Semantics of a Higher-Order Coordination Language

Matthias Radestock and Susan Eisenbach

Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ
E-mail: {mr3,se}@doc.ic.ac.uk

Abstract. A distributed program can be viewed as a composition of three parts. Firstly, there is a coordination 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.
Darwin is a language for describing distributed configurations in terms of component types, their instantiation to components with interfaces and the binding of those interfaces. A Darwin program thus defines a class of configurations. Although the language is very small it contains second-order constructs: component types can appear as parameters in the instantiation of other component types. Furthermore Darwin provides support for dynamic run-time instantiation of components. Component types therefore must have a run-time representation.
The coordination part of a distributed program has to be closely associated with the communication and computation part, as only the combination of the three will yield the complete program. On the semantic level we can achieve this by using the same formal description technique for all three. The π-calculus can serve as such a technique. The higher-orderedness of the coordination language can be captured by using the higher-order π-calculus. The semantics gives a precise meaning to Darwin programs. It turns out that by using the higher-order π-calculus this semantics can be expressed in a very concise and clear manner.

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.

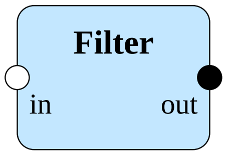
The Darwin coordination language [3, 1, 2] is an interconnection language for defining configurations of programs in distributed systems. Darwin separates the description of the system structure from the algorithms used to describe individual processes. Thereby a process instantiation and reuse is permitted 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 projects[4]. An important characteristic of the Darwin system is that it enables systems to be configured dynamically.

Milner's π-calculus[6, 5] 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 are the fundamental building blocks of the π-calculus.

In this paper we show that the π-calculus notions of processes and names have Darwin counterparts. Darwin is a higher-order language and derives much of its power from this higher-orderedness. It is therefore desirable for it to be captured in the semantics. However, attempts to define the semantics of such a language in terms of first-order definitions will either fail or produce complex definitions. The latter has a severe impact on the usefulness of the semantics for the analysis of programs. Fortunately the higher-orderedness in Darwin has a counterpart in the higher-orderedness of the higher-order π-calculus[9, 10]. This makes the higher-order π-calculus a good system for defining the semantics of Darwin. A further reason for choosing the π-calculus is the possibility of integrating the semantics of the coordination language with the semantics of the communication system and computational components of a distributed program. A π-calculus semantics of these has already been defined in our previous research[8]. The combination of the three will therefore enable us to define an integrated semantics for the whole distributed program.

2 Darwin

A distributed system consists of multiple concurrently executing and interacting computational components. The task of specifying the system as a collection of components with complex interconnection patterns quickly becomes unmanageable without the help of some structuring tools. The coordination language Darwin provides such a structuring tool. It has both a graphical and textual representation. Darwin allows distributed programs to be constructed from hierarchically structured specifications of components and their interconnections.



```
component Filter(int freq) {
    require in;
    provide out;
}
```

Fig. 1. Component type `Filter`

Composite components are constructed from the primitive computational components. Components interact by accessing services. This section gives a brief overview of Darwin.

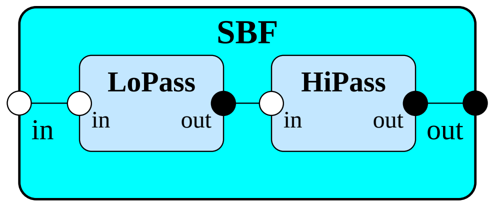
2.1 Components and Services

Darwin views components in terms of both the services they provide to allow other components to interact with them and the services they require to interact with other components. For example, the component of Fig. 1 is a *filter* component which **require**s a single service *in* and **provide**s a single service *out*. The diagrammatic convention used here is that filled in circles represent services provided by a component and empty circles represent services required by a component.

In general, a component may provide and require many services Provisions and requirements make up the *interface* of a component. It should be noted that the names of required and provided services are local to the component type specification. Components may be implemented and tested independently of the rest of the system of which they will form a part. This property is called *context independence* and permits the reuse of components during construction (through multiple instantiation) and simplifies replacement during maintenance. The example also illustrates the use of parameters of component types, e.g. *freq* in the example. These are passed to the underlying implementation of the component.

2.2 Composite Components

The primary purpose of the Darwin coordination language is to allow system architects to construct composite components from both basic computational components and other composite components. The resulting system is a hierarchically structured composite component which when elaborated at execution time results in a collection of concurrently (potentially distributed) executing computational component instances. Darwin is a declarative notation. Composite components are defined by declaring both the instances of other components they contain and the bindings between those components. Bindings, which associate the services required by one component with the services provided by others, can be visualised as filling in the empty circles of a component with the



```
component SBF(int loFreq,
              int hiFreq) {
    require in;
    provide out;

    inst lo: LoPass(loFreq);
    inst hi: HiPass(hiFreq);

    bind lo.in -- in;
    bind hi.in -- lo.out;
    bind out   -- hi.out;
}
```

Fig. 2. Composite component type `SimpleBandFilter`

solid circles provided by other components. The example of Fig. 2 defines a band filter that is implemented in terms of a chain of a low pass filter and a high pass filter. The output of the low pass is bound to the input of the high pass. The **inst** statement is used for declaring sub-components.

Bindings between requirements and provisions are declared by the **bind** statement. Requirements which cannot be satisfied inside the component can be made visible at a higher level by binding them to an interface requirement as has been done in the example for the requirement *in* of the low pass which is bound to *in*. Similarly services provided internally which are required outside are bound to an interface service provision. In general, many requirements may be bound to a single provision. A particular requirement may be bound to a single provision only. It should be noted that a service may transmit or receive information or do both. The many requirements to a single provision binding pattern may thus describe either one-to-many or many-to-one communication depending on the interaction mechanism used to implement the service.

