Semantics of a component-oriented programming language : COMPO

Julien RIXTE

June 27, 2016

Contents

Chapter 1

Introduction

First, we will give an overview and by the way an informal semantics of COMPO. Then, we will define an operational semantics of the language. In a third chapter, we will discuss the use of primitives. Once the theoretical aspect are dealt with, we will give a glimpse of our implementation of a certified interpreter of COMPO in Coq. To finish, we will focus on the reflexivity of the language.

Chapter 2

Description of Compo

This chapter allows to the reader to become familiar with the language Compo. It will also give us the opportunity to give informal semantics of the language constructions. All the definitions follow closely the Petr Spacek's thesis [2], unless stated otherwise.

2.1 Overview

As in object oriented programming, the main concepts in Compo are **descriptors** and **components**, where a component is an instance of a descriptor. A component can require or provide services threw **ports**. To call a service a component has to **connect** one of its **required ports** to a **provided port** of another component which provides the service needed. A descriptor can specify the **architecture** of composites (ie components containing subcomponents) by defining the connections between its subcomponents.

Component world	**Object world**
Component	Object
Descriptor	Class
Service	Method
Required port	Instance variable
Provided port	None
Connexion	None
Architecture	Constructor

Table 2.1: Correspondence between the component world and the object world

2.2 Descriptors and components

Here is an informal syntax for descriptors.

```
Descriptor descrName extends superName
{
  provides{
    provided port list
  }

  requires{
    externally required port list
  }
  internally requires{
    internally required port list
  }

  architecture{
    connection list
  }

  service service1(arg1,...,arg_n){ body }
  service service2(...){ body }

}
```

A descriptor completely specifies a component behavior. It explicits the different ports, the services and the architecture of the component.

To create a new component from a descriptor, we have to use the keyword **new** :

```
my_descriptor.new()
```

This expression will return the port default (see 2.3.1 below) of a new instance of the descriptor my_descriptor. Notice that the programmer has never access to a whole component. He only has a point of view of this component threw a port (see TODO), here the port default.

In the architecture section, the only ports a component can connect are its internally required ports and the externally required ports of its subcomponents. Subcomponents can be created by adding

```
connect my_internally_required_port to my_new_subcomponent_descriptor.new();
```

in the architecture section.

2.3 Ports

All the ports are compound of an identifier and a signature of service list. Each port has an **owner**, i.e. the component to which the port belongs. However, ports can have different roles (provided/required) or visibilities (external/internal/local).

2.3.1 Special ports

All the components have at list three ports : default, self and super.

Default The port default is a provided port which by default provides all the services implemented within the component. Its default point of view is the set of the external ports. Nevertheless, it can be overridden in the provides section of the descriptor.

Self This port allows a component to call its own services. It is consequently a required port, automatically connected to the port default.

Super This port has the same role as self, but it considers the component as an instance of the super descriptor. (TODO)

2.3.2 Role

A port can be either required or provided. The difference between those two kind of ports is their role in a connection : a required port express that a component needs some service that a provided port can furnish. Then, to call a service on a required port we need to connect this required port to a provided port which provides the services needed. For instance to call a service on the port default on a component viewed by a required port p.

```
connect my_internally_required_port to default@p;
my_internally_required_port.service_name(arg_1, ..., arg_n);
```

In some cases, we could want to express that the services required by a port are the same services as the services requires by another required port. We will call this mechanism **delegation**. A required port can then be **delegated** to another required port. For instance, a muscle need energy to function. If we consider the muscle as a subcomponent of a human body, the human body still needs energy. We can then delegate the need of energy of the muscle to the need of energy of the human body. This need can then be satisfied by a provided service in a component food.

In a similar manner, we can define delegation between to provided ports. For instance, if an enterprise has two branches, one producing pens and the other pencils, we can say that this enterprise produces both pens and pencil : the component enterprise has to provided ports pen and pencil which are delegated to the corresponding provided ports of the branches.

The table 2.2 summaries these different ways to connect components.

Connect \ To	Provided	Required
Provided	Delegation	$\times$
Required	Regular connection	Delegation

Table 2.2: Connection types

Connection chains The fact that provided ports can be delegated leads us to ask which port is actually providing the service. Let's consider a **connection chain**

$$r_1, ..., r_m, p_1, ..., p_n$$

where $r_1, ..., r_m$ are required port, $p_1, ..., p_n$ are provided port , r_1 is delegated to r_2, r_2 to r_3 ... r_{m-1} to r_m, r_m connected to p_1 and p_1 is delegated to p_2, ... ,p_{n-1} to p_n. p_1

is called the **providing port** of the chain and p_n **final port**. Even if the port which actually executes the service is the final port, we will consider that the port providing the service is the providing port. As an analogy, if we see required ports as pointers, they point on the providing port and not on the final port.

