The receptionist informs the available doctors of the names and illness of their perspective patients. The patient is given a prescription by the doctor. After a prescription is given, the patient leaves and the doctor becomes capable of seeing another patient.

Note that the example program assumes built-in functions *diagnose* and *rand*.

4 Translating DARWIN into the π-calculus

In order to model DARWIN, we shall need to add features to the π-calculus. However, this will will not make the language any more powerful, since the features can be defined in the calculus itself, and are thus mere abbreviations.

4.1 Asynchronous communication

The first feature is not required by DARWIN itself, but will be used by many of the concurrent languages on which it is superimposed, namely asynchronous communication channels, in which messages are queued.

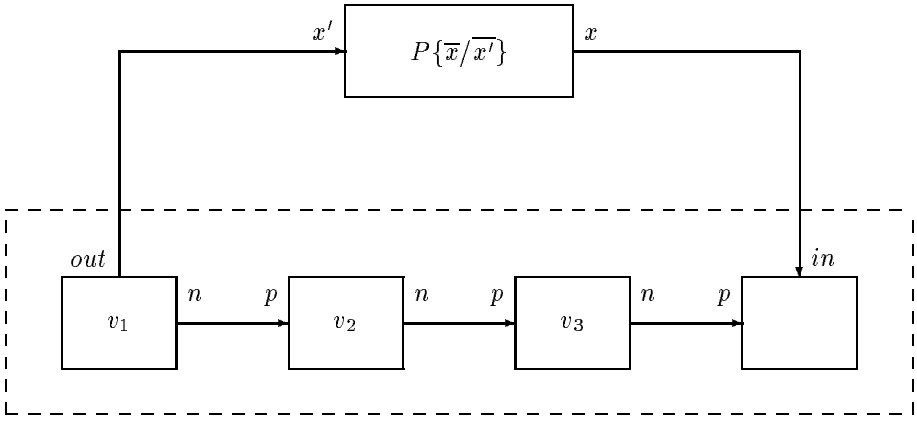

Figure 4: A queue containing three elements

The π-calculus uses synchronous communication, but can easily simulate queued streams. We define

$$(\nu_Q\ x)P \stackrel{\text{def}}{=} (\nu\ x)(\nu\ x')(\textsc{queue}(x, x') \mid P\{\overline{x}/\overline{x'}\})$$

A queued channel x is modelled by a pair of synchronous channels, with a queue process between them. The queue process contains an arbitrary number of element processes, each of which receives, holds and transmits a single value. At each stage, the queue consists of a number of active elements connected by channels as shown in figure 4. The first element holds the output channel. When requested by the process P, it yields its value along output channel, and then passes the output channel to the next element of the queue. The last element of the queue waits to receive a value from the input channel. When that happens, it passes the input channel (and a connexion to itself) to a new queue element, and waits until it receives the output channel, at which time it will be the head of the queue. Thus a queue element is defined as:

$$\textsc{element}(q) \stackrel{\text{def}}{=} q(in, p).in(v).(\nu\ n)\overline{q}(in, n).p(out).\overline{out}(v).\overline{n}(out).\mathbf{0}$$

Besides an indefinite number of element processes, the queue process includes a small process to start the first element:

$$\textsc{queue}(x, x') \stackrel{\text{def}}{=} (\nu\ q)((\nu\ n)\overline{q}(x, n).\overline{n}(x').\mathbf{0} \mid !\textsc{element}(q))$$

4.2 Data structures

We shall also need to define some data types in the π-calculus: Booleans, numbers and lists. The constructions are standard [10].

The values of these data types will be represented by port names. For example, Booleans are ports which expect a pair of ports and reply on one of them, thus permitting the implementation of conditionals. If a process P wishes to refer to the values $true$ and $false$, we wrap it as follows:

$$(\nu\ true)(\nu\ false)(P \mid !(true(x, y).\overline{x}().\mathbf{0}) \mid !(false(x, y).\overline{y}().\mathbf{0}))$$

Now suppose b is a name bound to either $true$ or $false$. We define

$$\textbf{if } b \textbf{ then } Q \textbf{ else } R \stackrel{\text{def}}{=} (\nu\ t)(\nu\ e)\overline{b}(t, e).(t().Q + e().R)$$

Darwin's **when** construct is a special case.

Natural numbers are given a unary encoding, using a similar scheme. For any number, one wants to know if it is zero or positive, and if the latter, what the preceding number is. Thus a number will be a port that expects to be passed two ports. If it is zero, it will respond on the first port, just like $true$ above. If it is positive, it will output its predecessor on the second port. That is, the successor of a number n is a port n' connected to a process

$$!(n'(x, y).\overline{y}(n).\mathbf{0})$$

Finally, we need a port $succ$ to supply such a port for each port n. The complete wrapping for P is

$$(\nu\ zero)(\nu\ succ)(P \mid !(zero(x, y).\overline{x}().\mathbf{0}) \mid !(succ(n, r).(\nu\ n')(\overline{r}(n').\mathbf{0} \mid !(n'(x, y).\overline{y}(n).\mathbf{0}))))$$

Inside P, we can introduce a new value (port) n', to be the successor of n, with the sequence

$$(\nu\ x)\overline{succ}(n,x).x(n').$$

As this construction shows, a function is modelled in the π-calculus by passing a new port, down which the result will be sent. All the usual arithmetic functions may be defined in the standard (but rather inefficient) way[1].

