What Do You Get From a π-calculus Semantics?

Matthias Radestock and Susan Eisenbach

Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ, United Kingdom
E-mail: {M.Radestock,S.Eisenbach}@ic.ac.uk

Abstract. Darwin is a programming system for the development of distributed and parallel programs. Darwin programs consist of three parts. Firstly, there is a configuration part which provides a hierarchical structure of components with dynamic binding. Secondly, there is the actual communication part which provides the interaction and synchronisation required by the system. Finally, there is the computation part providing the component programs written in C++. The subdivision of concurrent programs into organisation, communication and computation leads to programs that are easy to specify, compile and execute. In order to specify precisely the behaviour of Darwin programs, we translate the organisation and communication into the π-calculus, a formalism for modelling concurrent processes. The π-calculus specification enables us to deduce behavioural properties of Darwin programs.

1 Introduction

The behaviour of an executing program should not come as a surprise to the writer of that program. Yet with programs that run on parallel and distributed systems, it is notoriously difficult to say with certainty what can be expected. Giving a formal specification of a programming language enables one to have that certainty; without a formal specification a language is defined by its compiler. 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 the importance of a formal language specification becomes paramount.

But there is a problem with formality. Unless the world view that the formal system models is the same as that of the programming system it will be easier to execute a program than to prove useful properties about its behaviour. Therefore the underlying model that the specification language supports must be similar to that of the programming language.

The Darwin configuration language [MKD93, EP93, JKD92] is an interconnection language for the configuration of modules across processors. With this approach we achieve a separation of the description of the system structure from the algorithms used to describe individual processes. Thereby a process instantiation and reuse is permitted in different contexts. The separate structural description of the system is also of use during the design, construction, documentation and subsequent maintenance of a system. In essence, the structural description corresponds to the issue of *programming in the large* whereas the process description corresponds to *programming in the small*. The Darwin system grew out of Conic which has been used in large scale industrial problems[JMS89]. An important characteristic of the Darwin system is that it enables systems to be configured dynamically.

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[Mil89]. However, the modelling of dynamic systems is not straightforward using these languages. These formalisms have been used to prove properties about termination and consistency of programs, but do not have mobile processes as first class objects.

Milner's π-calculus[MPW89, Mil91] 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 or bindings with other processes. Links have names. These names (or addresses) are the fundamental building blocks of

the π-calculus. Darwin ports are like these names, helping to make the π-calculus a good system for describing the communication.

In this paper we show how to translate Darwin programs into their π-calculus equivalents. The full power of the Darwin language is not needed for this because it is a language for defining configuration models and we are interested in reasoning about instances of these models rather than about the models themselves. We first define a subset of Darwin called Static Darwin which is sufficient to express instances of Darwin programs. We then show how Darwin can be translated into Static Darwin. This is followed by the translation of Static Darwin into π-calculus. Various models for the communication system are investigated on π-calculus level. The paper concludes with what can be deduced about the behaviour of Darwin programs. In particular the Darwin runtime system flattens the component hierarchy before execution. We show that this runtime optimisation does not alter the behaviour of executing programs.

2 Darwin

The Darwin language[Dul92, MDK93, JKD92] is intended for the description of a hierarchical configurations of processes. Composite components are written in the configuration language itself and primitive components are written in C++. The Darwin system grew out of Conic which has been used in large scale industrial problems[JMS89]. Current implementations are based on the *Regis* distributed system[MKD93]. An important characteristic of the Darwin system is that it enables systems to be configured dynamically by making the addresses of ports first class objects.

The basic unit of the configuration language is a component or process. A Darwin configuration component consists of a list of declarations. Instances of components are declared using the **inst** keyword. The **inst** declaration actually causes a process to be created. The interface of a component with its environment is a vector of names that are **provide**d by the process, and another vector of names that are **require**d by the process. These names may be of any type; the Darwin configuration language does not place any restrictions on the types (but it does ensure that program construction is type secure). Although the configuration language does not place any restriction on what can be a provision or requirement, in the examples used here they will refer to communication ports for unidirectional communication where the information is transmitted from the required to the provided ports. The names of provisions and requirements may be referred to in the program defining the process.

A **bind** declaration is used in order to bind two components together to enable the transference of data . This is the core statement of the configuration language.

bind $id_1.requirement - id_2.provision$

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.

Darwin also enables a limited amount of control structuring of its declarations. **When** declarations are guards that enable one or more declarations to be optionally declared. Since it is not uncommon to want to perform the same declaration repetitively there are **forall** declarations.

The following is an example of a Darwin program and its graphical representation:

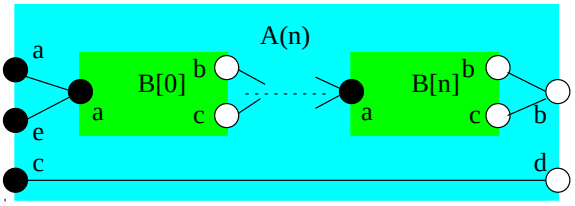

Fig. 1. Example of a Configuration

```
component B {
provide a;
require b,c;
        }
```

```
component A(int n) {
provide a, c, e;
require b, d;
array   B[n]:B;
forall i:0..n
        inst B[n];
forall i:0..n-1 {
        bind    B[i].b--B[i+1].a;
                B[i].c--B[i+1].a;
                }
bind    a--B[0].a; e--B[0].a;
        B[n].b--b; B[n].c--b;
        c--d;
        }
```

The abstract component `B` has three ports - one provided (`a`) and two required (`b,c`). The component `A` has three provided ports (`a,c,e`) and two required ports (`b,d`). Inside `A` $n+1$ instances of the abstract component `B` are chained together by connecting their required ports `b,c` to the provided port `a` of the next component in the chain. The provided port `a` of the first component is connected to the provided ports `a,e` of the composite component. Similarly the required ports `b,c` of the last component in the chain are connected to the required port `b` of the composite component. Finally there exists a direct link between the provided port `c` and the required port `d`.

3 The Translation

The translation of Darwin into the π-calculus is done in several steps. Figure 2 shows all the transformations. The definition of a distributed application consists of three parts:

- *Darwin specification* - the parameterised specification of the system configuration (i.e. the component specifications)
- *C++ process definition* - the parameterised definitions of all the component processes as well as auxiliary code
- *initial parameters* - parameters for creating instantiated specifications and definitions out of the parameterised form

The Darwin compiler translates the Darwin specification into C++ code. Together with the process definitions this is then compiled. The instantiation with the initial parameters takes place at the start of the execution. A Darwin program specifies a *class* of possible actual configurations. The initial parameters determine which of these configurations will be used.