This is not only a matter of vocabulary. Indeed, the instruction $!r$ will return the providing port of the connection chain starting from r and not the final port.

2.3.3 Visibility

External

Internal

Local A local port is a port which exists within a service. At the end of the service, it is deleted. If an internal port is connected (in the broad sense) to the local port, the connection will be broken. It is repaired at the end of the service (see TODO).

2.4 Services

2.4.1 Argument passing

2.4.2 Return

A service can either return a provided port or a required port. However, if it returns a provided port, it is not possible to chain calls :

```
(port.get_provided_port()).a_service(); "here get_provided_port returns a
    provided port"
```

will crash. In fact, this is not really a problem because the programmer could always write

```
require tmp_port;
connect tmp_port to port.get_provided_port();
tmp_port.a_service;
```

We believe that this is to heavy (TODO voc). Then, we will introduce transparent ports, which will allow to chain the service calls.

Definition 1. *A port p is **transparent** iff it is impossible to connect a port p' to p. If the programmer tries to connect a port p' to p, p' will be connected to the first non transparent port of the connection chain starting from p.*

If a provided port is returned we will create an temporary transparent required port connected to the provided port. It is then possible to call a service on this port. Moreover, if we want to connect a port to the return, it will be connected to the provided port and not to the transparent port. A transparent port can not be manipulated by the programmer.

2.4.3 Points of view

```
Descriptor PrivacyExample extends Component
{
   provides{
      default:{public_serv} - {private_prov}
      private_prov:{private_serv}
   internally requires{
      private:{private_serv()}
   }

   service public_serv{...}
   service private_serv{...}

}
```

(TODO : templates)

2.5 Our contributions

Visibility Both internal and external ports were already implemented by Petr Spacek. However internal ports were accessible via the operator @. Moreover, a component was able to build a connection with any port. This problem had already been pointed out by our predecessors who had suggested that it should not have the possibility to connect its external ports. We went even further : a component can only connect its own internally required ports and the required ports of its direct subcomponents.

Providing/final port Our predecessors identified two kinds of argument passing :

Call-by-require (TODO add the schemes of Jimmy's report)

Call-by-provide

They introduced the notation `!p` **only in the argument passing** to specify that the argument was passed by require. A first critic is that ! is the dereferencing operator in OCaml dereferencing and it would be more intuitive to use ! for the call-by-provide passing mode.

Moreover it seems reasonable to have also the possible to do a provided connection, i.e. to have the ability to connect to a provided port instead of the required port. We then decided that the operator ! will be an expression instead of an annotation. We also introduced the definitions of providing port and final port.

2.5.1 Points of view

The status of the variables and of the ports were not clearly define. Indeed, the initial vision of the expression

```
name@expr
```

implicitly considered that it was returning the port `name` of the owner of `expr`, while expr is in fact a port. We believe that it was not relevant because it would imply all the components *were* their ports. Instead, we proposed that each port is in fact a point of view of the component, which can occult some other ports.

2.6 Abstract syntax

Program (in a service) :

$$\begin{array}{lll} Name & ::= & camelCase \\ Expr & ::= & Name \mid Expr.Name(Expr, ..., Expr) \mid Name@Expr \mid !Expr \\ Prog & ::= & \perp \mid Prog; Prog \mid \textbf{require}\ Name \mid \\ & & \quad \textbf{connect}\ Name\ \textbf{to}\ Expr \mid \textbf{return}\ Prog \end{array}$$

Port :

$$\begin{array}{lll} Sig & ::= & Name(Name, ..., Name) \\ List[V] & ::= & \epsilon \mid V; List[V] \qquad \text{where } V \text{ is a variable of the grammar} \\ Port & ::= & Name : \{List[Sig]\} \end{array}$$

Descriptor :

$$\begin{array}{lll} Serv & ::= & \textbf{service}\ Sig\ \{Prog\} \\ Descr & ::= & \textbf{descriptor}\ Name\ \textbf{extends}\ Name\{ \\ & & \quad \textbf{provides}\ \{List[Port]\} \\ & & \quad \textbf{requires}\ \{List[Port]\} \\ & & \quad \textbf{internally requires}\ \{List[Port]\} \\ & & \quad \textbf{architecture}\ \{List[Connect]\} \\ & & \quad List[Service]\ (\textit{without}\ ;) \\ & & \} \end{array}$$

Figure 2.1: COMPO's abstract syntax

To add : ofKind (spa13 p.165) et * (universal interface) and

2.7 Repairing connections at the end of a service

Keep a list of port connected to a local port. Then two possibilities : *make all the local ports transparent and call the connect instructions on all the port of the list + pass the list to the calling service *optimisation : connected components of the graph of the local ports.