2.3 Name Server

Provisions and requirements can be **export**ed to and **import**ed from a *name server*. This feature of Darwin enables us to establish connections between program components and external components. The name-server can be viewed as a global name-space. Exported provisions are services that a component wants to make accessible to the outside world. To achieve this it *exports* the provision under a service description to the name server. The name server keeps track of all the exported provisions. Components in all programs connected to the name server can obtain access to these services by *importing* a handle from the name server. Imports are thus similar to requirements. To obtain the interface the requiring component queries the name server with a *service specification*. The name server will try to match this specification against the exported *service*

```
component Timer {                component Stopwatch {
    export tick '1HzTick';           provide start;
}                                    provide stop;
                                     import seconds '1HzTick';
                                 }
```

Fig. 3. Example of exports and imports

descriptions. The same service can have been exported by more than one component and the name server must therefore make a selection. A handle to a service will be returned or the import will fail if there is no matching service known to the name server. The matching and selection policies can be arbitrarily complex, for instance by incorporating quality of service parameters. In the simplest case the matching function is *equality*, i.e. the required service specification has to be equal to the provided service description. The selection policy can just be random choice. A simple implementation may just queue requests for unavailable services until they become available.

In the example in Fig. 3 a `StopWatch` component is implemented by using a system `Timer` component that exports a 1Hz signal under the service description `1HzTick`. The two components could be part of different Darwin programs.

2.4 Generic Component Types

Darwin allows the specification of component type parameters that hold component types. Typically this is employed for achieving a higher degree of implementation flexibility – instead of having fixed component types for sub-components the types can be determined at the instantiation of the component, thus making the component type definition generic. The example in Fig. 4 illustrates this.

The `BandFilter` component type defines components that are capable of filtering out a frequency range of some input signal. If the lower bound of the frequency range is less or equal than the higher bound the filter will filter out all frequencies in that range. Otherwise it will filter out all frequencies outside that range. The filtering is accomplished by a combination of a low-pass and high-pass filter `LoPass` and `HiPass`. The types of these filters are determined by the context, thus enabling the configuration of `BandFilter` components with various implementations of filters. We could instantiate `BandFilter` with

```
inst filter:BandFilter(LoPass1,HiPass1,10,20) .
```

The component type specified in the parameter declaration (in this case `Filter`) is required for static type checking – only component types with the same interface can be passed in.

Darwin is a higher-order language because it provides the ability to have component types as parameters to other component types. The example also shows how conditionals can be used in Darwin to create alternative configurations. Much of the expressive power of Darwin is derived from a combination of

```
component LoPass1(
    int freq) {

    require in;
    provide out;
}

component HiPass1(
    int freq) {

    require in;
    provide out;
}

component Mixer {
    require in1;
    require in2;
    provide out;
}
```

```
component BandFilter(
    component(Filter) LoPass,
    component(Filter) HiPass,
    int loFreq, int hiFreq) {

    require in;
    provide out;

    inst lo: LoPass(loFreq);
    inst hi: HiPass(hiFreq);

    when loFreq <= hiFreq {
        inst mix: Mixer;
        bind lo.in   -- in;
        bind hi.in   -- in;
        bind mix.in1 -- lo.out;
        bind mix.in2 -- hi.out;
        bind out     -- mix.out;
    }
    when loFreq >  hiFreq {
        bind lo.in -- in;
        bind hi.in -- lo.out;
        bind out   -- hi.out;
    }
}
```

Fig. 4. Composite component type `BandFilter`

these two features. It enables the definition of component types with a very high degree of flexibility and reusability.

3 Darwin in Higher-Order π-calculus

When devising a semantics for a language, in a particular specification language, it is advantageous to have a simple mapping between the various concepts of the two languages. This significantly reduces the complexity of the semantics and simplifies reasoning. We believe that Darwin and the higher-order π-calculus are two such languages. Once the conceptual mapping has been established devising the semantics is in most cases straightforward.

3.1 Components, Instantiation and Decomposition

A *component type* definition in Darwin can be mapped to a process definition in the higher-order π-calculus. *Parameters* of the component type are treated as process parameters to the process definition. This higher-orderedness it not needed for basic type parameters but it significantly simplifies their modelling. The processes passed in represent constants as Darwin doesn't have any language constructs for manipulating variables. The higher-orderedness *is* required for the modelling of component type parameters though because component types are represented as processes. Components are created from their component types by instantiating the latter. In higher-order π-calculus this is expressed as an application of the actual parameters to the process definition of the component type.

```
component A(int x, component(B) y) ...
```

would be translated into a process equation

$$A(X, Y) \stackrel{\text{def}}{=} \ldots$$

and an instantiation

```
inst a:A(p, q)
```

would be expressed as

$$A(P, Q) \ .$$

Darwin components can be decomposed. The *decomposition* is specified in the component type definition. The instantiation of such a component type will cause all sub-components to be instantiated from their respective component types. The behaviour of the component is determined by the parallel interaction of the behaviour of the sub-components via the 'glue' – the bindings between interfaces of these components. This form of decomposition is expressed in the higher-order π-calculus as a parallel composition of all the instantiations, thus emphasising the parallelism inherent in the sub-components creation and the parallel existence and operation of these components.

```
inst a:A(p, q); b:B(r)
```

inside a component type definition can be translated into

$$A(P, Q) \ | \ B(R) \ .$$

3.2 Control Structures and Expressions

The Darwin language contains a **when** control structure which is used to make the creation of sub-components and the binding of interfaces conditional upon parameters. The interface specified by a component type is constant. All instances of a component type will thus have the same interface but possibly a different internal structure. The **when** statement evaluates a Boolean expression. All the variables, constants and operators of such an expression can be represented as processes in the higher-order π-calculus. The conditional itself is expressed as a process expression parameterised by the Boolean expression process and the processes representing the 'true' and 'false' branches. Only the higher-ordered version of the π-calculus allows us to express these constructs in such an elegant way.

```
when r<0 { inst a:A(p, q); }
```

has a higher-order π-calculus equivalent of

$$\mathit{When}(\mathit{LessThan}(R, \mathit{Zero}), A(P, Q), \mathbf{0}) \ .$$

[12, 11] provide a more detailed investigation into the definition of Booleans, Boolean functions and other standard data types.

3.3 Interfaces and Bindings

The *provisions* of a component represent services provided by that component. Other components can use these services once they have acquired a handle to them. *Requirements* represent the intention of a component to acquire such a handle. They turn into handles once a *binding* to a provision has been established through Darwin. In the higher-order π-calculus provisions and requirements can be expressed as *bound* and *unbound* names respectively. Names get bound (i.e. the requirements turn into handles) as a result of a communication. We can thus express a binding as a communication between the process representing the providing component with the process representing the requiring component.