The instantiation of the Darwin specification can be carried out separately. The result (after some structural code transformations) is a specification in Static Darwin. Unlike the original Darwin specification it only describes *one* configuration. In a similar manner the C++ process

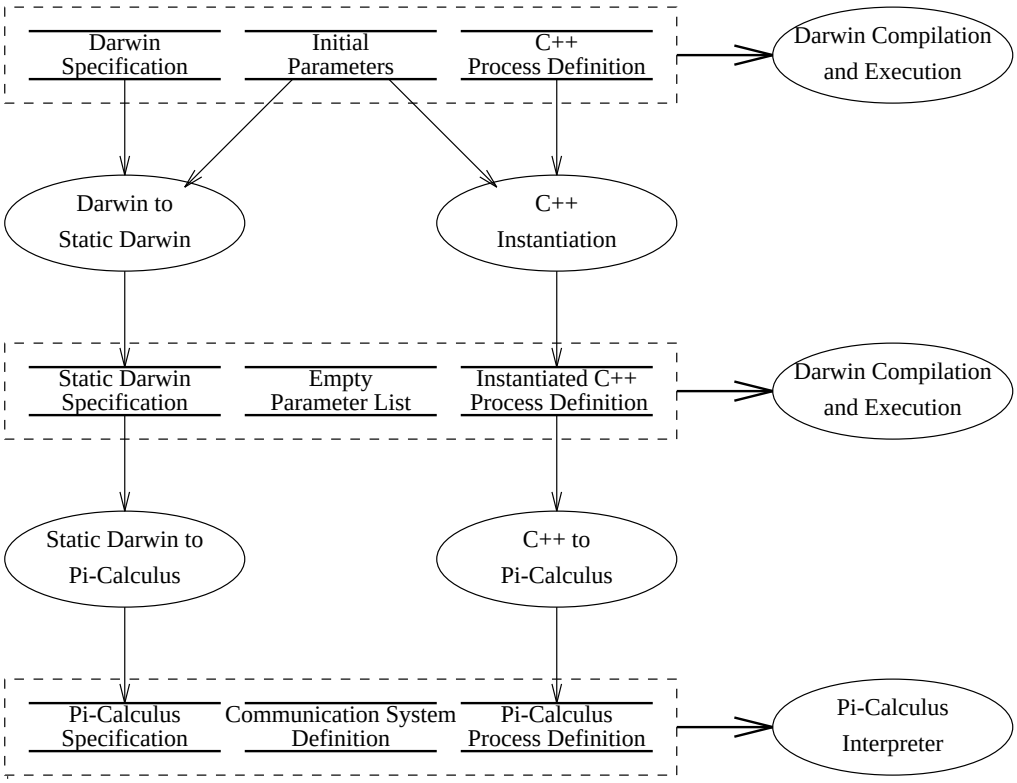

Fig. 2. The Translation of Darwin into π-calculus

definitions can be instantiated. This process is straightforward and is performed by textual substitution and replication. As the language Static Darwin is a subset of Darwin the Darwin compiler can translate the new specification and definition. No parameters are needed upon execution. The transformed specifications and definitions are equivalent to the original (three part) application definition in the sense that the resulting program exhibits the same behaviour as the original.

The Static Darwin specification can be translated into the π-calculus. The same needs to be done with the instantiated C++ process definition. Since the aim of the translation was to provide a representation of the communication activities rather than the whole program, it was not undertaken. A suitable π-calculus representation for the underlying communication system has been found. The π-calculus specification and π-calculus process definition together define the semantics of the original application definition. Since the π-calculus definitions can be interpreted and hence executed we can assert the observational equivalence between our formal semantics and the transformed specifications and definitions and hence can deduce the observational equivalence with the original program:

$$\begin{aligned} & Darwin\ Specification\ +\ C++\ Process\ Definition\ +\ Initial\ Parameters \\ = & \ Static\ Darwin\ Specification\ +\ Instantiated\ C++\ Process\ Definition \\ = & \ \pi - calculus\ Specification\ +\ \pi - calculus\ Process\ Definition \end{aligned}$$

3.1 Static Darwin

With Darwin we can specify a model for configurations. Any actual configuration is an instance of this model. An example to illustrate this relationship is shown in Figure 3. The language for describing a particular instance is *Static Darwin*. Reasoning about Darwin is reasoning about a set of possible configurations whereas reasoning about Static Darwin is reasoning about the *behaviour* of particular configurations. In our work we were interested in the latter.

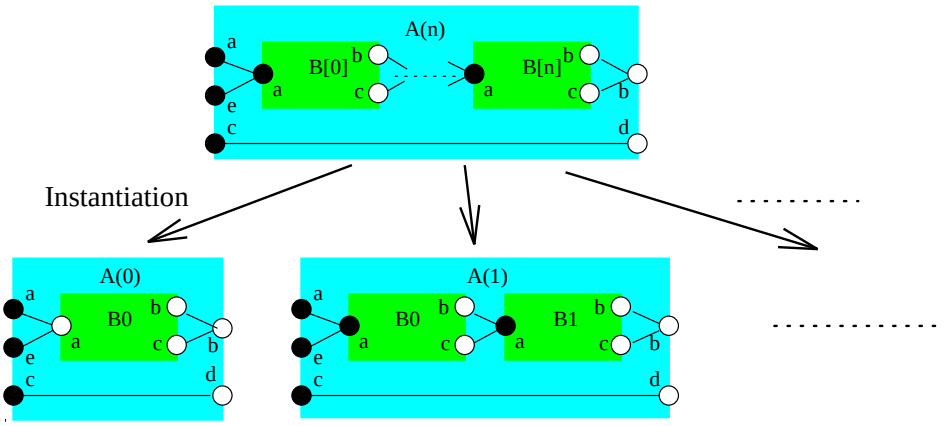

Fig. 3. Instances of a Darwin Specification

The limiting case of a specification for a model of configurations is that there exists only one possible configuration satisfying that model.[1] Clearly we must be able to express such a specification in Darwin. It turns out that the syntax of Darwin enables us to restrict the language to a subset where only these specifications can be expressed. Hence Static Darwin is Darwin without the following concepts:

- parameterised structures
- replicated structures (instance arrays, interface arrays, the **forall** statement)
- variant structures (**when**-statement)

The complete syntax of Static Darwin is listed in Appendix B. As Static Darwin is a subset of Darwin it can be compiled using the Darwin compiler.

