Communicating Process Architectures 2008
P.H. Welch et al. (Eds.)
IOS Press, 2008

Types, Orthogonality and Genericity: Some Tools for Communicating Process Architectures

Samson ABRAMSKY

Oxford University Computing Laboratory,
Wolfson Building, Parks Road, Oxford OX1 3QD, U.K.

samson@comlab.ox.ac.uk

Abstract. We shall develop a simple and natural formalization of the idea of *client-server architectures*, and, based on this, define a notion of *orthogonality* between clients and servers, which embodies strong correctness properties, and exposes the rich logical structure inherent in such systems. Then we generalize from pure clients and servers to *components*, which provide some services to the environment, and require others from it. We identify the key notion of *composition* of such components, in which some of the services required by one component are supplied by another. This allows complex systems to be built from ultimately simple components. We show that this has the logical form of the *Cut rule*, a fundamental principle of logic, and that it can be enriched with a suitable notion of *behavioural types* based on orthogonality, in such a way that correctness properties are preserved by composition. We also develop the basic ideas of how logical constructions can be used to develop *structured interfaces* for systems, with operations corresponding to logical rules. Finally, we show how the setting can be enhanced, and made more robust and expressive, by using *names* (as in the π-calculus) to allow clients to bind dynamically to generic instances of services.

Keywords. client, server, type, orthogonality, genericity, interaction.

Introduction

Concurrent programming has been a major topic of study in Computer Science for the past four decades. It is coming into renewed prominence currently, for several reasons:

- As the remit of Moore's law finally runs out, further performance increases must be sought from multi-core architectures.
- The spread of web-based applications, and of mobile, global and ubiquitous computing, is making programming with multiple threads, across machine boundaries, and interacting with other such programs, the rule rather than the exception.

These trends raise major questions of *programmability*: how can high-quality software be produced reliably in these demanding circumstances? Communicating process architectures, which offer general forms, templates and structuring devices for building complex systems out of concurrent, communicating components, have an important part to play in addressing this issue.

Hot Topics and Timeless Truths

Our aim in this paper is to explore some general mechanisms and structures which can be used for building communicating process architectures. We shall show how ideas from logic and semantics can play a very natural rôle here.

The paper is aimed at the broad CPA community with some prior exposure to formal methods, so we shall try to avoid making the technical development too heavy, and emphasize the underlying intuitions.

Our plan is as follows:

- We shall develop a simple and natural formalization of the idea of *client-server architectures*, and, based on this, define a notion of *orthogonality* between clients and servers, which embodies strong correctness properties, and exposes the rich logical structure inherent in such systems.
- Then we generalize from pure clients and servers to *components*, which provide some services to the environment, and require others from it. We identify the key notion of *composition* of such components, in which some of the services required by one component are supplied by another. This allows complex systems to be built from ultimately simple components. We show that this has the logical form of the *Cut rule*, a fundamental principle of logic, and that it can be enriched with a suitable notion of *behavioural types* based on orthogonality, in such a way that correctness properties are preserved by composition.
- We also develop the basic ideas of how logical constructions can be used to develop *structured interfaces* for systems, with operations corresponding to logical rules.
- Finally, we show how the setting can be enhanced, and made more robust and expressive, by using *names* (as in the π-calculus) to allow clients to bind dynamically to generic instances of services.

We shall build extensively on previous work by the author and his colleagues, and others. Some references are provided in the final section of the paper. Our main aim in the present paper is to find a simple and intuitive level of presentation, in which the formal structures flow in a natural and convincing fashion from the intuitions and the concrete setting of communicating process architectures.

1. Background: Clients and Servers

Our basic structuring paradigm is quite natural and familiar: we shall view interactions as occurring between *clients* and *servers*. A server is a process which offers services to its environment; a client makes uses of services.[1] Moreover, we will assume that services are stylized into a *procedural interface*: a service interface is structured into a number of *methods*, and each use of a method is initiated by a *call*, and terminated by a *return*. Thus service methods are very much like methods in object-oriented programming; and although we shall not develop an extensive object-oriented setting for the ideas we shall explore here, they would certainly sit comfortably in such a setting. However, our basic premise is simply that client-server interactions are structured into procedure-like call-return interfaces.

[1] At a later stage, we shall pursue the obvious thought that in a sufficiently global perspective, a given process may be seen as both a client and a server in various contexts; but it will be convenient to start with a simple-minded but clear distinction between the two, which roughly matches the standard network architecture concept.

1.1. Datatypes and Signatures

Each service method call and return will have some associated parameters. We assume a standard collection of basic data types, *e.g.* **int**, **bool**. The type of a service method m will then have the form

$$m : A_1 \times \cdots \times A_n \longrightarrow B_1 \times \cdots \times B_m$$

where each A_i and B_j is a basic type. We allow $n = 0$ or $m = 0$, and write **unit** for the empty product, which we can think of as a standard one-element type.