DARWIN's **forall** construct may now be represented using recursion and the conditional defined above.

An instance can be represented by a tuple containing of the provide and require ports of the sub-component. A tuple is represented by a port which outputs all the values at once.

Lists are defined in the same way as numbers. A list is either empty, in which case there is nothing more to be said of it, or not, in which case it contains a head element and a tail list. Thus an empty list is just like *true* or *zero* above, and a non-empty list with head h and tail t is a port l connected to a process

$$!(l(x,y).\overline{y}(h,t).\mathbf{0})$$

Thus the wrapping for P to provide lists is

$$\begin{array}{l}(\nu\ empty)(\nu\ cons)\\ (P \mid !(empty(x,y).\overline{x}().\mathbf{0}) \mid\\ \qquad !(cons(h,t,r).(\nu\ l)(\overline{r}(l).\mathbf{0} \mid !(l(x,y).\overline{y}(h,t).\mathbf{0}))))\end{array}$$

Arrays can be represented by lists.

4.3 Binding

We can simplify the restrictions on binding by grouping the externally visible names of a DARWIN component into a special *interface* instance, in which the provisions of the component appear as requirements, and vice versa. Thus all bindings will be of the form

bind $id_1.requirement$ -- $id_2.provision$

A provision in a primitive process will contain a value v, and is represented by a port p connected to a process

$$\textsc{Provide}(p,v) \stackrel{\text{def}}{=} !(p(a).\overline{a}(v).\mathbf{0})$$

This process receives addresses, represented in the π-calculus as ports that are to be read exactly once, and sends them its value. The process is replicated since a value may be used several times.

A requirement in a primitive process will contain an address a, and is represented by a port r connected to a process

$$\textsc{Require}(r,a) \stackrel{\text{def}}{=} r(x).\overline{x}(a).\mathbf{0}$$

This process receives a provision port and sends its address to it. Since the requirement may only be bound once, the process is not replicated.

The DARWIN statement

bind x -- y

will be represented as $\overline{x}(y)$. Suppose we bind such a primitive requirement and provision:

$$\begin{array}{l}\textsc{Provide}(p,v) \mid \textsc{Require}(r,a) \mid \overline{r}(p).\mathbf{0}\\ \rightarrow \textsc{Provide}(p,v) \mid \overline{p}(a).\mathbf{0}\\ \rightarrow \textsc{Provide}(p,v) \mid \overline{a}(v).\mathbf{0}\end{array}$$

So that the value v is sent to the address a.

An unassigned variable in a DARWIN component has a pair of ports, r for receiving a single provision, and p for receiving any number of addresses. One of these will be visible outside the component, and the other in the special *interface* instance, depending on whether the variable is provided or required. The ports are served by the following process

$$\textsc{Variable}(p,r) \stackrel{\text{def}}{=} r(x).!(p(a).\overline{x}(a).\mathbf{0})$$

If we bind such a variable to a requirement, we have

$$\begin{array}{l}\textsc{Variable}(p,r) \mid \textsc{Require}(r',a) \mid \overline{r'}(p).\mathbf{0}\\ \rightarrow \textsc{Variable}(p,r) \mid \overline{p}(a).\mathbf{0}\end{array}$$

After several such assignments, an unassigned variable carries a collection of addresses requiring values:

$$\textsc{Variable}(p,r) \mid \overline{p}(a_1).\mathbf{0} \mid \cdots \mid \overline{p}(a_n).\mathbf{0}$$

If such a variable is bound to a similar variable, its set of addresses is passed on:

$$\begin{array}{l}\textsc{Variable}(p,r) \mid \overline{p}(a_1).\mathbf{0} \mid \cdots \mid \overline{p}(a_n).\mathbf{0} \mid\\ \textsc{Variable}(p',r') \mid \overline{p'}(a'_1).\mathbf{0} \mid \cdots \mid \overline{p'}(a'_n).\mathbf{0} \mid\\ \overline{r'}(p).\mathbf{0}\end{array}$$

$$\begin{array}{l}\rightarrow \textsc{Variable}(p,r) \mid \overline{p}(a_1).\mathbf{0} \mid \cdots \mid \overline{p}(a_n).\mathbf{0} \mid\\ \quad !(p'(a).\overline{p}(a).\mathbf{0}) \mid \overline{p'}(a'_1).\mathbf{0} \mid \cdots \mid \overline{p'}(a'_n).\mathbf{0}\end{array}$$

$$\begin{array}{l}\rightarrow \textsc{Variable}(p,r) \mid \overline{p}(a_1).\mathbf{0} \mid \cdots \mid \overline{p}(a_n).\mathbf{0} \mid\\ \quad !(p'(a).\overline{p}(a).\mathbf{0}) \mid \overline{p}(a'_1).\mathbf{0} \mid \cdots \mid \overline{p}(a'_n).\mathbf{0}\end{array}$$

[1] Note that two successive applications of a function like *succ* to the same argument n will give two different ports, but we cannot compare them for equality; we can only compare their behaviours, and these are identical.