It was decided not to keep the replicated and variant structures in the language Static Darwin, but to unfold and replace them. When further translating Static Darwin the unfolding would have had to be done at some stage anyway. By doing it already in the first step (the translation from Darwin to Static Darwin) the complexity of Static Darwin is reduced.

3.2 The Translation of Darwin into Static Darwin

In the previous section we listed the Darwin features that are not part of Static Darwin. We now show how these features can be translated into Static Darwin.

Parameterised Structures In Darwin a component specification can have parameters. If an instance is created from that specification values are assigned to these parameters. Together with replicated and variant structures this is used to alter the component's structure. The most straightforward way is to create a separate component specification in Static Darwin for each of the possible cases arising from a parameterised instantiation. However, in most cases this is impossible as the number of cases is infinite. What we do instead is to create a specification for every *needed* case. The number of components in a running application is clearly finite. So is the number of cases arising from each parameterised component specification. As a result of this approach we get a translation of *one* particular instantiation of the Darwin specification. Altering the initial conditions (i.e. the parameters passed to the top-level component) requires a re-translation. This could be seen as a flaw but in fact Darwin does the same. The instances are created *once* at the beginning.

[1] It is assumed that the domain of configurations includes an element representing an empty configuration, i.e. a configuration with no components and no bindings.

The instantiation of a Darwin specification is carried out top-to-bottom, i.e. one particular instance is created by instantiating the parameterised component specification of the top component. The instantiator calls itself recursively to instantiate the parameterised component specifications of the types of subcomponents. If an instantiation statement

inst $I : C(val_1, \ldots, val_n)$

is encountered the associated parameterised component specification

component $C(arg_1, \ldots, arg_n)$

is instantiated (i.e. with a recursive call to the instantiator), yielding a specification in Static Darwin

component $C_{val_1,\ldots,val_n}$

and the instantiation statement itself is translated to

inst $I : C_{val_1,\ldots,val_n}$

The arguments can be of the types *int, char, char*, double*. The syntax of Darwin doesn't include expressions for changing the value of variables. Hence once the formal parameters have been instantiated they can be treated as constants. When we instantiate a parameterised component specification we therefore can replace all occurrences of variables (and explicitly declared constants) with the actual values. The only exception is the **forall** statement (see below). It implicitly declares a variable which changes with each cycle of the loop.

Replicated Structures Arrays of component instances and interfaces can be represented by listing all the elements. Thus an array A with n elements is translated (e.g. *unfolded* into a list of names, $A_0 \ldots A_{n-1}$). For accessing particular elements it is then sufficient to translate their names. The **forall** statement is a control structure often occurring in conjunction with arrays. A **forall** statement is translated into a sequence of instantiated *loop bodies*. For every cycle we create an instance of the loop body with the occurrences of the loop variable being instantiated to a particular value, e.g. we *unfold* it.

Variant Structures The translation of the **when** statement is straightforward. The associated sequence of statements is either included or discarded, depending on the result of evaluating the condition.

3.3 The Translation of Static Darwin into the π-calculus

Each component specification is translated into a process definition. The provided and required ports appear as parameters in the definition. The visibility of internal ports (i.e. ports of subcomponents) is restricted to the component process. The process is a parallel composition of

- subcomponent processes
- communication processes
- the actual process attached to the component[2]

Subcomponent processes are acquired by instantiating the associated process definition of the component specification of the subcomponent type. Each required or provided port name is indexed with the instance name thus avoiding name clashes if multiple instances of the same component type exist. A communication process is inserted for every binding of ports. The following example shows the translation of a simple Static Darwin specification. It illustrates the use of all the possible binding forms and of multiple instantiation.

[2] In Darwin only the components at the leaves of the decomposition hierarchy may have a process attached and hence become active components. Composite components are inactive.

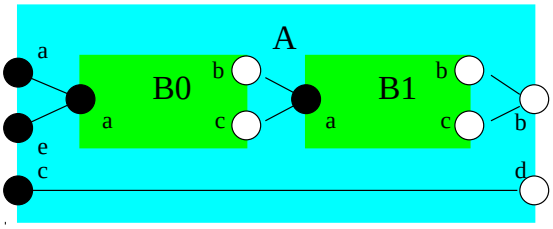

Fig. 4. Example

Darwin specification:

```
component B {
provide a;
require b,c;
        }

component A {
provide a, c, e;
require b, d;
inst    B0:B;
        B1:B;
bind    a--B0.a; e--B0.a;
        B0.b--B1.a; B0.c--B1.a;
        B1.b--b; B1.c--b;
        c--d;
        }
```

π-calculus representation:

$$P_B(a,b,c) \stackrel{\text{def}}{=} Q_B$$

$$P_A(a,c,e,b,d) \stackrel{\text{def}}{=} (\nu\, a_{B0}, b_{B0}, c_{B0}, a_{B1}, b_{B1}, c_{B1})($$
$$P_B(a_{B0}, b_{B0}, c_{B0}) \mid$$
$$P_B(a_{B1}, b_{B1}, c_{B1}) \mid$$
$$Comm(a, a_{B0}) \mid Comm(e, a_{B0}) \mid$$
$$Comm(b_{B0}, a_{B1}) \mid Comm(c_{B0}, a_{B1}) \mid$$
$$Comm(b_{B1}, b) \mid Comm(c_{B1}, b) \mid$$
$$Comm(c, d) \mid$$
$$Q_A)$$

Communication System Communication is modelled by sending names along channels. These names represent data transmitted between components. The output on a *provide* port is received from the communication process, representing the binding. It is then sent to the other participant of the connection. A component process can receive the information from the participating *require* port. Various ways of communication (e.g. synchronous, asynchronous, in sequence delivery) can be modelled by altering the specification for the communication process. For a synchronous, fault-free connection the process equation is as follows:

$$Comm(in, out) \stackrel{\text{def}}{=} in(l).l(x).l(r).\overline{out}(l).\overline{l}(x).\overline{l}(r).Comm(in, out)$$

The communication protocol is as follows:

1. The sender sends a name.
2. The sender sends the data and the name of a reply link along the channel represented by the previously sent name.
3. The receiver receives the name of the link. This is followed by the data and name of the reply link sent along that link.
4. The receiver acknowledges the receipt by sending something back along the reply link.
5. The sender receives the acknowledgement sent by the receiver along the reply link.