Examples

- A `integer counter` service has a method

  $$\mathtt{inc} : \mathbf{unit} \longrightarrow \mathbf{nat}$$

  It maintains a local counter; each time the method is invoked, the current value of the counter is returned, and the counter is incremented.
- A `stack` service has methods

  $$\begin{array}{ll} \mathtt{push} : & D \longrightarrow \mathbf{unit} \\ \mathtt{pop} : & \mathbf{unit} \longrightarrow \mathbf{unit} \\ \mathtt{top} : & \mathbf{unit} \longrightarrow D \end{array}$$

A *signature* Σ is a collection of such named, typed methods:

$$\Sigma = \{m_1 : T_1, \ldots, m_k : T_k\} .$$

A signature defines the interface presented by a service or collection of services to an environment of potential clients.

A more refined view of such an interface would distinguish a number of service types, each with its own collection of methods. This would be analogous to introducing *classes* as in object-oriented programming. We shall refrain from introducing this refined structure, since the notions we wish to explore can already be defined at the level of “flat” collections of methods.

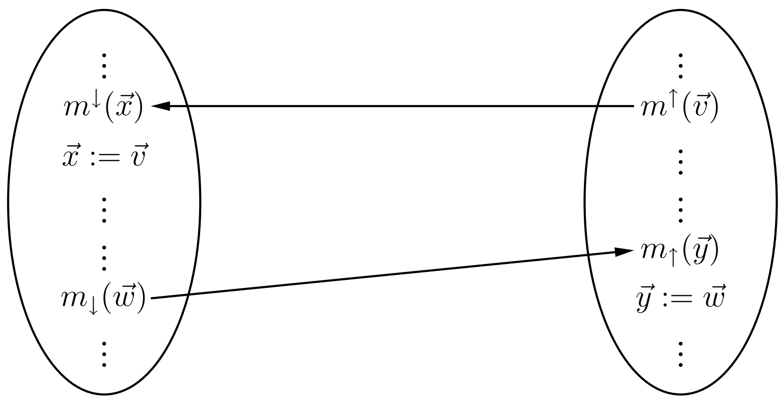
1.2. Client-Server Interactions

Although we have written service method types in a convenient functional form

$$m : A_1 \times \cdots \times A_n \longrightarrow B_1 \times \cdots \times B_m$$

a method interaction really consists of two separate *events*, each of which consists of a *synchronized interaction* between a client and server. From the point of view of a server offering a method m, it will offer a call $m^{\downarrow}(x_1, \ldots, x_n)$ to the environment. Here the variables $x_1, \ldots, x_n$ will be bound by the call, and used in the subsequent computation to perform the service method. A client performing a method call on m is trying to perform an action $m^{\uparrow}(v_1, \ldots, v_n)$ with actual values for the parameters. The method is actually invoked when such a call by the client synchronizes with a matching call offer by the server.

Dually, a service return involves the synchronization of an action $m_{\downarrow}(w_1, \ldots, w_m)$ by the server with an action $m_{\uparrow}(y_1, \ldots, y_m)$ by the client, which binds the values returned by the server for future use in the client.



Whereas in a sequential program, the caller simply waits after the call for the return, and dually the subroutine simply waits to be invoked, here we are assuming a concurrent world where multiple threads are running in both the server and the client. We should really think of the 'server' and 'client' as parallel compositions

$$s = \Big|\Big|_{i \in I} s_i, \qquad\qquad c = \Big|\Big|_{j \in J} c_j$$

where each s_i is offering one or more service methods. Thus multiple methods are being offered concurrently by the servers, and are being called concurrently by the clients.

To proceed further, we need some formal representation of processes. We shall not need to delve into the details of process algebra, nor do we need to commit ourselves to a particular choice from the wide range of process models and equivalences which have been proposed. Nevertheless, for definiteness we shall assume that the behaviour of processes is specified by *labelled transition systems* [21], which can fairly be described as the workhorses of concurrency theory.

Recall that a labelled transition system on a set of labels $\mathcal{L}$ is a structure (Q, R) where Q is a set of states, and $R \subseteq Q \times \mathcal{L} \times Q$ is the transition relation. We write $p \xrightarrow{a} q$ for $(p, a, q) \in R$. We assume, in a standard fashion, that $\mathcal{L}$ contains a *silent* or *unobservable* action τ [21]; transitions $p \xrightarrow{\tau} q$ represent internal computational steps of a system, without reference to its environment.

Now fix a server signature Σ. We define

$$\Sigma_s = \{m^{\downarrow}(\vec{x}), m_{\downarrow}(\vec{w}) \mid m \in \Sigma\}$$

the set of server-side actions over Σ. Dually, we define

$$\Sigma_c = \{m^{\uparrow}(\vec{v}), m_{\uparrow}(\vec{y}) \mid m \in \Sigma\}$$

the set of client-side actions over Σ.

A Σ-*server* is a process defined by a labelled transition system over a set of labels $\mathcal{L}$ such that $\Sigma_s \subseteq \mathcal{L}$ and $\Sigma_c \cap \mathcal{L} = \varnothing$. A Σ-*client* is a process defined by a labelled transition system over a set of labels $\mathcal{M}$ such that $\Sigma_c \subseteq \mathcal{M}$ and $\Sigma_s \cap \mathcal{L} = \varnothing$.