The residual process $!(p'(a).\overline{p}(a).\mathbf{0})$ passes on any subsequent bindings to the variable.

If such a variable is bound to a provision, the value will be sent to each address:

$$\textsc{Variable}(p, r) \mid \overline{p}(a_1).\mathbf{0} \mid \cdots \mid \overline{p}(a_n).\mathbf{0} \mid \textsc{Provide}(p', v) \mid \overline{r}(p').$$

$$\rightarrow !(p(a).\overline{p'}(a).\mathbf{0}) \mid \overline{p}(a_1).\mathbf{0} \mid \cdots \mid \overline{p}(a_n).\mathbf{0} \mid \textsc{Provide}(p', v)$$

$$\rightarrow !(p(a).\overline{p'}(a).\mathbf{0}) \mid \overline{p'}(a_1).\mathbf{0} \mid \cdots \mid \overline{p'}(a_n).\mathbf{0} \mid \textsc{Provide}(p', v)$$

$$\rightarrow !(p(a).\overline{p'}(a).\mathbf{0}) \mid \overline{a_1}(v).\mathbf{0} \mid \cdots \mid \overline{a_n}(v).\mathbf{0} \mid \textsc{Provide}(p', v)$$

The residual process

$$!(p(a).\overline{p'}(a).\mathbf{0}) \mid \textsc{Provide}(p', v)$$

has the same communication behaviour as

$$\textsc{Provide}(p, v) \mid \textsc{Provide}(p', v)$$

That is, any address sent to either p or p' will be send the value v.

5 Conclusions

This paper has presented a formal semantics in the π-calculus for the configuration language Darwin. The π-calculus is a new formalism. It is simple and elegant. There were no major problems that had to be worked around in order to write the Darwin definition. The least natural part was the definition of basic data types[10]. As these are not an interesting part of a configuration language this is not a serious defect. This demonstrates that the π-calculus is as powerful as a practical language for solving parallel or distributed configuration problems.

The semantics provided by the π-calculus definition should be compared with existing semantics for Darwin-like languages such as those in [3, 13]. The underling model of the π-calculus, where communication is primarily about exchanging names which then may be used for further communication, is very similar to the Darwin notion of a service. This leads to a substantially shorted definition then others have produced. More importantly, it makes the π-calculus an ideal language for specifying Darwin programs. This should help Darwin programmers to gain a greater understanding of what behaviour their programs will exhibit.

References

[1] D. Andrews. The Vienna development method. In D. Ince and D. Andrews, editors, *The Software Life Cycle*, pages 221–259. Butterworths, 1990.

[2] D. Andrews and W. Henhapl. Pascal. In D. Bjorner and C. Jones, editors, *Formal Specification and Software Development*, pages 175–252. Prentice Hall, 1982.

[3] S. C. Cheung. Darwin in Z. Technical Report REX-WP5-ICST-040-V1.0, REX, March 1991.

[4] N. Dulay. The Darwin configuration language. Imperial College Department of Computing Internal Report, March 1992.

[5] N. Dulay and J. Magee. MPP: Message passing Pascal. Imperial College Department of Computing Internal Report, April 1991.

[6] J. Kramer J. Magee and M. Sloman. Constructing distributed programs in CONIC. *IEEE Transactions on Software Engineering*, 15:663–375, 1989.

[7] J. Kramer, J. Magee, M.S. Sloman, and N. Dulay. Configuring object based distributed programs in REX. *IEEE Software Engineering Journal*, March 1992.

[8] J. Magee, N. Dulay, and J. Kramer. Structuring parallel and distributed programs. In *Proceedings of the IEEE International Workshop on Configuring Distributed Systems*, March 1992.

[9] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[10] R. Milner. The polyadic π-calculus: A tutorial. Technical Report ECS-LFCS 91-180, University of Edinburgh, October 1991.

[11] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. Technical Report ECS-LFCS 91, University of Edinburgh, 1989.

[12] M. Peltu. Out of the scullery. *Computing*, May 1992.

[13] M.D. Rice and S.B. Seidman. A formal model for module interconnection languages. Auburn University Internal Report, 1990.

[14] D. Walker. π-calculus semantics of object-oriented programming languages. In *Conference on Theoretical Aspects of Computer Software*, Tohoku University, Japan, 1991.