Chapter 3

Type system

The COMPO language is typed. Question

3.1 Environments

3.1.1 The local environment

The local environment contains the type of all the local ports of a service (i.e. the arguments and the temporary variables). We need to store the connections of the ports and the role (required /provided) because otherwise, we will not be able to type the instructions ! and @ (to type them, we need to follow the connexion chain). Consequently it can be represented as a partial function $\Theta : Name \rightarrow Interface \times \{R, P\} \times Name$

3.1.2 The descriptor environment

To type all the program, we will need to know the type of all the ports belonging to components. However, the type of a port can not change during the execution. Therefore we can just store the types of the ports in the descriptors : $\Delta : Name \rightarrow Name \rightarrow Interface \times \{R, P\} \times Name$

3.1.3 The component environment

3.1.4 The current component

When we are typing the body of a service, we need to know to which descriptor this service belongs.We will call this desciptor δ.

3.2 What is typed?

3.2.1 Port

A port type consists mainly in the signature list of its services. Though, there are other candidates to be added to the port type :

Interface The interface is the main element of the type of a port. It is compounded of a partial function which associates a name to a service signature. A service signature consists of a list of interfaces for the arguments and an interface for the return. However, this definition is not well founded. Consequently, we need to name interfaces : the only way to break the induction loop is to use named interface instead of interface. We have to use a co-inductive definition :

$$\frac{}{\bigcup_{\delta \in Im(\Delta)} \delta\#name.Dom(\delta\#\pi) \subset \mathscr{P}_T}$$

$$\frac{\begin{array}{l} a_1 \in \mathscr{P}_T \\ \quad\vdots \\ a_n \in \mathscr{P}_T \end{array} \qquad ret \in \mathscr{P}_T}{< (a_1, ..., a_n), ret > \in \mathscr{I}}$$

$$\frac{}{\mathfrak{P}_F(\mathscr{I}) \times \{P, R\} \subset \mathscr{P}_T}$$

Figure 3.1: Definition of $\mathscr{I}$ and $\mathscr{P}_T$

Once we have defined $\mathscr{I}$, we need to define a subtype relation on it, coinductively again. Th sub typing relation does not take into account the role of the port. That is why we use the wild-card character _.

Provided/required The role of a port will be checked by the type checker. Even if in the semantics the role of ports will be stored in the component, it seems more consistent that the role of port is included in its type.

Connexion If we want to type the instruction !*port*, we could need to include the connection of a port in its type. Indeed, the signature list of *port* can be different from that of !*port*. We would then need to follow a connection chain to be able to give the exact type of !*port*. However, to make a parallel with the object world, we do not specialize the type of an object when we apply a dereferencing operator. We could then just consider that the type of !*port* is the same as the type of *port*.

Here we will choose not to store the connexion of the port.

The type of a port is consequently an element of the set

$$\mathscr{P}_T = \mathscr{I} \times \{P, R\}$$

3.2.2 Component

The type of a component is a list of port types list of ports + descriptor

$$\frac{\begin{array}{c} a_1 \succeq a'_1 \\ \vdots \\ a_n \succeq a'_n \end{array} \qquad ret \preceq ret'}{< (a_1, ..., a_n), ret > \preceq < (a'_1, ..., a'_n), ret' >}$$

$$\frac{getType(descr.port) \preceq b}{descr.port \preceq b}$$

$$\frac{a \preceq getType(descr.port)}{a \preceq descr.port}$$

$$\frac{\begin{array}{c} \tau_1 \preceq \upsilon_1 \\ \vdots \\ \tau_n \preceq \upsilon_n \end{array}}{< \{serv_1 : \tau_1; ...; serv_n : \tau_n\}_{,>} \preceq < \{serv_1 : \upsilon_1; ...; serv_n : \upsilon_n; ...; serv_p :: \upsilon_p\}_{,>}}$$