We shall now define an operation which expresses how a client interacts with a server. Let s be a Σ-server, and c a Σ-client. We define the behaviour of $s \lhd_\Sigma c$ by specifying its labelled transitions. Firstly, we specify that for actions outside the interface Σ, s and c proceed concurrently, without interacting with each other:

$$\frac{s \xrightarrow{\lambda} s'}{s \lhd_\Sigma c \xrightarrow{\lambda} s' \lhd_\Sigma c} \; (\lambda \notin \Sigma_s) \qquad\qquad \frac{c \xrightarrow{\lambda} c'}{s \lhd_\Sigma c \xrightarrow{\lambda} s \lhd_\Sigma c'} \; (\lambda \notin \Sigma_c)$$

Then we give the expected rules describing the interactions corresponding to method calls and returns:

$$\frac{s \xrightarrow{m^{\downarrow}(\vec{x})} s' \qquad c \xrightarrow{m^{\uparrow}(\vec{v})} c'}{s \lhd_{\Sigma} c \xrightarrow{m^{\downarrow\uparrow}(\vec{x}:\vec{v})} s'[\vec{v}/\vec{x}] \lhd_{\Sigma} c'} \qquad\qquad \frac{s \xrightarrow{m_{\downarrow}(\vec{w})} s' \qquad c \xrightarrow{m_{\uparrow}(\vec{y})} c'}{s \lhd_{\Sigma} c \xrightarrow{m_{\downarrow\uparrow}(\vec{w}:\vec{y})} s' \lhd_{\Sigma} c'[\vec{w}/\vec{y}]}$$

Here $s'[\vec{v}/\vec{x}]$ denotes the result of binding the values $\vec{v}$ to the variables $\vec{x}$ in the continuation process s', and similarly for $c'[\vec{w}/\vec{y}]$.

2. Orthogonality

Intuitively, there is an obvious *duality* between clients and servers; and where there is duality, there should be some *logical structure*. We now take a first step towards articulating this structure, which we shall use to encapsulate the key *correctness properties* of a client-server system.

We fix a server signature Σ. Let s be a Σ-server, and c a Σ-client. A *computation* of the server-client system $s \lhd_{\Sigma} c$ is a sequence of transitions

$$s \lhd_{\Sigma} c \xrightarrow{\lambda_1} s_1 \lhd_{\Sigma} c_1 \xrightarrow{\lambda_2} \cdots \xrightarrow{\lambda_k} s_k \lhd_{\Sigma} c_k \, .$$

We shall only consider finite computations.

For every such computation, we have an associated *observation*, which is obtained by concatenating the labels $\lambda_1 \cdots \lambda_k$ on the transitions. For each method $m \in \Sigma$, we obtain an *m-observation* by erasing all labels in this sequence other than those of the form $m^{\downarrow\uparrow}(\vec{x} : \vec{v})$ and $m_{\downarrow\uparrow}(\vec{w} : \vec{y})$, and replacing these by $m^{\downarrow\uparrow}$ and $m_{\downarrow\uparrow}$ respectively.

We say that $s \lhd_{\Sigma} c$ is *Σ-active* if for some computation starting from $s \lhd_{\Sigma} c$, and some $m \in \Sigma$, the corresponding m-observation is non-empty. A similar notion applies to clients c considered in isolation.

We say that s *is orthogonal to* c, written $s \perp c$, if for every computation

$$s \lhd_{\Sigma} c \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_k} s_k \lhd_{\Sigma} c_k$$

the following conditions hold:

Σ-receptiveness. If c_k is Σ-active, then so is $s_k \lhd_{\Sigma} c_k$.

Σ-completion. The computation can be extended to

$$s \lhd_{\Sigma} c \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_k} s_k \lhd_{\Sigma} c_k \cdots \xrightarrow{\lambda_{k+l}} s_{k+l} \lhd_{\Sigma} c_{k+l}$$

with $l \geq 0$, whose m-observation o_m, for each $m \in \Sigma$, is a sequence of alternating calls and returns of m:

$$o_m \in (m^{\downarrow\uparrow} m_{\downarrow\uparrow})^* \, .$$

This notion of orthogonality combines several important correctness notions, "localized" to the client-server interface:

- It incorporates a notion of *deadlock-freedom*. Σ-receptiveness ensures that whenever the client wishes to proceed by calling a service method, a matching offer of such a service must eventually be available from the server. Σ-completion ensures that every method execution which is initiated by a call must be completed by the corresponding return. Note that, since this condition is applied along all computations, it takes account of non-deterministic branching by the server and the client; no matter how the non-deterministic choices are resolved, we cannot get into a state where either the client or the server get blocked and are unable to participate in the method return.

- There is also an element of *livelock-freedom*. It ensures that we cannot get indefinitely delayed by other concurrent activities from completing a method return. Note that our definition in terms of extensions of finite computations in effect incorporates a *strong fairness assumption*; that transitions which are enabled infinitely often must eventually be performed. An alternative treatment could be given in terms of maximal (finite or infinite) computations, subject to explicit fairness constraints.
- Finally, notice that this notion of orthogonality also incorporates a *serialization requirement*: once a given method has been invoked, no further offers of that method will be made until the invocation has been completed by the method return. The obvious way of achieving this is to have each method implementation running as a single sequential thread within the server. This may seem overly restrictive, but we shall see later how with a suitable use of *names* as unique identifiers for multiple instances of generic service methods, this need not compromise expressiveness.