The requirements and provisions of a Darwin component are the only interface between a component and the outside. A component encapsulates its sub-components, thus preventing direct outside access to them. During the instantiation of a component type, elements (i.e. provisions or requirements) of the sub-component interfaces can be bound to elements of interfaces of other sub-components or to elements of the interface of the containing component. This special case of *third party binding* (where the container component is the third party) is the only type of binding supported by Darwin. However, interfaces do have a run-time representation and the run-time system may thus support other types of binding. The third party binding requires the binding party to know the two other parties. This can be achieved by making the interface of components part of the left hand side of the defining process equation, i.e. making the provisions and requirements part of the parameter list. The responsibility for creating provisions and requirements then lies with the container component which is always the binding component. It must therefore be able to determine the interface of a sub-component from its component type. Hence the interface must be component type specific rather than instance specific, i.e. all instances of a component type must have the same interface. In Darwin this is enforced by disallowing the use of control structures in the interface declaration part of a component type definition.

```
component B(int z) { provide x; require y, w; }
```

is represented as

$$B(Z, x', y', w') \ \stackrel{\mathrm{def}}{=} \ (\nu\ x)(!\overline{x'}(x).\mathbf{0} \ \mid \ y'(y).w'(w).B'(Z, x, y, w)) \ .$$

Provisions and requirements are both turned into communication links. The communication takes place with the binding component which is also the component that instantiated the process. The provisions of a component are made known to the binding component by communicating them along their respective communication links. The replication operator ensures that a provision can be bound to any number of requirements. Components wait to receive bindings for their requirements from the binding component through the respective communication links. Once all these bindings have been establish the process proceeds. Only primitive components (i.e. components with no sub-components) define new provisions. The provisions of composite components are obtained from the bindings. Similarly composite components do not wait for their requirements to be bound but only forward those bindings to the sub-components when they occur. The instantiation of sub-components in the container component

```
inst b:B(r); c:B(s);
```

is translated into

$$(\nu\, x_b, y_b, w_b, x_c, y_c, w_c)(B(R, x_b, y_b, w_b) \ \mid\ B(S, x_c, y_c, w_c)) \ .$$

A binding

```
bind b.y--c.x, b.w--c.x;
```

translates to

$$x_c(a).\overline{y_b}(a).\mathbf{0} \ \mid\ x_c(a).\overline{w_b}(a).\mathbf{0} \ .$$

Bindings to the interface of the binding component, as in

```
bind p--b.x, c.y--q, r--q; ,
```

where p, q, r are elements of the interface of the binding component, are expressed as

$$x_b(a).!\overline{p'}(a).\mathbf{0} \ \mid\ q'(a).\overline{y_c}(a).\mathbf{0} \ \mid\ q'(a).\overline{r'}(a).\mathbf{0}$$

where p', q', r' are obtained from the parameter list of the process definition (cf. above). The ! replication operator ensures that the outside can make multiple bindings to the provision.

3.4 Exports and Imports

The higher-order π-calculus is well-suited for defining the behaviour of name servers. However, it might be too low level for describing complex matching and selection policies. Also the name server functionality is not defined in Darwin but rather by the particular implementation. It therefore has to be treated as a 'black box' with a well-defined interface. The simplest name server interface that still allows arbitrarily complex matching and selection policies, just contains the two methods *export* and *import*. Both methods are expected to succeed, i.e. a name server must (eventually) accept all exported provisions and imported requirements will be queued until a matching service becomes available. This precise definition of the interface enables the definition of the higher-order π-calculus semantics for the **export** and **import** clauses in Darwin.

```
export x 'service1'; import y 'service2';
```

is translated into

$$\overline{export}(x, \mathit{service1}).\mathbf{0} \ \mid \ (\nu\ c)(\overline{import}(c, \mathit{service2}).c(y).P)$$

where P is the translation of the remaining part of the surrounding component type definition. The names *export* and *import* are globally scoped and form the interface to the name-server.

The operational semantics of the π-calculus defines that a communication event takes place whenever there is an input and output along the same channel. The execution of processes is suspended until such matches occur. If there are several input and output requests along the same channel, pairs are selected nondeterministically. Comparing this semantic definition to our specification for a simple name-server above we can observe that some of the name server functionality is part of the π-calculus semantics. This isn't surprising as the calculus is name based. For the simple case we can therefore eliminate the explicit name server altogether and translate the above program fragment into

$$!\overline{\mathit{service1}}(x).\mathbf{0} \ \mid \ \mathit{service2}(y).P \ .$$

We use the exported names as communication channels and utilise the replication operator to ensure that exported provisions can be imported multiple times.

4 The Higher-Order π-calculus Semantics of Darwin

The translation from Darwin into higher-order π-calculus is carried out by the semantic function $\mathcal{P}$ :: $\mathsf{Darwin} \rightarrow \mathsf{HO}\pi$. In order to translate the instantiation of component types we require information about the types. As the type definitions can appear in any order in the program the translation requires two passes. The fist pass determines the signatures of all component types and is of type $\mathcal{S}'$ :: $\mathsf{Darwin} \rightarrow \{\mathsf{CSig}\}$. The signature contains all the formal parameters of a component type, and its interfaces and its type is thus $\mathsf{CSig} = (\mathsf{CName}, [(\mathsf{TNAME}, \mathsf{FName})], [\mathsf{PName}], [\mathsf{RName}])$. The first field is the component type name, the second field is a list of formal parameter types and identifiers and the following fields are lists of names of provisions and requirements respectively. The second pass of the translation takes both the original Darwin program and the set of component type signatures and produces the higher-order π-calculus translation: $\mathcal{T}'$ :: $\mathsf{Darwin} \rightarrow \{\mathsf{CSig}\} \rightarrow \mathsf{HO}\pi$. The semantic function $\mathcal{P}$ is just

$$\mathcal{P}[\![p]\!] = \mathcal{T}'\ p\ (\mathcal{S}'\ p) \ .$$

Because there are two passes the processing of individual component type definitions can be carried out independently from each other. We can therefore rewrite $\mathcal{P}$ to