Figure 3.2: Subtyping relation in Compo

3.2.3 Descriptors

list of signature + architecture + port declarations

3.3 Type rules

$$Var_{loc}\ \frac{}{\Gamma, \Theta, \sigma \vdash v : \Theta(v); \Gamma}$$

$$Var_{descr}\ \frac{}{\Gamma, \Theta, \sigma \vdash v : \Gamma(\sigma)(v); \Gamma}\ v \notin Dom(\Theta)$$

$$Call\ \frac{\begin{array}{c}\Gamma, \Theta, \sigma \vdash expr : T; \Gamma' \\ \Gamma', \Theta', \sigma \vdash arg_1 : T_1; \Gamma_1 \\ \vdots \\ \Gamma_{n-1}, \Theta_{n-1}, \sigma \vdash arg_n : T_n; \Gamma_n\end{array} \qquad T\#type(serv) =< (T_1, ...T_n), T_{ret} >}{\Gamma, \Theta, \sigma \vdash expr.serv(arg_1, ...arg_n) : T_{ret}; \Gamma_n}$$

$$At\ \frac{\Gamma, \Theta, \sigma \vdash expr : T; \Gamma'}{\Gamma, \Theta, \sigma \vdash port@expr : T'; \Gamma'}$$

where $T' = \Gamma(findProv(T)\#self_c)(port)$

$$Path\ \frac{\Gamma, \Theta, \sigma \vdash expr : T; \Gamma'}{\Gamma, \Theta, \sigma \vdash_e !expr : findProv(T); \Gamma'}$$

$$New\ \frac{\Gamma', \Theta, max \vdash arch \implies (); \Gamma''}{\Gamma, \Theta, \sigma \vdash descr.new() \implies val; \Gamma''}$$

with :

- $newCompo =< [\![descr\#\pi_P]\!]_{pp}, [\![descr\#\pi_R]\!]_{rp}, >$
- $max = max(Dom(\Gamma)) + 1$
- $\Gamma' = \Gamma + [max \to newCompo]$
- $val = \Gamma''(max)\#\pi_P(default)$

Figure 3.3: Type system of Compo expressions

$$\perp \; \frac{}{\Gamma, \Pi, \Omega, \sigma \vdash_p \perp \Longrightarrow \perp; \Gamma, \Pi, \Omega}$$

$$\mathit{Seq} \; \frac{\Gamma, \Theta, \sigma \vdash_p \mathit{prog1} : T; \Gamma', \Theta' \qquad \Gamma', \Theta', \sigma \vdash_p \mathit{prog2} : U; \Gamma'', \Theta''}{\Gamma, \Theta, \sigma \vdash_p \mathit{prog1}; \mathit{prog2} : U; \Gamma'', \Theta''}$$

$$\mathit{Ret} \; \frac{\Gamma, \Theta, \sigma \vdash \mathit{expr} : T; \Gamma'}{\Gamma, \Theta, \sigma \vdash \mathbf{return}\ \mathit{expr} : T; \Gamma', \Theta'}$$

+connect Ω

$$\mathit{Require} \; \frac{}{\Gamma, \Theta, \sigma \vdash \mathbf{require}\ \mathit{name} : T : (); \Gamma, \Theta + [\mathit{name} \rightarrow < \mathit{type}, \sigma, \mathit{empty}, 0 >]}$$

$$\mathit{Connect} \; \frac{\Gamma, \Theta, \sigma \vdash_e \mathit{expr}_1 : T_1; \Gamma' \qquad \Gamma', \Theta', \sigma \vdash_e \mathit{expr}_2 : T_2; \Gamma''}{\Gamma, \Theta, \sigma \vdash_p \mathbf{connect}\ \mathit{expr}_1\ \mathbf{to}\ \mathit{expr}_2 : (); \Gamma_{res}, \Theta_{res}}$$

with :

- $T_1\#\mathit{interface} \subset T_2\#\mathit{interface}$
- $(\Gamma_{res}, \Theta_{res}) = \begin{cases} (\Gamma'', \Pi + [v_1\#\mathit{name} \rightarrow \mathit{newPort}]) & \text{if } v1\#\mathit{name} \in \mathit{Dom}(\Pi) \\ (\Gamma'' + [v_1\#\mathit{self}_c \rightarrow \mathit{newCompo}], \Pi) & \text{else} \end{cases}$
- $\Omega_{res} = \begin{cases} \Omega'' \cup \{v_1\} & \text{if } v_1 \notin \mathit{Dom}(\Pi) \text{ and } v_2 \in \mathit{Dom}(\Pi) \\ \Omega'' & \text{else} \end{cases}$
- $\mathit{newPort} = \begin{cases} < v_1\#\mathit{name}, v_1\#\mathit{self}_c, v_2\#\mathit{sigs}, \\ \qquad v_2\#\mathit{cp}_{\mathit{name}}, v_2\#\mathit{cp}_c, v_2\#\mathit{ap} > & \text{if } v_1\#\mathit{name} \in \mathit{Dom}(\Pi) \\ < v_1\#\mathit{name}, v_1\#\mathit{self}_c, v_1\#\mathit{sigs}, \\ \qquad v_2\#\mathit{self}_{\mathit{name}}, v_2\#\mathit{self}_c, v_1\#\mathit{ap} > & \text{else} \end{cases}$
- $\mathit{newCompo} =< \Gamma''(v_1\#\mathit{self}_c)\#\mathit{servs}, \pi_P, \pi_{ER}, \pi_{IR} >$ where