The specification allows for both synchronous and asynchronous communication. We illustrate this in the following example:

$$System \stackrel{\text{def}}{=} Sender \mid Comm(req, prov) \mid Receiver$$
$$Sender_{sync} \stackrel{\text{def}}{=} (\nu\, l, r)(\overline{req}(l).\overline{l}(x).\overline{l}(r).r(y).Sender'_{sync})$$

$$Sender_{async} \stackrel{\text{def}}{=} (\nu\ l,r)(\overline{req}(l).(\overline{l}(x).\overline{l}(r).r(y).\mathbf{0}\ \mid\ Sender'_{async}))$$
$$Sender_{nseq} \stackrel{\text{def}}{=} (\nu\ l,r)(\overline{req}(l).\overline{l}(x).\overline{l}(r).r(y).\mathbf{0}\ \mid\ Sender'_{nseq})$$
$$Receiver \stackrel{\text{def}}{=} prov(l).l(x).l(r).\overline{r}(y).Receiver'$$

The synchronisation scheme is entirely determined by the sender. The reply link is needed to ensure that the receiver actually received the message. In a direct communication this is not needed as the π-calculus relies on synchronous interaction, but because we have a communication system (represented by the *Comm* process), this is not sufficient. The sender would become unblocked as soon as the the interaction with that communication system takes place. The data might never actually get to the receiver. By introducing the reply link we offer a way to overcome this. If the sender wishes to make sure that the receiver gets the sent message it can block until it receives the acknowledgement along the reply link. This is illustrated in $Sender_{sync}$. In-sequence-delivery is ensured by blocking on the interaction with the communication system, as it is done in $Sender_{async}$. Finally $Sender_{nseq}$ shows how an out-of-sequence delivery becomes possible. It should be noted that the above communication model *Comm* is an abstraction. For instance, in an actual distributed system the acknowledgement would be sent through the communication process, i.e. in the opposite direction to the message. Here a separate, direct connection is established between sender and receiver along the reply link.

Dynamic Ports and Dynamic Binding *Dynamic ports* are ports which are created and destroyed at run-time. *Dynamic binding* is establishing and destroying links of communication at run-time. One way of doing this is to communicate *port references* between components. The reference to a port of a component can be passed along channels to other components. The component processes of these components can then send/receive information directly to/from these ports.[3] This form of *dynamic binding* enables the alteration of the static structure set up by an instance of a Darwin specification.

The translation of this concept into π-calculus turns out to be straightforward as ports are represented as names and name passing is one of the key concepts of the calculus. To illustrate how such a communication can be achieved we use the example from Figure 4 and formulate a process description for the component process B.

$$Q_B \stackrel{\text{def}}{=} a(m,n).(\overline{m}(a,b).\mathbf{0}\ \mid\ n(u,v).\mathbf{0})\ \mid\ \overline{c}(a,b).\mathbf{0}$$

This process can receive two port references m and n from its port a. It then outputs the references to two of its own ports (a and b) to the port addressed by the first of the received port references, and tries to input two port references from the second one. Independently the process attempts to output two of its own port references (a and b) to its own port c. This model does not use any communication process since the component processes communicate with each other directly. It also enables the simplification of the binding of the two component processes which can be defined as:

$$P_{B0,B1} \stackrel{\text{def}}{=} (\nu\ i,j,k)(P_B(i,j,j)\ \mid\ P_B(j,k,k)) \tag{1}$$
$$= (\nu\ i,j,k)((i(m,n).(\overline{m}(i,j).\mathbf{0}\ \mid\ n(u,v).\mathbf{0})\ \mid\ \overline{j}(i,j).\mathbf{0})\ \mid\ (j(m,n).(\overline{m}(j,k).\mathbf{0}\ \mid\ n(u,v).\mathbf{0})\ \mid\ \overline{k}(j,k).\mathbf{0})) \tag{2}$$
$$= (\nu\ i,j,k)((i(m,n).(\overline{m}(i,j).\mathbf{0}\ \mid\ n(u,v).\mathbf{0}))\ \mid\ (\overline{i}(j,k).\mathbf{0}\ \mid\ j(u,v).\mathbf{0})\ \mid\ \overline{k}(j,k).\mathbf{0})) \tag{3}$$
$$= (\nu\ i,j,k)((\overline{j}(i,j).\mathbf{0}\ \mid\ k(u,v).\mathbf{0})\ \mid\ (j(u,v).\mathbf{0}\ \mid\ \overline{k}(j,k).\mathbf{0})) \tag{4}$$
$$= \mathbf{0} \tag{5}$$

[3] The current version of Darwin only allows information to be *sent* to port references.

We want to focus on the interaction of the two processes and therefore have excluded interactions with the environment by restricting the visibility of the ports. The above equations show the evolution of the resulting process. Figure 5 illustrates the associated configurations. The first

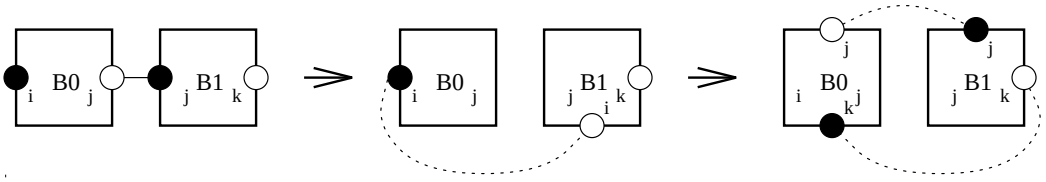

Fig. 5. Dynamic Binding by Passing of Port References

configuration corresponds to the second equation which denotes the static configuration. The component $B0$ passes port references to its ports i and j to the component $B1$ along the the link between the two components. As a result we get a new (dynamic) link from a dynamic port i in $B1$ to the port i in $B0$. Another dynamic Port j in $B1$ is created as well. $B1$ then transmits port references to its ports to $B0$ along the dynamic link. $B0$ creates two new dynamic ports and links them to the corresponding ports in $B1$.