$$\mathcal{P}[\![p]\!] = \biguplus_{d \in D} d \text{ where } L = \mathcal{P}'[\![p]\!],\ \sigma = \mathsf{mapx}\ L\ (\lambda x.\mathcal{S}[\![x]\!]), \\ D = \mathsf{mapx}\ L\ (\lambda x.\mathcal{T}[\![x]\!]_{\sigma})$$

$$
\begin{aligned}
\mathcal{S}\left[\!\!\left[\begin{array}{l}\textbf{component}\\ c(t_1\ f_1, t_2\ f_2, \ldots, t_n\ f_n)\{B\}\end{array}\right]\!\!\right] &= \begin{array}{l}(c, [(t_1, f_1), (t_2, f_2), \ldots, (t_n, f_n)], I_p, I_r)\\ \text{where } (\_, \_, I_p, I_r) = \mathcal{S}[\![B]\!]\end{array}\\
\mathcal{S}[\![\textbf{provide}\ p;\ B]\!] &= \begin{array}{l}(\bot, \bot, [p] +\!\!+\, I_p, I_r)\\ \text{where } (\_, \_, I_p, I_r) = \mathcal{S}[\![B]\!]\end{array}\\
\mathcal{S}[\![\textbf{require}\ r;\ B]\!] &= \begin{array}{l}(\bot, \bot, I_p, [r] +\!\!+\, I_r)\\ \text{where } (\_, \_, I_p, I_r) = \mathcal{S}[\![B]\!]\end{array}\\
\mathcal{S}[\![x]\!] &= \begin{array}{l}(\bot, \bot, [], [])\\ \text{where } x \text{ is any Darwin construct}\end{array}
\end{aligned}
$$

Fig. 5. Semantic function for the first pass

where $\uplus$ is a textual concatenation operator that combines individual higher-order π-calculus process definitions into one big higher-order π-calculus-'program'. The higher-order function mapx maps a function over the elements of a set or list. The function $\mathcal{S}$ has the type $\mathcal{S} :: \mathsf{CTDef} \rightarrow \mathsf{CSig}$ and $\mathcal{T}$ has the type $\mathcal{T} :: \mathsf{CTDef} \rightarrow \{\mathsf{CSig}\} \rightarrow \mathsf{HO}\pi$, i.e. they both translate a single component type definition. $\mathcal{T}$ produces a single defining equation in higher-order π-calculus for a component type definition. $\mathcal{P}' :: \mathsf{Darwin} \rightarrow \{\mathsf{CTDef}\}$ is a simple parsing function that turns a Darwin program into a set of component type definitions. We can define it as

$$
\begin{aligned}
\mathcal{P}'[\![\mathit{ctdef}]\!] &= \{\mathit{ctdef}\}\\
\mathcal{P}'[\![\mathit{prog_1}\ \mathit{prog_2}]\!] &= \mathcal{P}'[\![\mathit{prog_1}]\!] \cup \mathcal{P}'[\![\mathit{prog_2}]\!]
\end{aligned}
$$

since a Darwin program is a sequence of component type definitions and the smallest program contains just one such definition.

4.1 First Pass

The first pass of the translation of Darwin into higher-order π-calculus is accomplished by a semantic function (Fig. 5) that extracts the names of the formal parameters, provisions and requirements from a component type definition. The information is returned in a tuple of type CSig. The component type name and the names of the formal parameters can be obtained easily from the head of a component type declaration. For the names of provisions and requirements we have to scan the body of the declarations for **provide** and **require** clauses. As the interface declaration of a component type is the first part of the declaration body we terminate the function once we encounter a different Darwin construct. As a simplification we assume that the **provide** and **require** statements as well as **export**, **import**, **bind** and **inst** below only take one argument. The actual Darwin statements can normally take a list of arguments, e.g. **provide** p, q, r**;**. The expressiveness of Darwin doesn't suffer from this simplification as we can always transform such a statement into a list of statements, e.g. **provide** p**; provide** q**; provide** r**;**.

The $+\!\!+$ operator in the definition is the list concatenation operator. The names must be stored in lists rather than sets as during the second pass of the translation the elements will be identified by their position. Applying $\mathcal{S}$ to the `BandFilter` component from Fig. 4 we get the following result:

$$\mathcal{S}\left[\!\!\left[\begin{array}{l}\texttt{component BandFilter(}\\ \texttt{component(Filter) LoPass,}\\ \texttt{component(Filter) HiPass,}\\ \texttt{int loFreq, int hiFreq)}\\ \texttt{\{}\ldots\texttt{\}}\end{array}\right]\!\!\right] = \begin{array}{l}(\mathit{BandFilter}, [\\ (\texttt{component(Filter)}, \mathit{LoPass}),\\ (\texttt{component(Filter)}, \mathit{HiPass}),\\ (\texttt{int}, \mathit{loFreq}), (\texttt{int}, \mathit{hiFreq})],\\ [\mathit{out}], [\mathit{in}])\end{array}$$

4.2 Second Pass

The second pass of the translation is subdivided into two stages. The first stage splits the component type definition into a header and body part and extracts the header information containing all elements of the signature plus the exports and imports. Information provided by the first pass is only needed for the second stage which from the extracted header information, the definition body and the set of component type signatures generates a defining higher-order π-calculus equation for the component type.

$$\mathcal{T}[\![c]\!]_\sigma = \mathcal{C}[\![b]\!]_\sigma(s) \text{ where } (s, b) = \mathcal{H}[\![c]\!]$$

The first and second stage are represented by the semantic functions $\mathcal{H}$ :: $\mathsf{CTDef} \rightarrow (\mathsf{CSig}', \mathsf{Darwin})$ and $\mathcal{C}$ :: $\mathsf{Darwin} \rightarrow \{\mathsf{CSig}\} \rightarrow \mathsf{CSig}' \rightarrow \mathsf{HO}\pi$ where $\mathsf{CSig}' = (\mathsf{CSig}, \{(\mathsf{SName}, \mathsf{PName})\}, \{(\mathsf{SName}, \mathsf{RName})\})$. Unlike the names of the provisions and requirements in the signature, the exports and imports can be kept in a set rather than a list, because they're always referred to by name rather than by index. Each element of the set is a pair containing the service name and the name of the exported/imported provision/requirement.