2.1. The Galois Connection

We shall now see how the orthogonality relation allows further structure to be expressed. Let $\mathcal{S}$ be the set of Σ-servers over a fixed ambient set of labels $\mathcal{L}$, and $\mathcal{C}$ the set of Σ-clients over a set of labels $\mathcal{M}$. We have defined a relation $\perp \subseteq \mathcal{S} \times \mathcal{C}$. This relation can be used in a standard way to define a *Galois connection* [13] between $\mathbb{P}(\mathcal{S})$ and $\mathbb{P}(\mathcal{C})$.

Given $S \subseteq \mathcal{S}$, define

$$S^{\perp} = \{c \in \mathcal{C} \mid \forall s \in S.\, s \perp c\}$$

Similarly, given $C \subseteq \mathcal{C}$, define

$$C^{\perp} = \{s \in \mathcal{S} \mid \forall c \in C.\, s \perp c\}$$

The following is standard [13]:

Proposition 2.1 1. *For all $S, T \subseteq \mathcal{S}$: $S \subseteq T \;\Rightarrow\; T^{\perp} \subseteq S^{\perp}$.*
2. *For all $C, D \subseteq \mathcal{C}$: $C \subseteq D \;\Rightarrow\; D^{\perp} \subseteq C^{\perp}$.*
3. *For all $C \subseteq \mathcal{C}$ and $S \subseteq \mathcal{S}$: $S^{\perp} \supseteq C \iff S \subseteq C^{\perp}$.*
4. *For all $C \subseteq \mathcal{C}$ and $S \subseteq \mathcal{S}$: $S^{\perp} = S^{\perp\perp\perp}$ and $C^{\perp} = C^{\perp\perp\perp}$.*

We define $\bar{S} = S^{\perp\perp}$, $\bar{C} = C^{\perp\perp}$. The following is also standard:

Proposition 2.2 *The mappings $S \mapsto \bar{S}$ and $C \mapsto \bar{C}$ are closure operators, on $\mathbb{P}(\mathcal{S})$ and $\mathbb{P}(\mathcal{C})$ respectively. That is:*

$$S \subseteq \bar{S}, \quad \bar{\bar{S}} = \bar{S}, \quad S \subseteq T \;\Rightarrow \bar{S} \subseteq \bar{T}$$

and similarly

$$C \subseteq \bar{C}, \quad \bar{\bar{C}} = \bar{C}, \quad C \subseteq D \;\Rightarrow\; \bar{C} \subseteq \bar{D}\,.$$

We can think of closed sets of servers as *saturated* under "tests" by clients: $\bar{S}$ is the largest set of servers which passes all the tests of orthogonality under interaction with clients that the servers in S do. Similarly for closed sets of clients. We shall eventually view such closed sets as *behavioural types*.

2.2. Examples

We shall give some simple examples to illustrate the definitions. We shall use the notation of CCS [21], but we could just as well have used CSP [16], or other process calculi.

Firstly, we define a signature with two operations:

$$\begin{array}{l}\texttt{inc} : \mathbf{unit} \longrightarrow \mathbf{unit} \\ \texttt{dec} : \mathbf{unit} \longrightarrow \mathbf{unit}\end{array}$$

This specifies an interface for a binary semaphore. We define a server $s \equiv \texttt{binsem}(0)$, where:

$$\begin{array}{l}\texttt{binsem}(0) = \texttt{inc}^{\downarrow}.\texttt{inc}_{\downarrow}.\texttt{binsem}(1) \\ \texttt{binsem}(1) = \texttt{dec}^{\downarrow}.\texttt{dec}_{\downarrow}.\texttt{binsem}(0)\end{array}$$

We define a client c by

$$c \;\equiv\; \texttt{inc}^{\uparrow} \,|\, \texttt{inc}_{\uparrow} \,|\, \texttt{dec}^{\uparrow} \,|\, \texttt{dec}_{\uparrow} \,.$$

Then s is orthogonal to c. However, $\texttt{dec}^{\uparrow}.c$ is not orthogonal to s, since Receptivity fails; while $\texttt{inc}_{\downarrow}.s$ is not orthogonal to c, since Completion fails.

3. The Open Universe

Thus far, we have assumed a simple-minded but convenient dichotomy between clients and servers. There are many reasons to move to a more general setting. The most compelling is that the general condition of systems is to be *open*, interacting with an environment which is not completely specified, and themselves changing over time. Rather than thinking of complete systems, we should think in terms of *system components*.

Relating this to our service-based view of distributed computation, it is natural to take the view that the general case of a system component is a process which:

1. *requires* certain services to be provided to it by its environment
2. *provides* other services to its environment.

Such a component is a *client* with respect to the services in (1), and a *server* with respect to the services in (2). The interface of such a component has the form

$$\Sigma \Longrightarrow \Delta$$

where Σ and Δ are server signatures as already defined; we assume that $\Sigma \cap \Delta = \varnothing$. The idea is that Σ describes the interface of the component to its environment *qua* client, and Δ the interface *qua* server.