The example illustrates how port references can be used to create configurations which evolve rapidly. The power of the π-calculus in representing such configurations becomes apparent.

Lazy Instantiation Recent versions of Darwin support *lazy instantiation*, i.e. a component is created when one of its ports is accessed for the first time. Lazy instantiation is straightforward to express in the π-calculus. Any input or output prefix can be viewed as a trigger for a lazy instantiation of the remaining process expression. In Darwin the instantiation is triggered when a *provide* port is accessed i.e. something is sent to the port. Hence for a component that is to be created dynamically we generate a special process that intercepts these requests, creates the component and forwards the request to it. Given the component specification C which declares n provided and m required ports, named $p_0 \ldots p_{n-1}$ and $r_0 \ldots r_{m-1}$, the resulting π-calculus process is then $C(p_0, \ldots, p_{n-1}, r_0, \ldots, r_{m-1})$. The process equation for lazy instantiation is then:

$$C'(p_0, \ldots, p_{n-1}, r_0, \ldots, r_{m-1}) \stackrel{\text{def}}{=} \sum_{i=0}^{n-1} p_i(l).(C(p_0, \ldots, p_{n-1}, r_0, \ldots, r_{m-1}) \mid \overline{p_i}(l).\mathbf{0})$$

The process definitions for lazy instantiation of the components used in the example are:

$$\begin{aligned} P'_A(a,c,b,d) &\stackrel{\text{def}}{=} a(l).(P_A(a,c,e,b,d) \mid \overline{a}(l).\mathbf{0}) \\ &\quad + c(l).(P_A(a,c,e,b,d) \mid \overline{c}(l).\mathbf{0}) \\ &\quad + e(l).(P_A(a,c,e,b,d) \mid \overline{e}(l).\mathbf{0}) \\ P'_B(a,b,c) &\stackrel{\text{def}}{=} a(l).(P_B(a,b,c) \mid \overline{a}(l).\mathbf{0}) \end{aligned}$$

Assuming that the subcomponent **B1** in **A** is to be instantiated lazily (i.e. the statement in the Darwin code `inst B1: B` is replaced by `inst B1: dyn B`), then the resulting process definition for **A** would have $P'_B(a_{B1}, b_{B1}, c_{B1})$ instead of $P_B(a_{B1}, b_{B1}, c_{B1})$ as an element of the parallel composition.

4 Translation of the Communication Primitives

Communication primitives are the link between the high-level language, used to describe the component processes, and the configuration. The given translation of some commonly used primitives

into the π-calculus restricts the translation of the high-level language to the parts which are relevant for modelling communication. The primitives are translated using the semantic function $\mathcal{C}_\pi$ which takes as its argument a sequence of C++ statements and returns a π-calculus expression:

$$\begin{aligned}
\mathcal{C}_\pi[\![p.\mathbf{out}(x);P]\!] &\stackrel{\text{def}}{=} (\nu\, l,r)(\overline{p}(l).\overline{l}(x).\overline{l}(r).r(y).\mathcal{C}_\pi[\![P]\!]) \\
\mathcal{C}_\pi[\![p.\mathbf{send}(x);P]\!] &\stackrel{\text{def}}{=} (\nu\, l,r)(\overline{p}(l).(\overline{l}(x).\overline{l}(r).r(y).\mathbf{0} \mid \mathcal{C}_\pi[\![P]\!])) \\
\mathcal{C}_\pi[\![p.\mathbf{in}(x);P]\!] &\stackrel{\text{def}}{=} p(l).l(x).l(r).\overline{r}(y).\mathcal{C}_\pi[\![P]\!]
\end{aligned}$$

The intended interpretation of the statements is that $p.\mathbf{out}(x)$ sends the value x to the port (referenced by) p. It blocks until this operation is carried out successfully (synchronous communication). The non-blocking version of this statement is called **send**. To receive a value from a port p and store it in a variable x the $p.\mathbf{in}(x)$ statement is used. Like **out**, **in** blocks until successful completion.

$$\mathcal{C}_\pi\left[\!\left[\begin{array}{l}\mathbf{select}\ \{ \\ \quad G_0{:}p_0(x_0) \Rightarrow P_0 \\ \quad G_1{:}p_1(x_1) \Rightarrow P_1 \\ \quad \ldots \\ \quad G_{n-1}{:}p_{n-1}(x_{n-1}) \Rightarrow P_{n-1} \\ \quad \mathbf{else} \Rightarrow P_n \\ \quad \};P\end{array}\right]\!\right] \stackrel{\text{def}}{=}$$

$$\prod_{i=1}^{2^n-1}\left(\begin{array}{l}\mathcal{B}_\pi\left[\!\left[\bigwedge_{j=0}^{n-1} cond(f(i,j),\mathcal{B}[\![G_j]\!],\neg\mathcal{B}[\![G_j]\!])\right]\!\right](c) \mid \\ c(x).[x=\texttt{TRUE}]\sum_{j=0}^{n-1} cond(f(i,j),\mathcal{C}_\pi[\![p_j.\mathbf{in}(x_j);P_j;P]\!],\mathbf{0})\end{array}\right) \mid$$

$$\mathcal{B}_\pi\left[\!\left[\bigwedge_{j=0}^{n-1}\neg\mathcal{B}[\![G_j]\!]\right]\!\right](c) \mid c(x).[x=\texttt{TRUE}]\mathcal{C}_\pi[\![P_n;P]\!]$$

The **select** statement alters the course of computation depending on the port which sent the data. If information is available at more than one port a non-deterministic choice is made. (Unlike the C++ implementation which is deterministic.) If no information is available the process waits. The set of ports upon which a selection is made is determined by guards which evaluate to *Booleans*. The **else** branch is entered if none of the guards evaluates to *true*. The guard evaluation is done with the semantic function $\mathcal{B}$ which takes a regular C++ expression, evaluates it and returns a logical *true* or *false*. Another semantic function, $\mathcal{B}_\pi[\![b]\!](c)$, creates a translation of a Boolean value b into the π-calculus. A name (`TRUE` or `FALSE`) is reported along the channel c. The function *cond* takes as its first argument a Boolean and returns the second or third argument if the first argument is *true* or *false* respectively:

$$cond(b,c_1,c_2) = \underline{if}\ b\ \underline{then}\ c_1\ \underline{else}\ c_2$$

The auxiliary function $f(i,j)$ tests whether bit j in the binary representation of i is set:

$$f(i,j) = ((i \wedge_N 2^j) =_N 2^j)$$

The values of the guards change during execution. Hence the translation of the statements would have to be carried out at run-time. However, by devising semantic functions which translate C++ expressions into π-calculus representations of Booleans and implementing logical operators for these representations on the π-calculus level a complete translation outside a run-time context becomes possible.