First and Second Stage The definition of $\mathcal{H}$ (Fig. 6) resembles that of $\mathcal{S}$ (Fig. 5). As the component type definition of `BandFilter` (Fig. 4) doesn't include any exports or imports the application of $\mathcal{H}$ yields

$$\mathcal{H}\left[\!\!\left[\begin{array}{l}\texttt{component BandFilter(}\\ \texttt{component(Filter) LoPass,}\\ \texttt{component(Filter) HiPass,}\\ \texttt{int loFreq, int hiFreq)}\\ \texttt{\{}\ldots\texttt{\}}\end{array}\right]\!\!\right] = \begin{array}{l}((\mathit{BandFilter}, [\\ (\texttt{component(Filter)}, \mathit{LoPass}),\\ (\texttt{component(Filter)}, \mathit{HiPass}),\\ (\texttt{int}, \mathit{loFreq}), (\texttt{int}, \mathit{hiFreq})],\\ [\mathit{out}], [\mathit{in}]), \{\}, \{\},\\ \texttt{inst lo: LoPass(loFreq);}\ldots)\end{array}$$

The semantic function $\mathcal{C}$ (Fig. 7) translates the header information and embeds the translation of the remaining definition body. The latter is carried

$$
\begin{aligned}
\mathcal{H}\left[\!\!\left[\begin{array}{l}\textbf{component}\\ c(t_1\ f_1, t_2\ f_2, \ldots, t_n\ f_n)\\ \{B\}\end{array}\right]\!\!\right] &= \begin{array}{l}((c, [(t_1, f_1), (t_2, f_2), \ldots, (t_n, f_n)],\\ I_p, I_r), I_e, I_i, b)\\ \text{where } ((\_, \_, I_p, I_r), I_e, I_i, b) = \mathcal{H}[\![B]\!]\end{array}\\
\mathcal{H}[\![\textbf{provide}\ p;B]\!] &= \begin{array}{l}((\bot, \bot, [p] +\!\!+\ I_p, I_r), I_e, I_i, b)\\ \text{where } ((\_, \_, I_p, I_r), I_e, I_i, b) = \mathcal{H}[\![B]\!]\end{array}\\
\mathcal{H}[\![\textbf{require}\ r;B]\!] &= \begin{array}{l}((\bot, \bot, I_p, [r] +\!\!+\ I_r), I_e, I_i, b)\\ \text{where } ((\_, \_, I_p, I_r), I_e, I_i, b) = \mathcal{H}[\![B]\!]\end{array}\\
\mathcal{H}[\![\textbf{export}\ p\ s;B]\!] &= \begin{array}{l}((\bot, \bot, I_p, I_r), \{(p, s)\} \cup I_e, I_i, b)\\ \text{where } ((\_, \_, I_p, I_r), I_e, I_i, b) = \mathcal{H}[\![B]\!]\end{array}\\
\mathcal{H}[\![\textbf{import}\ r\ s;B]\!] &= \begin{array}{l}((\bot, \bot, I_p, I_r), I_i, \{(r, s)\} \cup I_i, b)\\ \text{where } ((\_, \_, I_p, I_r), I_e, I_i, b) = \mathcal{H}[\![B]\!]\end{array}\\
\mathcal{H}[\![x;B]\!] &= \begin{array}{l}((\bot, \bot, [], []), \{\}, \{\}, x;b)\\ \text{where } (\_, \_, \_, b) = \mathcal{H}[\![B]\!]\\ \text{where } x \text{ is any Darwin construct}\end{array}\\
\mathcal{H}[\![x]\!] &= \begin{array}{l}((\bot, \bot, [], []), \{\}, \{\}, x)\\ \text{where } x \text{ is any Darwin construct}\end{array}
\end{aligned}
$$

Fig. 6. Semantic function for the first stage of the second pass

out by the semantic function $\mathcal{B}$ (Fig. 8). We use a Roman font for literal π-calculus names, e.g. C and F. The indices are variables that will be replaced with their real values, e.g. F_{f_1} will turn into $\mathrm{F}_{\texttt{param1}}$ if f_1 gets bound to `param1` in the equation. The equation in Fig. 7 may seem complicated, but the generated higher-order π-calculus definitions are actually rather small, as the `Filter` example illustrates:

$$
\mathrm{C}_{\texttt{Filter}}(\mathrm{F}_{\texttt{freq}}, \mathrm{out}', \mathrm{in}') \overset{\text{def}}{=} (\nu\ \mathrm{out})(!\overline{\mathrm{out}'}(\mathrm{out}).\mathbf{0}\ \mid \\ \mathrm{in}'(\mathrm{in}).\mathrm{C}_{\texttt{Filter}}{}'(\mathrm{F}_{\texttt{freq}}, \mathrm{out}, \mathrm{in}))
$$

The definition of $\mathcal{C}$ captures the distinction between primitive and composite components in Darwin. A component is primitive if and only if the body of the component type definition contains only **provide**, **require**, **export** and **import** statements. As the first stage of the second pass parses exactly those constructs the second stage is invoked with an empty remaining component type definition body, denoted by $\bot$. The semantic function produces a reference to a process definition C'_c with the same signature as the process definition of the component type. C'_c identifies the π-calculus translation of the computational part of the primitive component. From the definition we can see that provisions, exports and imports are treated in parallel while requirements form a sequential composition. This is to ensure that the required names are bound in the remainder of the formula. The computational part of a primitive component is only invoked