A process p has this interface type, written $p :: \Sigma \Longrightarrow \Delta$, if it is described by a labelled transition system on the set of labels

$$\Sigma_c \cup \Delta_s \cup \{\tau\} \,.$$

Here Σ_c is the client-side set of actions of Σ, and Δ_s the server-side set of actions of Δ, as previously defined, while τ is the usual silent action for internal computational steps. Note that this interface type is now regarded as exhaustive of the possible actions of p. Note also that a *pure* Δ*-server* is a process $s :: \Longrightarrow \Delta$ with empty client-side signature, while a *pure* Σ*-client* is a process $c :: \Sigma \Longrightarrow$ with empty server-side signature. Thus servers and clients as discussed previously are special cases of the notion of component.

3.1. Composition

We now look at a fundamental operation for such systems: *composition*, *i.e.* plugging one system into another across its specified interface. It is composition which enables the construction of complex systems from ultimately simple components.

We shall write interface types in a partitioned form:

$$p :: \Sigma_1, \ldots \Sigma_k \Longrightarrow \Delta\,.$$

This is equivalent to $p :: \Sigma \Longrightarrow \Delta$, where $\Sigma = \bigcup_{i=1}^{k} \Sigma_i$. Writing the signature in this form implicitly assumes that the Σ_i are *pairwise disjoint*. More generally, we assume from now on that all signatures named by distinct symbols are pairwise disjoint.

Suppose that we have components

$$p :: \Delta' \Longrightarrow \Sigma, \qquad q :: \Sigma, \Delta'' \Longrightarrow \Theta\,.$$

Then we can form the system $p \lhd_\Sigma q$. Note that our definition of $s \lhd_\Sigma c$ in Section 1 was sufficently general to cover this situation, taking $s = p$ and $c = q$, with $\mathcal{L} = \Delta'_c \cup \Sigma_s \cup \{\tau\}$, and $\mathcal{M} = \Sigma_c \cup \Delta''_c \cup \Theta_s \cup \{\tau\}$.

We now define

$$p \odot_\Sigma q = (p \lhd_\Sigma q)\backslash\Sigma\,.$$

The is the process obtained from $p \lhd_\Sigma q$ by replacing all labels $m^{\downarrow\uparrow}(\vec{x} : \vec{v})$ and $m_{\downarrow\uparrow}(\vec{w} : \vec{y})$, with $m \in \Sigma$, by τ. This internalizes the interaction between p and q, in which p provides the services in Σ required by q.

Proposition 3.1 *The process $p \odot_\Sigma q$ satisfies the following interface specification:*

$$p \odot_\Sigma q :: \Delta', \Delta'' \Longrightarrow \Theta\,.$$

The reader familiar with the sequent calculus formulation of logic will recognize the analogy with the *Cut rule*:

$$\frac{\Gamma \Longrightarrow A \qquad A, \Gamma' \Longrightarrow B}{\Gamma, \Gamma' \Longrightarrow B}$$

This correspondence is no accident, and we will develop a little more of the connection to logic in the next section, although we will not be able to explore it fully here.

3.2. Behavioural Types

We now return to the idea of behavioural types which we discussed briefly in the previous Section. Given a signature Σ, we define

$$\mathcal{C}(\Sigma) = \{c \mid c :: \ \Sigma \Longrightarrow\}, \qquad\qquad \mathcal{S}(\Sigma) = \{s \mid s :: \Longrightarrow \Sigma\}$$

the sets of pure Σ-clients and servers. We define an orthogonality relation

$$\perp_\Sigma \subseteq \mathcal{C}(\Sigma) \times \mathcal{S}(\Sigma)$$

just as in Section 1, and operations $(\cdot)^\perp$ on sets of servers and clients. We use the notation $A[\Sigma]$ for a *behavioural type of servers on* Σ, *i.e.* a set $A \subseteq \mathcal{S}(\Sigma)$ such that $A = A^{\perp\perp}$.

Given a component $p :: \Sigma \Longrightarrow \Delta$ and behavioural types $A[\Sigma]$ and $B[\Delta]$, we define $p :: A[\Sigma] \Longrightarrow B[\Delta]$ if the following condition holds:

$$\forall s \in A.\, s \odot_\Sigma p \in B\,.$$

This is the condition for a component to satisfy a behavioural type specification, guaranteeing important correctness properties across its interface.

Proposition 3.2 *The following are equivalent:*

1. $p :: A[\Sigma] \Longrightarrow B[\Delta]$
2. $\forall s \in A, c \in B^\perp.\, (s \odot_\Sigma p) \perp_\Delta c$

3. $\forall s \in A, c \in B^{\perp}.\ s \perp_{\Sigma} (p \odot_{\Delta} c)$.

These properties suggest how $(\cdot)^{\perp}$ plays the rôle of a logical negation. For example, one can think of property (3) as a form of *contraposition*: it says that if c is in $B^{\perp}$, then $p \odot_{\Delta} c$ is in $A^{\perp}$. Note that $s \odot_{\Sigma} p$ is a Δ-server, while $p \odot_{\Delta} c$ is a Σ-client, so these expressions are well-typed.

3.3. Soundness of Composition

We now come to a key point: *the soundness of composition with respect to behavioural type specifications*. This means that the key correctness properties across interfaces are preserved in a compositional fashion as we build up a complex system by plugging components together.