$$\pi_P\ (\mathit{resp}\ \pi_{ER}, \pi_{IR}) = \begin{cases} \Gamma''(v_1\#\mathit{self}_c)\#\pi_P\ (\mathit{resp}\ \pi_{ER}, \pi_{IR}) + [\mathit{name} \rightarrow \mathit{newPort}] \\ \qquad \text{if } \mathit{name} \in \mathit{Dom}(\Gamma'(v_1\#\mathit{self}_c)\#\pi_P)\ (\mathit{resp}\ \pi_{ER}, \pi_{IR}) \\ \Gamma''(v_1\#\mathit{self}_c))\#\pi_P\ (\mathit{resp}\ \pi_{ER}, \pi_{IR}) \\ \qquad \text{else} \end{cases}$$

Figure 3.4: Type system of Compo programs

Chapter 4

COMPO's semantics

In this section, we are going to describe a big-step semantics of COMPO. We choose the big-step style because the aim of this internship is more to describe the language with precision than to prove theorems with the semantics. The, the big-step style is more relevant than a denotational or a small-step style [1]. Indeed, fix points make denotational more difficult to understand and small-step semantics do not describe what an instruction do as directly as big-steps semantics.

"We speculate that it would be easier to convince the standards committee in charge of a given programming language of the adequacy of a big-step formalization than to convince them of the adequacy of a small-step formalization."[1]

However, even if we are going to use a big-step style to describe programs, we will use a denotational style to describe the declaration of the language's structures e.g. signatures, services, ports, descriptors. We made this choice because the semantics of the declaration of a descriptor, for instance, only consists in parsing the different sections (e.g. requires, architecture) and make them available in a mathematical structure. Thus, it can be seen as a transformation of the text into a higher-level structure. Moreover, no fix-points will appear in the semantics.

4.1 Environments

In order to describe the program's state at each moment of the execution, we need several environments. Almost all the environments that we are going to introduce are partial functions from a set of identifiers to a value domain.

4.1.1 The descriptor environment

At each moment of the execution, we can invoke $descr.new()$. Consequently, we need an environment $\Delta : Name \to \mathscr{D}$, where $\mathscr{D}$ is the value domain of descriptors. Once all the descriptors have been evaluated, this environment will not change during the execution. (TODO à moins qu'on puisse déclarer de nouveaux descripteurs à partir de la couche réflexive).

4.1.2 The required port environment

This environment is similar to the standard environment of variables : its scope is local and it evolves each time a required port is declared or connected. Consequently, we need a partial function $\Pi : Name \rightarrow \mathscr{P}$, where $\mathscr{P}$ is the value domain of ports (as described in 4.2.6, page 17). This environment will only contain the required port declared in the body of a service. Consequently, the required ports of the component which owns the service are not in Π.

4.1.3 The component environment

This environment gives us references to components. As we will store connections in ports, ports will need to store components. However, if a port p owns the whole component to which it is connected, if this component is changed by another port, it will not be changed in p. Thus, we will need a partial function $\Gamma : \mathbb{N} \rightarrow \mathscr{C}$, where $\mathscr{C}$ is the value domain of components as defined in 4.2.7, page 18. Each component has then a unique identifier. To refer to $\mathbb{N}$ as the set of references to components, we define $\mathscr{C}_{ref} = \mathbb{N}$.

4.1.4 The current component

In the execution of a service, we need to be aware of the component to which this service belongs. Indeed, as the require ports of the current component are not in Δ, we have to add it in the state of a program, otherwise we could not access those ports. As all components are contained in Γ, we only need $\sigma \in \mathbb{N}$ the reference to the current component

4.2 Value domains

In the semantics, we will need as many domain values as structures in the language. We give a summary of the value domain in the table 4.2.8 and notations are specified in the annex.

4.2.1 Error

We already have the error program $\perp$. This program, when it is evaluated, have to return a value that we will call $\perp$ too. For each incorrect program, the semantics will give this value.

4.2.2 Side effect

Some programs as **connect** do not return any value. Then we need a value for side effects denoted ().