5 Reasoning about Instances of Darwin Specifications

The translation of Darwin into the π-calculus enables us to reason about particular instantiations of the specification in terms of a well-defined mathematical framework. We can detect errors in Darwin specifications and experiment with various models of the communication system.

5.1 Errors in the Specification

There are a variety of possible errors in a Darwin specification that can be detected at the π-calculus level. Some of these errors are not detected by the Darwin compiler.

Unbound Ports As long as a port is not used (i.e. one of the component processes tries to send something to it or to receive something from it) an unbound port doesn't affect the course of computation. In case of a synchronous communication the sender goes into the deadlock state if it tries to access an unbound port. The same holds for the receiver in any communication model. An asynchronous send doesn't block the sender but one subprocess (the one handling that particular communication) is deadlocked.

It should be noted that the π-calculus representation would allow us to model late binding. Any suspended process could proceed as soon as the previously unbound port becomes bound. This third party binding cannot be expressed yet in Darwin, but is currently being developed.

Multi-cast Multi-cast is the binding (direct or indirect) of a required port to more than one provided port. The intended interpretation is that the information is sent to *all* of the ports. Multi-cast is not supported in the current version of Darwin, but we can translate such a specification into π-calculus. The result, however, is *not* the expected multi-cast but a non-deterministic choice. Exactly one port is selected and the information transmitted to it.

Infinite Recursive Structure An example of an infinite recursive structure is:

```
component C {
provide a;
inst    c:C;
        }
```

The Darwin compiler does *not* detect this. Instead it generates code that when executed keeps on creating components until the system crashes due to memory exhaustion. In the π-calculus translation we can observe the same behaviour. The translation of the above component yields:

$$P_C(a) \stackrel{\text{def}}{=} (a_c)(P_C(a_c) \mid Q_C)$$

This structure can evolve without any external trigger resulting in an ever growing parallel composition of processes. One way to prevent this in Darwin is to declare the instance as *dynamic*. The translation then gives

$$P_C(a) \stackrel{\text{def}}{=} (a_c)(P'_C(a_c) \mid Q_C)$$
$$P'_C(a) \stackrel{\text{def}}{=} a(l).(P_C(a) \mid \overline{a}(l).\mathbf{0})$$

which can be rewritten as

$$P_C(a) \stackrel{\text{def}}{=} (a_c)(a_c(l).(P_C(a_c) \mid \overline{a_c}(l).\mathbf{0}) \mid Q_C)$$

Now the structure only evolves when Q_C sends something to the port a_c, creating exactly one more process - the same behaviour as we get from executing the compiled Darwin specification.

Directionality Constraints Darwin enforces some constraints on the **bind** statement. The general rule is that the **bind** statement expresses the direction of the communication. Data is transmitted from the "left side" to the "right side". When binding two subcomponents the *require* port has to be the first argument and the *provide* port the second argument. If the arguments are in the wrong order the Darwin compiler issues an error message. In the π-calculus translation it results in a communication process that can never communicate with its environment. In a synchronous communication we get a deadlock of sender and receiver.

5.2 Modelling the Communication System

By changing the specification of the communication process we can experiment with different models for communication systems and observe the effects on the overall system. An example of a simple communication process was shown in Section 3.3. A communication system which always accepts messages but doesn't preserve order can be specified as follows:

$$Comm(in, out) \stackrel{\text{def}}{=} in(l).l(x).l(r).(\overline{out}(l).\overline{l}(x).\overline{l}(r).\mathbf{0} \mid Comm(in, out))$$

The following process buffers the messages in a queue and hence preserves the order:

$$\begin{aligned} Comm(in, out) &\stackrel{\text{def}}{=} (\nu\ q)(\overline{q}().\mathbf{0} \mid Comm'(in, out, q)) \\ Comm'(in, out, q) &\stackrel{\text{def}}{=} in(l).l(x).l(r).(\nu\ s)(q().\overline{out}(l).\overline{l}(x).\overline{l}(r).\overline{s}().\mathbf{0} \mid Comm(in, out, s)) \end{aligned}$$

Finally a system that may duplicate or discard messages but preserves the order:

$$\begin{aligned} Comm(in, out) \stackrel{\text{def}}{=}\ & in(l).l(x).l(r). \\ & (\tau.\overline{out}(l).\overline{l}(x).\overline{l}(r).Comm(in, out) \\ & +\tau.\overline{out}(l).\overline{l}(x).\overline{l}(r).\overline{out}(l).\overline{l}(x).\overline{l}(r).Comm(in, out) \\ & +\tau.Comm(in, out)) \end{aligned}$$

The simplicity of these equations makes reasoning about the overall behaviour of the resulting system straightforward.

5.3 Flattening the Hierarchy

The decomposition hierarchy of an instance of a Darwin specification is *flattened* at run-time. The process of *flattening* eliminates all composite components and their ports. Communication links then only exist between ports of primitive components. Flattening is only possible when the composite components have no processes attached since generally one can't eliminate the ports of such components. This is a required property of all Darwin specifications. Flattening is transparent to the user, i.e. the resulting system behaves exactly the same as the original hierarchical system. It is necessary to do the transformation at run-time as only then is all the required structural information available. The Darwin run-time system combines the flattening with the process of instantiation upon the start of an application.

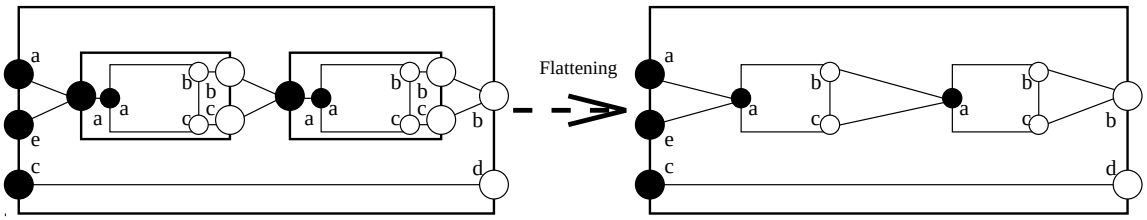

Fig. 6. Example for the Flattening of a Structure

The example in Figure 6 illustrates the flattening. It is assumed that the outermost component is the top-level component. Its ports represent the connections to the outside world. Hence they cannot disappear.