$$\begin{array}{l}
\mathcal{C}[\![b]\!]_\sigma(s) = \mathrm{C}_c(\mathrm{F}_{f_1}, \mathrm{F}_{f_2}, \ldots, \mathrm{F}_{f_{n_f}}, {p_1}', {p_2}', \ldots, {p_{n_p}}', {r_1}', {r_2}', \ldots, {r_{n_r}}') \quad \overset{\mathrm{def}}{=} \\
\textsf{if } b{=}\bot \textsf{ then } (\nu\, p_1, p_2, \ldots, p_{n_p})(!\overline{{p_1}'}(p_1).\mathbf{0} \mid !\overline{{p_2}'}(p_2).\mathbf{0} \mid \ldots \mid !\overline{{p_{n_p}}'}(p_{n_p}).\mathbf{0} \mid \\
\qquad\qquad {e_1}'(x).!\overline{e'_1}(x).\mathbf{0} \mid {e_2}'(x).!\overline{e'_2}(x).\mathbf{0} \mid \ldots \mid {e_{n_e}}'(x).!\overline{e'_{n_e}}(x).\mathbf{0} \mid \\
\qquad\qquad {i_1}'(x).!\overline{i'_1}(x).\mathbf{0} \mid {i_2}'(x).!\overline{i'_2}(x).\mathbf{0} \mid \ldots \mid {i_{n_e}}'(x).!\overline{i'_{n_e}}(x).\mathbf{0} \mid \\
\qquad\qquad {r_1}'(r_1).{r_2}'(r_2).\ \ldots\ .{r_{n_r}}'(r_{n_r}). \\
\qquad\qquad \mathrm{C}'_c(\mathrm{F}_{f_1}, \mathrm{F}_{f_2}, \ldots, \mathrm{F}_{f_{n_f}}, p_1, p_2, \ldots, p_{n_p}, r_1, r_2, \ldots, r_{n_r})) \\
\qquad\quad \textsf{else } {e_1}'(x).!\overline{e'_1}(x).\mathbf{0} \mid {e_1}'(x).!\overline{e'_2}(x).\mathbf{0} \mid \ldots \mid {e_{n_e}}'(x).!\overline{e'_{n_e}}(x).\mathbf{0} \mid \\
\qquad\qquad {i_1}'(x).!\overline{i'_1}(x).\mathbf{0} \mid {i_2}'(x).!\overline{i'_2}(x).\mathbf{0} \mid \ldots \mid {i_{n_e}}'(x).!\overline{i'_{n_e}}(x).\mathbf{0} \mid \\
\qquad\qquad \mathcal{B}[\![b]\!]_\sigma(s) \\
\text{where } ((c, [(t_1, f_1), (t_2, f_2), \ldots, (t_{n_f}, f_{n_f})], [p_1, p_2, \ldots, p_{n_p}], [r_1, r_2, \ldots, r_{n_r}]), \\
\qquad \{(e'_1, e_1), (e'_2, e_2), \ldots, (e'_{n_e}, e_{n_e})\}, \{(i'_1, i_1), (i'_2, i_2), \ldots, (i'_{n_i}, i_{n_i})\}) = s
\end{array}$$

Fig. 7. Semantic function for the second stage of the second pass

after all requirements have been bound, i.e. turned into handles. In case the component is composite the translation of the remaining definition body is processed in parallel with the exports and imports. Requirements are bound inside the translation of the definition body – and only if they are used in a binding. This ensures that the instantiation of component types doesn't get 'stuck' in cases where requirements are bound to provisions of the same component.

Translation of the Definition Body $\mathcal{B} :: \textsf{Darwin} \rightarrow \{\textsf{CSig}\} \rightarrow \textsf{CSig}' \rightarrow \textsf{HO}\pi$ (Fig. 8) uses the ideas about the mapping of concepts from Darwin to higher-order π-calculus (cf. Sect. 3) in order to translate the remainder of the component type definition body. The most complicated of the defining equations is the one relating to the instantiation of component types. It is the place where the higher-orderedness of the higher-order π-calculus is needed in order to obtain a concise and intuitive definition.

It would be sufficient to pass in the component type name instead of the $\textsf{CSig}'$ structure s. We could then lookup the signature in σ. However, we would then not be able to verify whether an element in a **bind** statement has been correctly declared as a provision, requirement or import, as $\textsf{CSig}$ doesn't contain information about exports and imports. Furthermore it would obscure what σ is *actually* needed for – to lookup the signatures of component types that are being instantiated.

There is the special case that the component type in the **inst** statement is not specified as a type identifier but is contained in a variable that has been passed in as a parameter to the current component. We identify this case by checking whether the component type part of the **inst** statement is contained in the parameter list of the current component type. The latter is obtained from

$$
\begin{aligned}
\mathcal{B}\left[\!\!\left[\begin{matrix}\textbf{inst } c{:}t(e_1, e_2,\\ \ldots, e_n);\ B\end{matrix}\right]\!\!\right]_\sigma(s) = {} & (\nu\ p_{1c}, p_{2c}, \ldots, p_{ic}, r_{1c}, r_{2c}, \ldots, r_{jc})\\
& (C(a_1, a_2, \ldots, a_n, p_{1c}, p_{2c}, \ldots p_{ic},\\
& r_{1c}, r_{2c}, \ldots r_{jc}) | \mathcal{B}[\![B]\!]_\sigma(s))\\
& \text{where } (t', \_, [p_1, p_2, \ldots, p_i], [r_1, r_2, \ldots, r_j]) \in \sigma\\
& \qquad t' = \mathsf{if}\ t \in Q\ \mathsf{then}\ t''\ \mathsf{else}\ t\\
& \qquad (\textbf{component}(t''), t) \in P\\
& \qquad \{(t_1, f_1), (t_2, f_2), \ldots, (t_m, f_m)\} = P\\
& \qquad Q = \{f_1, f_2, \ldots, f_m\}\\
& \qquad C = \mathsf{if}\ t \in Q\ \mathsf{then}\ \mathrm{F}_t\ \mathsf{else}\ \mathrm{C}_t\\
& \qquad ((\_, P, \_, \_), \_, \_) = s\\
& \qquad [a_1, a_2, \ldots, a_n] = \mathsf{mapx}\ [e_1, e_2, \ldots e_n]\\
& \qquad\qquad\qquad\qquad (\lambda x.\mathcal{E}[\![x]\!]P)\\
\mathcal{B}[\![\textbf{bind } c_1.r{-}c_2.p;\ B]\!]_\sigma(s) = {} & p_{c_2}(x).\overline{r_{c_1}}(x).\mathbf{0}\ \mid\ \mathcal{B}[\![B]\!]_\sigma(s)\\
\mathcal{B}[\![\textbf{bind } p_0{-}c.p;\ B]\!]_\sigma(s) = {} & p_c(x).!\overline{p_0'}(x).\mathbf{0}\ \mid\ \mathcal{B}[\![B]\!]_\sigma(s)\\
\mathcal{B}[\![\textbf{bind } c.r{-}r_0;\ B]\!]_\sigma(s) = {} & r_0'(x).\overline{r_c}(x).\mathbf{0}\ \mid\ \mathcal{B}[\![B]\!]_\sigma(s)\\
\mathcal{B}[\![\textbf{bind } r{-}p;\ B]\!]_\sigma(s) = {} & p'(x).\overline{r'}(x).\mathbf{0}\ \mid\ \mathcal{B}[\![B]\!]_\sigma(s)\\
\mathcal{B}\left[\!\!\left[\begin{matrix}\textbf{when}\, expr\\ \{block\};\ B\end{matrix}\right]\!\!\right]_\sigma(s) = {} & \mathrm{When}(\mathcal{E}[\![expr]\!]P, \mathcal{B}[\![block;B]\!]_\sigma(s), \mathcal{B}[\![B]\!]_\sigma(s))\\
& \text{where } ((\_, P, \_, \_), \_, \_) = s
\end{aligned}
$$