Firstly, we extend the definition of behavioural type specification for components to cover the case of partitioned inputs. We define $p :: A_1[\Sigma_1], \ldots, A_k[\Sigma_k] \Longrightarrow B[\Delta]$ if:

$$\forall s_1 \in A_1, \ldots, s_k \in A_k.\ (\bigg\|_{i=1}^{k} s_i) \odot_{\Sigma} p \in B\,.$$

Here $\|_{i=1}^{k} s_i$ is the parallel composition of the s_i; note that, since the Σ_i are disjoint, these processes cannot communicate with each other, and simply interleave their actions freely.

Now we can state our key soundness property for composition:

Proposition 3.3 *Suppose we have* $p :: A'[\Delta'] \Longrightarrow B[\Sigma]$ *and* $q :: B[\Sigma], A''[\Delta''] \Longrightarrow C[\Theta]$. *Then*

$$p \odot_{\Sigma} q :: A'[\Delta'], A''[\Delta''] \Longrightarrow C[\Theta]\,.$$

This can be formulated as an inference rule:

$$\frac{p :: A'[\Delta'] \Longrightarrow B[\Sigma] \qquad q :: B[\Sigma], A''[\Delta''] \Longrightarrow C[\Theta]}{p \odot_{\Sigma} q :: A'[\Delta'], A''[\Delta''] \Longrightarrow C[\Theta]}$$

4. Structured Types and Behavioural Logic

We shall now develop the logical structure inherent in this perspective on communicating process architectures a little further.

We shall consider some ways of combining types. The first is very straightforward. Suppose we have behavioural types $A[\Sigma]$ and $B[\Delta]$ (where, as always, we are assuming that Σ and Δ are disjoint). We define the new type

$$A[\Sigma] \otimes B[\Delta] = \{s \parallel t \mid s \in A \text{ and } t \in B\}^{\perp\perp}$$

over the signature $\Sigma \cup \Delta$.

Now we can see the partitioned-input specification

$$p :: A_1[\Sigma_1], \ldots, A_k[\Sigma_k] \Longrightarrow \Delta$$

as equivalent to

$$p :: A_1[\Sigma_1] \otimes \cdots \otimes A_k[\Sigma_k] \Longrightarrow \Delta\,.$$

Moreover, we have the "introduction rule":

$$\frac{p :: A[\Sigma] \Longrightarrow B[\Delta], \qquad q :: A'[\Sigma'] \Longrightarrow B'[\Delta']}{p \parallel q :: A[\Sigma], A'[\Sigma'] \Longrightarrow B[\Delta] \otimes B'[\Delta']}$$

Thus $\otimes$ allows us to combine disjoint collections of services, so as to merge independent components into a single system.

More interestingly, we can form an implication type

$$A[\Sigma] \multimap B[\Delta]\,.$$

This describes the type of components which need a server of type A in order to produce a server of type B:

$$A[\Sigma] \multimap B[\Delta] = \{p :: \Sigma \Longrightarrow \Delta \mid \forall s \in A.\, s \odot_{\Sigma} p \in B\}\,.$$

We can define an introduction rule:

$$\frac{p :: A[\Sigma], B[\Delta] \Longrightarrow C[\Theta]}{p :: A[\Sigma] \Longrightarrow B[\Delta] \multimap C[\Theta]}$$

and an elimination rule

$$\frac{C[\Theta] \Longrightarrow A[\Sigma] \multimap B[\Delta], \qquad q :: C'[\Theta'] \Longrightarrow A[\Sigma]}{q \odot_{\Sigma} p :: C[\Theta], C'[\Theta'] \Longrightarrow B[\Delta]}$$

These correspond to abstraction and application in the (linear) λ-calculus [15].

This gives us a first glimpse of how logical and 'higher-order' structure can be found in a natural way in process architectures built from client-server interfaces. We refer the interested reader to [5,6,3] for further details, in a somewhat different setting.

4.1. Logical Wiring and Copycat Processes

Given a signature $\Sigma = \{m_1 : T_1, \ldots, m_k : T_k\}$, it is useful to create renamed variants of it:

$$\Sigma^{(i)} = \{m_1^{(i)} : T_1, \ldots, m_k^{(i)} : T_k\}\,.$$

We can define an "identity component" which behaves like a copycat process [9,2], relaying information between input and output:

$$\mathtt{id}_{\Sigma} :: \Sigma^{(1)} \Longrightarrow \Sigma^{(2)}$$

$$\mathtt{id}_{\Sigma} = \sum_{m \in \Sigma} m^{(2)\downarrow}(\vec{x}).m^{(1)\uparrow}(\vec{x}).m_{\uparrow}^{(1)}(\vec{y}).m_{\downarrow}^{(2)}(\vec{y}).\mathtt{id}_{\Sigma}$$

In a similar fashion, we can define a component for function application (or Modus Ponens):

$$\mathtt{app}_{\Sigma,\Delta} :: (\Sigma^{(2)} \multimap \Delta^{(1)}) \otimes \Sigma^{(1)} \Longrightarrow \Delta^{(2)}$$

$$\mathtt{app}_{\Sigma,\Delta} = \mathtt{id}_{\Sigma} \parallel \mathtt{id}_{\Delta}\,.$$