Flattening as Graph Rewriting The system structure of an instantiated Darwin specification can be represented as a directed graph whose nodes are ports and whose edges are communication links (i.e. bindings between ports). A global relabelling of the ports to obtain unique names is a necessary first step before a representation in terms of a graph becomes possible.

It should be noted that this is not a full representation of the structure of an application, as it doesn't include the primitive components (i.e. their attached processes). In fact a complete representation consists of the connection graph and a set of primitive components and the names of their ports. As the flattening doesn't affect the primitive components we only focus on the connection graph. The graph can be specified as a relation $\mathsf{conn}(\text{PORT}, \text{PORT})$ with the following axioms:

$$\mathsf{conn}(x, y) \longrightarrow \neg\mathsf{conn}(y, x) \tag{1}$$

$$\mathsf{conn}(x, y) \wedge \mathsf{conn}(x, z) \longrightarrow y = z \tag{2}$$

The first axiom ensures that the graph is directed and that no port can be connected to itself. The second axiom is enforced by the Darwin compiler to prevent multi-casting. The structure of any particular system can now be expressed in terms of a conjunction:

$$\mathit{System} \stackrel{\text{def}}{=} \bigwedge_{i=1}^{n} \mathsf{conn}(x_i, y_i)$$

Having represented the system structure as a graph we can now express the process of flattening as *graph rewriting*[BvEJ+87]. There is only one rewrite rule:

> A port can be eliminated by connecting each port that sends information to it to each port that receives data from it.

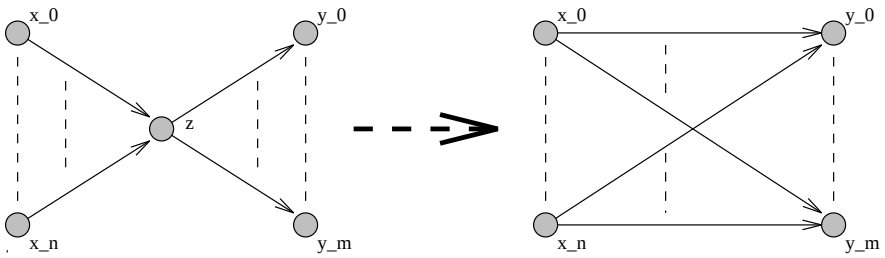

Fig. 7. The Rewrite Rule

This rule, shown in Figure 7, can be expressed formally as follows:

$$\frac{\bigwedge_{i=0}^{n} \mathsf{conn}(x_i, z) \wedge \bigwedge_{j=0}^{m} \mathsf{conn}(z, y_j) \wedge \bigwedge_{k=1}^{p} \mathsf{conn}(u_k, v_k)}{\bigwedge_{i=0}^{n} \bigwedge_{j=0}^{m} \mathsf{conn}(x_i, y_j) \wedge \bigwedge_{k=1}^{p} \mathsf{conn}(u_k, v_k)} \quad u_k \neq z, v_k \neq z$$

We only carry out one rewriting at a time. The other connections (represented by the last subterm) remain unaffected. As long as at least one such transition is possible any system specified in the form of *System* can be transformed into an expression that appears as precondition in the rewrite rule. Similarly any expression appearing as a postcondition can be transformed back into the *System* form.

In the above equation it is ensured that only ports with at least one incoming and one outgoing binding are considered for rewriting. Thereby we prevent the elimination of ports of primitive components and of the top-level component. By setting $m = 0$ we can enforces axiom (2) of the connection graph specification.

Flattening on the π-calculus Level In the π-calculus the hierarchy is represented by the scope of the nested ν-operator. After a global relabelling of the ports we can eliminate these. The purpose of the ν-operator is to both introduce fresh names and to limit their visibility. The global relabelling is only possible if the ν operator doesn't occur in a recursive structure as then the set of names grows dynamically. The translation of a meaningful instantiation of a Darwin specification doesn't include such recursion. The visibility of ports is global after the relabelling. During the process of eliminating the ν-operators, substitution functions are applied to all the processes in the system to avoid the capture of names. Thereby we ensure that processes cannot access ports they couldn't access before the relabelling. The result of this whole process is a definition in the π-calculus of the form

$$System \stackrel{\text{def}}{=} \prod_{i=1}^{n} Comm(x_i, y_i)\sigma_i \mid \prod_{j=1}^{m} Q_j\,\sigma'_j$$

where x_i and y_i are global names. Assuming that the only free variables of the communication process are those passed as parameters the substitutions σ_i can be dropped. This proposition ensures that the communication processes don't interact with the component processes and with each other by other ports than those whose names are passed as parameters. All communication systems that we've investigated earlier in this paper satisfy this proposition.

The substitution for the component processes cannot be dropped. However, for convenience we shall use a shorter notation:

$$System \stackrel{\text{def}}{=} \prod_{i=1}^{n} Comm(x_i, y_i) \mid \prod_{j=1}^{m} Q'_j \quad with\ \mathsf{fv}(Comm(x,y)) \subseteq \{x,y\}\ \ and\ \ Q'_j = Q_j\,\sigma'_j$$

The rewrite rule can then be formulated as follows:

$$\frac{\prod_{i=0}^{n} Comm(x_i, z) \mid \prod_{j=0}^{m} Comm(z, y_j) \mid \prod_{k=1}^{p} Comm(u_k, v_k) \mid \prod_{l=1}^{q} Q'_l}{\prod_{i=0}^{n}\prod_{j=0}^{m} Comm(x_i, y_j) \mid \prod_{k=1}^{p} Comm(u_k, v_k) \mid \prod_{l=1}^{q} Q'_l} \quad \begin{array}{l} u_k \neq z \\ v_k \neq z \\ z \notin \mathsf{fv}(Q'_l) \end{array}$$

The last of the side conditions ensures that no ports accessed by component processes are eliminated. While this doesn't prevent us from attaching component processes to composite components, it imposes restrictions on the nature of these processes that make their existence very unlikely. However, it is not required that all component processes have this property. It is a side condition and hence only ensures that those components are not flattened. We therefore can get a partially flattened hierarchy. Generally the flattening can be stopped at any stage, i.e. although further rewriting is possible it doesn't need to be carried out.