Fig. 8. semantic function for the translation of the definition body

the CSig' structure s. We use the signature of the component type specified in the type part of the parameter for determining the parameters of the instantiation. The higher-order π-calculus parameter identifier F_t is used for identifying the process to be instantiated. The higher-ordered constructs of the calculus are employed, because F_t is a *process variable* inside the process of the current component type which gets bound by the instantiation of that type. The instantiation parameters can be C expressions. $\mathcal{E}$ evaluates these expressions in order to obtain the values for the actual parameters.

The **bind** statement occurs in four variations and the semantic function accordingly has a defining equation for each of them. Finally there is the semantics of the **when** statement where *When* is the name of process that takes three processes as an argument. The first process is expected to represent a Boolean. If it is *true*, *When* evolves into the process specified in the second argument. Otherwise it evolves into the process specified by the third argument.

In applying $\mathcal{C}$ and $\mathcal{B}$ to the `BandFilter` component type we obtain its defining equation. For sake of clarity we have split the definition into four equations and

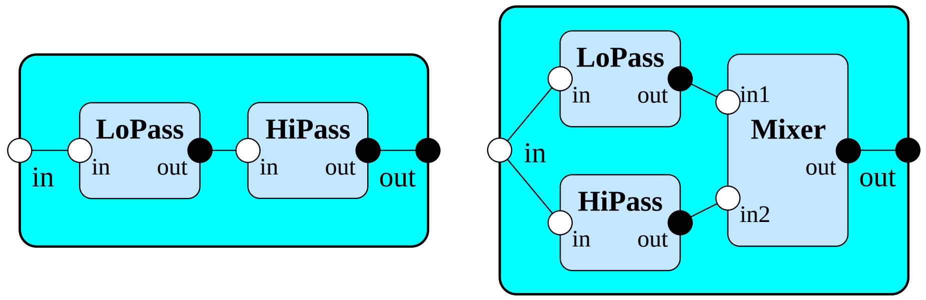


Fig. 9. Possible configurations of `BandFilter` components

carried out some optimisation on the processing of the **when** statements:

$$\begin{aligned}
\mathrm{C}_{\mathtt{BandFilter}}(&\mathrm{F}_{\mathtt{LoPass}}, \mathrm{F}_{\mathtt{HiPass}}, & \overset{\mathrm{def}}{=} \quad & (\nu\ \mathrm{out}_{\mathtt{lo}}, \mathrm{in}_{\mathtt{lo}})(\mathrm{F}_{\mathtt{LoPass}}(\mathrm{F}_{\mathtt{loFreq}}, \mathrm{out}_{\mathtt{lo}}, \mathrm{in}_{\mathtt{lo}}) \ \mid \\
&\mathrm{F}_{\mathtt{loFreq}}, \mathrm{F}_{\mathtt{hiFreq}}, & & (\nu\ \mathrm{out}_{\mathtt{hi}}, \mathrm{in}_{\mathtt{hi}})(\mathrm{F}_{\mathtt{HiPass}}(\mathrm{F}_{\mathtt{hiFreq}}, \mathrm{out}_{\mathtt{hi}}, \mathrm{in}_{\mathtt{hi}}) \ \mid \\
&\mathrm{out}', \mathrm{in}') & & \mathrm{When}(\mathrm{LessThanOrEqual}(\mathrm{F}_{\mathtt{loFreq}}, \mathrm{F}_{\mathtt{hiFreq}}), \\
& & & \qquad \mathrm{C}^{1}_{\mathtt{BandFilter}}, \mathrm{C}^{2}_{\mathtt{BandFilter}})))
\end{aligned}$$

$$\begin{aligned}
\mathrm{C}^{1}_{\mathtt{BandFilter}} \quad & \overset{\mathrm{def}}{=} \quad (\nu\ \mathrm{out}_{\mathtt{mix}}, \mathrm{in1}_{\mathtt{mix}}, \mathrm{in2}_{\mathtt{mix}})(\mathrm{C}_{\mathtt{Mixer}}(\mathrm{out}_{\mathtt{mix}}, \mathrm{in1}_{\mathtt{mix}}, \mathrm{in2}_{\mathtt{mix}}) \ \mid \\
& \qquad \mathrm{in}'(x).\overline{\mathrm{in}_{\mathtt{lo}}}(x).\mathbf{0} \ \mid \ \mathrm{in}'(x).\overline{\mathrm{in}_{\mathtt{hi}}}(x).\mathbf{0} \ \mid \\
& \qquad \mathrm{out}_{\mathtt{lo}}(x).\overline{\mathrm{in1}_{\mathtt{mix}}}(x).\mathbf{0} \ \mid \ \mathrm{out}_{\mathtt{hi}}(x).\overline{\mathrm{in2}_{\mathtt{mix}}}(x).\mathbf{0} \ \mid \\
& \qquad \mathrm{out}_{\mathtt{mix}}(x).!\overline{\mathrm{out}'}(x).\mathbf{0} \ \mid \ \mathrm{C}^{2}_{\mathtt{BandFilter}}) \\
\mathrm{C}^{2}_{\mathtt{BandFilter}} \quad & \overset{\mathrm{def}}{=} \quad \mathrm{When}(\mathrm{GreaterThan}(\mathrm{F}_{\mathtt{loFreq}}, \mathrm{F}_{\mathtt{hiFreq}}), \mathrm{C}^{3}_{\mathtt{BandFilter}}, \mathbf{0}) \\
\mathrm{C}^{3}_{\mathtt{BandFilter}} \quad & \overset{\mathrm{def}}{=} \quad \mathrm{in}'(x).\overline{\mathrm{in}_{\mathtt{lo}}}(x).\mathbf{0} \ \mid \ \mathrm{out}_{\mathtt{lo}}(x).\overline{\mathrm{in}_{\mathtt{hi}}}(x).\mathbf{0} \ \mid \\
& \qquad \mathrm{out}_{\mathtt{hi}}(x).!\overline{\mathrm{out}'}(x).\mathbf{0} \ \mid \ \mathbf{0}
\end{aligned}$$