To express the key property of this process, we need a suitable notion of equivalence of components. Given $p, q :: \Sigma \Longrightarrow \Delta$, we define $p \approx_{\Sigma,\Delta} q$ iff:

$$\forall s \in \mathcal{S}(\Sigma), c \in \mathcal{C}(\Delta).\ (s \odot_{\Sigma} p) \bot_{\Delta} c \iff (s \odot_{\Sigma} q) \bot_{\Delta} c\,.$$

Now suppose we have processes

$$p :: \Theta_1 \Longrightarrow \Sigma^{(2)} \multimap \Delta^{(1)}, \qquad q :: \Theta_2 \Longrightarrow \Sigma^{(1)}\,.$$

Then, allowing ourselves a little latitude with renaming of signatures, we can express the key equivalence as follows:

$$(p \parallel q) \odot_{(\Sigma^{(2)} \multimap \Delta^{(1)}) \otimes \Sigma^{(1)}} \mathtt{app}_{\Sigma,\Delta} \approx_{\Theta_1 \otimes \Theta_2, \Delta} q \odot_{\Sigma} p\,.$$

5. Genericity and Names

The view of service interfaces presented by signatures as developed thus far has some limitations:

- It is rather rigid, depending on methods having unique globally assigned names.
- It is behaviourally restrictive, because of the serialization requirement on method calls.

Note that allowing concurrent method activations would cause a problem as things stand: there would not be enough information available to associate method returns with the corresponding calls. The natural way to address this is to assign a unique name to each concurrent method invocation; this can then be used to associate a return with the appropriate call. The use of distinct names for the various threads which are offering the same generic service method also allows for more flexible and robust naming.

The notion of names we are using is that of the π-calculus and other ‘nominal calculi’ [22]. Names can be compared for equality and passed around in messages, and new names can be generated in a given scope, and these are the only available operations. We find it notationally clearer to distinguish between name constants α, β, γ, ... and name variables a, b, c, ... but this is not an essential point.

These considerations lead to the following revised view of the actions associated with a method m:

- On the server side, a server offers a *located instance* of a call of m, as an action $\langle\alpha\rangle m^{\downarrow}(\vec{x})$. Several such calls with distinct locations (*i.e.* names α) may be offered concurrently, thus alleviating the serialization requirement. The corresponding return action will be $\langle\alpha\rangle m_{\downarrow}(\vec{w})$. The name can be used to match this action up with the corresponding call.
- On the client side, there are two options for a call: the client may either invoke a located instance it already knows:

  $$\langle\alpha\rangle m^{\uparrow}(\vec{v})$$

  or it may invoke a generic instance with a name variable:

  $$(a) m^{\uparrow}(\vec{v})\,.$$

  This can synchronize with any located instance which is offered, with a being bound to the name of that instance. A return will be on a specific instance:

  $$\langle\alpha\rangle m_{\uparrow}(\vec{y})\,.$$

The definition of the operation $s \lhd_{\Sigma} c$ is correspondingly refined in the rules specifying the client-server interactions, as follows:

$$\frac{s \xrightarrow{\langle\alpha\rangle m^{\downarrow}(\vec{x})} s' \qquad c \xrightarrow{(a) m^{\uparrow}(\vec{v})} c'}{s \lhd_{\Sigma} c \xrightarrow{\langle\alpha:a\rangle m^{\downarrow\uparrow}(\vec{x}:\vec{v})} s'[\vec{v}/\vec{x}] \lhd_{\Sigma} c'[\alpha/a]}$$

$$\frac{s \xrightarrow{\langle\alpha\rangle m^{\downarrow}(\vec{x})} s' \qquad c \xrightarrow{\langle\alpha\rangle m^{\uparrow}(\vec{v})} c'}{s \lhd_{\Sigma} c \xrightarrow{\langle\alpha\rangle m^{\downarrow\uparrow}(\vec{x}:\vec{v})} s'[\vec{v}/\vec{x}] \lhd_{\Sigma} c'} \qquad \frac{s \xrightarrow{\langle\alpha\rangle m_{\downarrow}(\vec{w})} s' \qquad c \xrightarrow{\langle\alpha\rangle m_{\uparrow}(\vec{y})} c'}{s \lhd_{\Sigma} c \xrightarrow{\langle\alpha\rangle m_{\downarrow\uparrow}(\vec{w}:\vec{y})} s' \lhd_{\Sigma} c'[\vec{w}/\vec{y}]}$$

It is easy to encode these operations in the π-calculus [22], or other nominal calculi. They represent a simple design pattern for such calculi, which fits very naturally with the server-client setting we have been considering.

Note that in generic method calls, the client is *receiving* the name while sending the parameters to the call, while conversely the server is sending the name and receiving the parameters. This "exchange of values" [20] can easily be encoded, but is a natural feature of these interactions.

5.1. Orthogonality and Types Revisited

It is straightforward to recast the entire previous development in the refined setting. The main point is that the orthogonality relation is now defined in terms of the symbols $\langle\alpha\rangle m^{\downarrow\uparrow}$ and $\langle\alpha\rangle m_{\downarrow\uparrow}$. Thus serialization is only required on instances.

6. Further Directions

The ideas we have presented can be developed much further, and to some extent have been; see the references in the next section.