4.2.3 Signatures

The signature of a service consists in its name, the type of its arguments and the return type. Here we will consider that arguments of signatures are not typed. Consequently, two signatures are equal if and only if they have the same name and the same number of arguments.

$$\mathscr{S}_{ig} = Name \times \mathbb{N}$$

4.2.4 Services

A service is the implementation of a signature. To characterize a service, we need three informations :

- its name, which is actually contained by the signature of the service
- the ordered list of the arguments name of the service. We define $\mathscr{L}(E)$ the set of the elements belonging to a set E by induction :
  - $\mathbf{nil} \in \mathscr{L}(E)$
  - $x \in E \implies l \in \mathscr{L}(E) \implies x :: l \in \mathscr{L}(E)$

  The argument list domain is then $\mathscr{L}(Name)$.
- the implementation of the service i.e. an element of $Prog$.

To conclude,

$$\mathscr{S}_{erv} = \mathscr{S}_{ig} \times \mathscr{L}(Name) \times Prog$$

Let's note that we could have chosen to only store the name of the service instead of its signature because the number of arguments can be deduced of the list. However, it will be simpler to store the signature because it will allow us, at the call of a service, to directly test the equality of signatures.

4.2.5 Connexions

There are to possibilities to represent connexions :

- add a new value domain connexion which would be a set of port pairs.
- store directly in the ports the ports to which they are connected.

We will adopt the second option. This choice is justified by two facts. First, in his thesis, Petr Spacek decided to not reify the connections (TODO : citation + explication) and considering a connexion as a value would be similar to reifiy a connexion. (TODO justifier parce que c'est quand même un peu gratuit). Second, it will simplify the semantics rules because we will not have an intermediate object to get the component connected to the port on which we call a service.

4.2.6 Ports

First of all, let's explain why we can consider that required ports and provided ports can share the same domain value. The main difference between a provided port and a required port is that the signature of the provided port (if it is not delegated) demands an implementation of the signatures it contains. However, as two provided ports of a same component can provide a same service, it's seems that the sles ports accessibles depuis notre port ervices should be owned by components and not by ports to avoid redundancy. That is why the provided ports will not contain the implementation of the services they provide.

Consequently, both required ports and provided ports only contain signatures and as a provided port can be connected to another provided port, both need to store the port to which they are connected. Thus we will use a unique value domain $\mathscr{P}$ for provided ports and required ports.

Here we list the necessary informations to characterize a port :

Name The name of a port allows us to identify it. Two ports, even if one is provided and the other required can not have the same name.

Signatures We need to know the signature set of a port to verify that the call of a service is authorized. As the order in which signatures appear does not matter, we will use signature sets rather than signature lists.

Connexion We chose to store connexions directly in the ports. To do this, we need to store the port and the component to which our port is connected[1]. In fact, as a component must store all its ports, knowing the name of the port and the reference of the component is enough to be able to identify it.

Owner component When the instruction *name@p* is evaluated, we need to access to the port called *name* in the component to which p belongs. Then each port must store a reference to the component to which it belongs.

Accessible ports Access to an internal port is forbidden from a service which does not belong to the owner of this internal port.(TODO)

To conclude, we define

$$\mathscr{P} = Name \times \mathscr{C}_{ref} \times \mathfrak{P}(\mathscr{S}_{ig}) \times Name \times \mathscr{C}_{ref} \times \mathfrak{P}(Name)$$

We then define those field names : $< self_{name}, self_c, sigs, cp_{name}, cp_c, ap >$ with

$self_{name}$	the port's name
$self_c$	the identifier of the component to which the port belongs
$sigs$	the signature set
cp_{name}	the name of the port to which our port is connected
cp_c	the identifier of the component to which our port is connected
ap	the ports accessible from our port via the operator @

[1]here we use the word connected in the broad sense : it can be either connected or delegated

4.2.7 Components

A component has to store :

- its services
- its provided ports
- its external required ports
- its internal required ports

In fact, there is no need to store the services in each component. Indeed, it would be a loss of memory as all the components of a same descriptor will have the same services. Consequently, we will instead store the name of the descriptor of the component and access the services on the descriptor.

Each of those structures is identified by a name and two ports, even if they don't have the same visibility, can not have the same name. A component is then characterized by three partial functions which domain is included in *Name*. Names are consequently a redundant information but using partial functions rather than sets highly simplifies the notations.

$$\mathscr{C} = Name \times (Name \rightarrow \mathscr{P})^3$$

4.2.8 Descriptors

The descriptor value is not really different from the syntactic declaration of a descriptor. As a descriptor is not instantiated, we can not declare ports nor execute the architecture. We will only separate the different sections of the descriptor and store them in different fields :

$$\mathscr{D} = Name^2 \times (Name \rightarrow \mathscr{P})^3 \times Prog \times Name \rightarrow \mathscr{S}_{erv}$$