Lazy Instantiation If the system includes components that are lazily instantiated the flattening is interleaved with ordinary π-calculus transitions. Components can only be flattened after they've been instantiated. For dynamically instantiated components this happens when the component is being accessed for the first time. Recalling the representation of lazy instantiation from section 3.3 we see that after this initial access the structure of the resulting π-calculus sentence matches the definition of *System* in the previous section, i.e. after another relabelling further rewriting of the definition becomes possible. However, it is not an exact match. When the flattening takes place at the beginning no messages have been exchanged via the communication system. If the flattening is done dynamically messages may be on their way to other components, i.e. they are buffered in the communication system. On the π-calculus level one or more communication processes may have evolved and are not in their initial state anymore. They might never even return to that state. One example of such a process is the communication system that queues messages (cf. Section 5.2). To allow the flattening to take place we have to introduce additional rewrite rules that are specific to the model of the communication system used.

6 Summary and Future Work

We have shown how programs written in the Darwin configuration language can be translated into the π-calculus. The translation is done in two steps. The result of the first step is a program in Static Darwin, a subset of Darwin. This new language incorporates all features that are necessary for specifying any particular system configuration. It has been demonstrated how a Darwin program can be instantiated to yield a Static Darwin program. The result of the second step is to translate Static Darwin into the π-calculus. The result was then used as a basis for investigating different models for the communication system, specified in π-calculus. It's been shown how certain properties of Darwin programs can be detected at the π-calculus level.

The next step in our research will be to investigate real Darwin programs with the help of a theorem prover for the π-calculus. Also, by reversing the translation process, we will use the π-calculus as a means for specifying Darwin programs. Further research in the area of flattening and its representation as graph rewriting will be carried out to devise a strategy for applying these methods in systems with dynamic bindings and lazy instantiations in order to reduce possible communication overheads.

7 Acknowledgements

We gratefully acknowledge the advice and help provided by the Distributed Software Engineering Research Section at the Imperial College Department of Computing, and the financial support provided by the DTI under grant ref: IED4/410/36/002.

References

[BvEJ+87] H.P. Barendregt, M.C.J.D van Eekelen, J.R.W.Glaurt, J.R. Kennaway, M.J. Plasmeijer, and M.R.Sleep. Term graph rewriting. In *PARLE'87 Proceedings*, 1987.

[Dul92] N. Dulay. The Darwin configuration language. Imperial College Department of Computing Internal Report, March 1992.

[EP93] S. Eisenbach and R. Patterson. π-calculus semantics for the concurrent configuration language darwin. In *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, volume 2. IEEE Computer Society Press, 1993.

[JKD92] M.S. Sloman J. Kramer, J. Magee and N. Dulay. Configuring object based distributed programs in rex. In *IEE Software Engineering Journal*, March 1992.

[JMS89] J. Kramer J. Magee and M. Sloman. Constructing distributed programs in conic. *IEEE Transactions on Software Engineering*, 15, 1989.

[MDK93] J. Magee, N. Dulay, and J. Kramer. Structuring parallel and distributed programs. *Software Engineering Journal*, 8(2):73–82, March 1993.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS 91-180, University of Edinburgh, October 1991.

[MKD93] J. Magee, J. Kramer, and N. Dulay. Darwin/mp: An environment for parallel and distributed programming. In *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, volume 2. IEEE Computer Society Press, 1993.

[MPW89] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part i and ii. Technical Report ECS-LFCS 89-86/87, University of Edinburgh, 1989.

[RE94] M. Radestock and S. Eisenbach. What do you get from a π-calculus semantics? In *To appear in: PARLE'94 Conference Proceedings*. Springer-Verlag, 1994.

[Wal91] D. Walker. π-calculus semantics of object-oriented programming languages. In *Conference on Theoretical Aspects of Computer Software*, Tohoku University, Japan, September 1991.

A The π-calculus

The π-calculus is an elementary calculus for describing and analysing concurrent systems with evolving communication structure[MPW89, Wal91, Mil91]. The following is a brief description of the version of the calculus that has been used throughout this paper.

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 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 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.
  - $[x = y]P$ behaves like P if x and y are identical and like $\mathbf{0}$ otherwise.
  - $\mathbf{0}$ 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.

B Static Darwin Syntax

$$
\begin{array}{ll}
\mathit{DarwinSpec} & = \mathit{Component} \\
 & \mid \mathit{DarwinSpec}\ \mathit{Component} \\
\mathit{Component} & = \textbf{component}\ \mathit{Id}\ \mathit{ComponentBody} \\
\mathit{ComponentBody} & = \\
 & \mid \{\ \mathit{SentenceList}\ \} \\
\mathit{SentenceList} & = \mathit{Sentence} \\
 & \mid \mathit{SentenceList}\ \mathit{Sentence} \\
\mathit{Sentence} & = ; \\
 & \mid \textbf{provide}\ \mathit{InterfaceList}\ ; \\
 & \mid \textbf{require}\ \mathit{InterfaceList}\ ; \\
 & \mid \textbf{inst}\ \mathit{InstanceList}\ ; \\
 & \mid \textbf{bind}\ \mathit{BindClauseList}\ ; \\
\mathit{InterfaceList} & = \mathit{Interface} \\
 & \mid \mathit{InterfaceList}\ ,\ \mathit{Interface} \\
\mathit{Interface} & = \mathit{Id} \\
\mathit{InstanceList} = & = \mathit{Instance} \\
 & \mid \mathit{InstanceList}\ ;\ \mathit{Instance} \\
\mathit{Instance} & = \mathit{Id} : \mathit{Id} \\
 & \mid \mathit{Id} : \textbf{dyn}\ \mathit{Id} \\
\mathit{BindClauseList} & = \mathit{Binding} \\
 & \mid \mathit{BindClauseList}\ ;\ \mathit{Binding} \\
\mathit{Binding} & = \mathit{Port}\ \text{- -}\ \mathit{Port} \\
\mathit{Port} & = \mathit{Interface} \\
 & \mid \mathit{Id}\ .\ \mathit{Interface} \\
\mathit{Id} & = (\mathit{letter} \mid \_)\{\mathit{letter} \mid \mathit{digit} \mid \_\}
\end{array}
$$

This article was processed using the LaTeX macro package with LLNCS style