One aspect which we would have liked to include in the present paper concerns *data assertions*. The notion of orthogonality we have studied in this paper deals with control flow and synchronization, but ignores the flow of data between clients and servers through the parameters to method calls and returns. In fact, the correctness properties of this data-flow can also be encapsulated elegantly in an extended notion of orthogonality. We plan to describe this in a future publication.

Acknowledgements

This paper builds on much previous work, mainly by the author and his colleagues, although the specific form of the presentation, and the emphasis on procedural interfaces and client-server interactions, has not appeared previously in published form.

We mention some of the main precursors.

- Particularly relevant is the work on *Interaction categories* and *Specification structures* — although no categories appear in the present paper! Much of this was done jointly with the author's former students Simon Gay and Raja Nagarajan [1,9,5,6,7,8]. See also [12] with Dusko Pavlovic.
- Ideas from Game semantics [2,10,9,11], many developed in collaboration with Radha Jagadeesan and Pasquale Malacaria, have also been influential; for example, the four types of actions associated with method calls correspond to the four-fold classification of moves in games as Player/Opponent and Question/Answer.
- Various notions of orthogonality in the same general spirit as that used here, but mainly applied in logical and type-theoretic settings, have appeared over the past 15–20 years. Some examples include [15,19,17]. The application to notions in concurrency originated with the present author [1,5,6,3].
- The closest precursor of the present paper is an unpublished lecture given by the author in 2002 [4].
- General background is of course provided by work in concurrency theory, especially process algebra [16,21,22].

Recent work in a related spirit includes work on session types, see e.g. [23]. Another possibly relevant connection, of which the author has only recently become aware, is with the Handshake algebra developed for asynchronous circuits [18], and a game semantics for this algebra which has appeared recently [14].

References

[1] S. Abramsky. Interaction categories (extended abstract). In G. L. Burn, S. J. Gay, and M. D. Ryan, editors, *Theory and Formal Methods 1993*, pages 57–69. Springer-Verlag, 1993.

[2] S. Abramsky. Semantics of Interaction: an introduction to Game Semantics. In P. Dybjer and A. Pitts, editors, *Proceedings of the 1996 CLiCS Summer School, Isaac Newton Institute*, pages 1–31, Cambridge University Press, 1997.

[3] S. Abramsky. Process realizability. In F. L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation: Proceedings of the 1999 Marktoberdorf Summer School*, pages 167–180. IOS Press, 2000.

[4] S. Abramsky. Reactive refinement. Oxford University Computing Laboratory seminar, 2002.

[5] S. Abramsky, S. Gay, and R. Nagarajan. Interaction categories and the foundations of typed concurrent programming. In M. Broy, editor, *Proceedings of the 1994 Marktoberdorf Summer Sxhool on Deductive Program Design*, pages 35–113. Springer-Verlag, 1996.

[6] S. Abramsky, S. Gay, and R. Nagarajan. Specification structures and propositions-as-types for concurrency. In G. Birtwistle and F. Moller, editors, *Logics for Concurrency: Structure vs. Automata—Proceedings of the VI I Ith Banff Higher Order Workshop*, pages 5–40. Springer-Verlag, 1996.

[7] S. Abramsky, S. Gay, and R. Nagarajan. A type-theoretic approach to deadlock-freedom of asynchronous systems. In M. Abadi and T. Ito, editors, *Theoretical Aspects of Computer Software*, volume 1281 of *Springer Lecture Notes in Computer Science*, pages 295–320. Springer-Verlag, 1997.

[8] S. Abramsky, S. J. Gay, and R. Nagarajan. A specification structure for deadlock-freedom of synchronous processes. In *Theoretical Computer Science*, volume 222, pages 1–53, 1999.

[9] S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. *Journal of Symbolic Logic*, 59(2):543–574, 1994.

[10] S. Abramsky and R. Jagadeesan. New foundations for the geometry of interaction. *Information and Computation, 111(1)*, pages 53–119, 1994.

[11] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. In *Information and Computation*, volume 163, pages 409–470, 2000.

[12] S. Abramsky and D. Pavlovic. Specifying processes. In E. Moggi and G. Rosolini, editors, *Proceedings of the International Symposium on Category Theory In Computer Science*, volume 1290 of *Springer Lecture Notes in Computer Science*, pages 147–158. Springer-Verlag, 1997.

[13] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, second edition, 2002.

[14] L. Fossati. Handshake games. *Electronic Notes in Theoretical Computer Science*, 171(3):21–41, 2007.

[15] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 1987.

[16] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[17] M. Hyland and A. Schalk. Glueing and orthogonality for models of linear logic. *Theoretical Computer Science*, 294:183–231, 2003.

[18] M. B. Josephs, J. T. Udding, and J. T. Yantchev. Handshake algebra. Technical Report SBU-CISM-93-1, South Bank University, 1993.

[19] R. Loader. Linear Logic, Totality and Full Completeness. LICS 1994: 292-298, 2004.

[20] G. Milne and R. Milner. Concurrent processes and their syntax. *Journal of the ACM*, 26(2):302–321, 1979.

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

[22] R. Milner. *Communication and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[23] V. Vasconcelos, S. J. Gay, and A. Ravara. Type checking a multithreaded functional language with session types. *Theoretical Computer Science*, 368((1-2)):64–87, 2006.