Notation	Value type	Domain
$\mathscr{S}_{ig}$	Signature	$Name \times \mathbb{N}$
$\mathscr{S}_{erv}$	Service	$\mathscr{S}_{ig} \times \mathscr{L}(Name) \times Prog$
$\mathscr{P}$	Port	$Name \times \mathscr{C}_{ref} \times \mathfrak{P}(\mathscr{S}_{ig}) \times Name \times \mathscr{C}_{ref} \times \mathfrak{P}(Name)$
$\mathscr{C}$	Component	$Name \times (Name \rightarrow \mathscr{P})^3$
$\mathscr{D}$	Descriptor	$Name^2 \times List[Port]^3 \times Prog \times Name \rightarrow \mathscr{S}_{erv}$

Table 4.1: Value domains

4.3 Semantics of the language structures

4.3.1 Miscellaneous functions

Semantics of lists Lists appear frequently in Compo: there are lists of arguments, of ports, of services and of signature. We then need a function which takes a semantic function on a domain value D and lift it to a semantic function on $\mathscr{L}(D)$ or $\mathfrak{P}(D)$. More formally, we want to lift $[\![]\!] : E \to F$ (where E is a subset of elements of the grammar of Compoand F a value domain) to a function $lift : List[E] \to \mathfrak{P}(E)$ or $\mathscr{L}(E)$.

Such a function is naturally defined by

$$\begin{array}{lll} lift_{\mathfrak{P}}: & (E \to F) & \to List[E] \to \mathfrak{P}(E)) \\ & [\![]\!] & \to \begin{cases} \epsilon & \to \emptyset \\ x_1; ...; x_n & \to \{[\![x_1]\!]\} \cup lift_{\mathfrak{P}}([\![]\!])(x_2; ...; x_n) \end{cases} \end{array}$$

In the same way, we define

$$\begin{array}{lll} lift_{\mathscr{L}}: & (E \to F) & \to List[E] \to \mathscr{L}(E)) \\ & [\![]\!] & \to \begin{cases} \epsilon & \to \mathbf{nil} \\ x_1; ...; x_n & \to [\![x_1]\!] :: lift_{\mathscr{L}}([\![]\!])(x_2; ...; x_n) \end{cases} \end{array}$$

From now on, we would take the liberty to write $[\![]\!](x_1; ...; x_n)$ instead of $lift_{\mathfrak{P}/\mathscr{L}}([\![]\!])(x_1; ...; x_n)$ when there is no ambiguity.

4.3.2 Semantics of signatures

Let us define the function $[\![]\!]_{sig} : Sig \to \mathscr{S}_{ig}$. It just consists in extracting the name of the signature and counting the number of arguments in a signature . Then we will need $[\![]\!]_{args} : (Name, ..., Name) \to \mathbb{N}$ which counts the number of elements of a list :

$$\begin{array}{lcl} [\![()]\!]_{args} & = & 0 \\ [\![(arg_1, ..., arg_n)]\!]_{args} & = & 1 + [\![(arg_1, ..., arg_{n-1})]\!]_{args} \end{array}$$

We can directly deduce

$$[\![name(arg_1, ..., arg_n)]\!]_{sig} =< name, [\![(arg_1, ..., arg_n)]\!]_{args} >$$

4.3.3 Semantics of services

The semantics of a service can be directly deduced of the domain value of services : we only need to extract a signature, an argument list and a body.

$$\begin{array}{l} [\![\mathbf{service}\ name(arg_1, ..., arg_n)\{body\}]\!]_{serv} = \\ \qquad < [\![name(arg_1, ..., arg_n)]\!]_{sig}, lift_{\mathscr{L}}(Id_{Name})(arg_1, ..., arg_n), body > \end{array}$$

where Id_{Name} is the identity function on $Name$.

4.3.4 Semantics of ports

Here, we will describe the semantics of a port declaration. A port needs to know to which component it belongs. Consequently, the semantics of the declaration of a port depends on the component where this declaration is located. We then search a function with the form

$$[\![]\!]_p : Port \rightarrow \mathscr{C}_{ref} \rightarrow \mathscr{P}$$

such that $[\![name : \{sig_1; ...; sig_n\}]\!]_p(\sigma)$ returns a value of the port domain owned by the component σ.

As we can see in the definition of the value domain $\mathscr{P}$, a port is always connected to another port. We then need to find a solution to connect ports at initialization. To do that, we have to distinguish provided and required ports : at initialization , a provided port will be connected to itself (as it provides its own services) and a required port to a void provided port. Moreover, the point of views of internal port and an external port are different. We then need three functions for ports even if we have only one domain.

Provided ports To specify that a provided port is not delegated, we are going to connect it to itself. It will then be easy to know whether a port is final : it just consists in testing the equality between $(self_{name}, self_c)$ and (cp_{name}, cp_c).