5 Configurations

The Darwin compiler translates a Darwin specification (i.e. the coordination part of a program) into a language that can relate both to the process definitions of the primitive components (i.e. the computation part) and a run-time system specification (i.e. the communication part). Linked together they form an executable program. The Darwin specification and process definitions are parameterised. A Darwin program specifies a *class* of possible actual *configurations*. Initial parameters supplied at program startup determine which of these configurations will be established. For instance the instantiation of the `BandFilter` component type from Fig. 4 could result in two structurally different configurations (Fig. 9).

A configuration is created from a Darwin program by instantiating exactly one component type. The execution parameters supplied to the program identify this component type and its instantiation parameters. The instantiation of this top-level component will cause its sub-components to be created as well. The instantiation process continues recursively until component types of *primitive components* are encountered. These are types of non-composite components and hence they only specify interfaces. Unlike types for composite components, types of primitive components can be associated with *behaviour descriptions*, which are programs that during its execution can communicate with other programs along the bound interfaces of the primitive components. The behaviour descriptions are separated from the configuration descriptions in order to achieve a clear division of the structural from the computational aspects of a distributed program.

The execution of a Darwin program can thus be divided into two stages – instantiation and computation. *Instantiation* creates a configuration from a Darwin program and *computation* is the execution of programs associated with the primitive components in the thus established configuration. These programs communicate with each other using system and application domain specific Darwin communication libraries. Ultimately we are interested in the semantics of configurations because they, in combination with the communication and computation part, determine the behaviour of a program. The higher-order π-calculus semantics of Darwin is executable by applying the rewrite rules of the π-calculus. This elaboration covers the instantiation stage of a program and results in a configuration description, thus providing a smooth transition to the second stage of the execution.

6 Summary and Future Work

We have shown how the semantics of a coordination language can be defined in an elegant and concise way by using the higher-order π-calculus. The higher-orderedness of Darwin has been captured by using higher-order constructs of the calculus. A common problem when defining the semantics of a language is that concepts of the language are lost or at least not readily visible on the semantic level. However, we found a close correspondence between Darwin and higher-order π-calculus. The semantics can therefore retain many of the concepts of the language. Darwin is a language for describing configurations, i.e. *structural* aspects of a distributed application. Darwin programs make this structure explicit. The same can be said about the higher-order π-calculus semantics. This significantly simplifies reasoning on the semantic level. We are currently investigating this opportunity by employing automated tools such as PICT[7].

A further feature of the semantics is that it can be executed by applying the rewrite rules of the calculus. Execution means instantiation of a component type. The result is a configuration, described in terms of the higher-order π-calculus. This enables the integration of the semantics of the configuration, communication and computation part of a program, because the semantics of the latter two has been defined in terms of the π-calculus already (cf. [8]). Consequently the entire distributed application, with all it's aspects can be given a unified higher-order π-calculus semantics. We can reason about each part separately or in conjunction with the other parts. Because the calculus is executable we can even use it for simulation and experimental implementation of applications. This is particularly important for investigating the impact of changes in the coordination language or the communications system on the execution of a program. We are planning to use PICT and other higher-order π-calculus tools for that purpose.

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 EPSRC under grant ref: GR/K73282.

References

1. S. Eisenbach and R. Paterson. π-calculus semantics for the concurrent configuration language darwin. In *Proc. of the 26th Annual Hawaii Int. Conf. on System Sciences*, volume 2. IEEE Computer Society Press, 1993.
2. J. Kramer, J. Magee, M. Sloman, and N. Dulay. Configuring object-based distributed programs in REX. *IEE Software Engineering Journal*, 7(2):139–149, March 1992.
3. J. Magee, J. Kramer, and N. Dulay. Darwin/mp: An environment for parallel and distributed programming. In *Proc. of the 26th Annual Hawaii Int. Conf. on System Sciences*, volume 2. IEEE Computer Society Press, 1993.
4. J. Magee, J. Kramer, and M. Sloman. Constructing distributed programs in conic. *IEEE Transactions on Software Engineering*, 15(6), 1989.
5. R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS 91-180, University of Edinburgh, October 1991.
6. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100:1–77, 1992. Also as Tech. Rep. ECS-LFCS 89-85/86, University of Edinburgh.
7. Benjamin Pierce, Didier Rémy, and David Turner. PICT: A typed, higher-order concurrent programming language based on the π-calculus, 1994. Available by anonymous FTP from `pub/bcp` on `ftp.dcs.ed.ac.uk`.
8. M. Radestock and S. Eisenbach. What do you get from a π-calculus semantics? In *Proc. of PARLE'94 Parallel Architectures and Languages Europe*, number 817 in Lecture Notes in Computer Science, pages 635–647. Springer-Verlag, 1994.
9. Davide Sangiorgi. From π-calculus to higher-order π-calculus – and back. Technical report, Computer Science Dept., University of Edinburgh, 1992.
10. Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis, Computer Science Dept., University of Edinburgh, May 1993.
11. D. Walker. Some results on the π-calculus. In *Concurrency: Theory, Language, and Architecture*, Oxford, UK, September 1989.
12. D. Walker. π-calculus semantics of object-oriented programming languages. In *Conf. on Theoretical Aspects of Computer Software*, Tohoku University, Japan, September 1991.