Moreover, a provided port is an external port. Consequently, we cannot access internal ports via a provided port.

$$[\![name : \{sig_1; ...; sig_n\}]\!]_{pp}(\sigma) = \\ < name, \sigma, [\![]\!]_{sig}(sig_1; ...; sig_n), name, \sigma, Dom(\sigma\#\pi_X) >$$

Required port A required port which is not connected will behave as follow : when a service is called on this port or if it is used in the right part of an @, the program will crash. This behavior can be simulated by connecting the required port to a void provided port with a void point of view on the component to which it belongs.

Let us define $\otimes =< void, 0, \emptyset, void, 0, \emptyset >$ the void provided port and $C_\otimes =< \otimes, \emptyset, \emptyset, [] >$ the component with only one port : $\otimes$. We will need to access this component. We can then assume that it will always be referenced by 0 in Γ at any moment of the execution.

Invariant 1. *At any time of the execution,* 0 *refers to the component* $C_\otimes$

The point of view of external required ports is the same as that of provided ports. However, internal ports can see all the ports of the component to which they belong. Finally, we can define

$$[\![name : \{sig_1; ...; sig_n\}]\!]_{erp}(\sigma) = \\ < name, \sigma, [\![sig_1; ...; sig_n]\!]_{sig}, empty, 0, Dom(\sigma\#\pi_X) >$$

and

$$[\![name : \{sig_1; ...; sig_n\}]\!]_{irp}(\sigma) =< name, \sigma, [\![sig_1; ...; sig_n]\!]_{sig}, empty, 0, Dom(\sigma\#\pi)) >$$

It may be worth to remind that we took the freedom to write $[\![sig_1; ...; sig_n]\!]_{sig}$ instead of $lift_{\mathfrak{P}}([\![]\!]_{sig})(sig_1; ...; sig_n)$.

4.3.5 Semantics of descriptors

The semantics of a descriptor only consists in extracting informations. However, we do not know the reference of the owner to initialize the provided ports. The best would have been to evaluate it later. Though, we will need to verify the specialization of ports in the inheritance mechanism. We will then initialize the connexion of the provided ports and we will correct it when `new` is called.

$$\begin{array}{l}
[\![\textbf{descriptor}\ descr\_name\ \textbf{extends}\ super\_name\{ \\
\quad \textbf{provides}\ \{pp\} \\
\quad \textbf{requires}\ \{erp\} \\
\quad \textbf{internally requires}\ \{irp\} \\
\quad \textbf{architecture}\ \{arch\} \\
\quad servs \\
\}]\!]_{descr}(\Delta) = \\
\begin{cases} d & \text{if } inheritanceCorrectness(\Delta, d) \\ \bot & \text{else} \end{cases}
\end{array}$$

where

$$\begin{array}{l}
d =< descr\_name, super\_name, [default \to default_{port}] + [\![pp]\!]_{pp}(0), \\
[self \to self_{port}; super \to super_{port}] + [\![erp]\!]_{erp}(0), [\![irp]\!]_{irp}(0), arch, [\![servs]\!]_{serv} >
\end{array}$$

.

Note that all the fields except the name are optional in descriptor. Let us precise the value of $[\![]\!]_{descr}$ when a field has not been given.

super_name if the descriptor do not inherit from any other descriptor, $super\_name = descr\_name$

pp/erp/irp void function

service void list

architecture ϵ (the void word?? TODO)

Let us define *inheritanceCorrectness* by closely follow the choices made by Petr Spacek [2].

Inheritance verification

4.4 Big-step semantics of expressions

The rules of the semantics will follow the scheme :

$$\textit{environments} \vdash \textit{expression} \implies \textit{value}\ ;\ \textit{new environments}$$

As the descriptor environment Δ does not vary during the execution, it will not appear in the rules. (TODO reflexivity?)

4.4.1 Some simplifications

Let us give some definitions to facilitate the reading of the rules

Initialization of an environment At the beginning of a service execution, we have to initialize the environments of require ports. In the service we will have to access to

- the required ports of the component owning the service
- the parameter ports

A first approach would be to not distinguish the instance ports and the local ports (i.e. the parameter ports and the temporary ports). Though, a local port and an instance port do not behave exactly the same way : an instance port appears in a connection chain while a local port is transparent (see 2.3.3 page 6).

That is why we will not include the required port environment of the component in the environment of the service. The initialization then consists in create an environment for the parameter ports.

Use of the required port environment However, this decision add a little difficulty : if a local port has the same name as an instance port, we have to specify that we are referring to the local port. We will then use an augmented environment Π^+_σ, depending on a component σ :

$$\Pi^+_\sigma(x) = \begin{cases} \Pi(x) & \textit{if } x \in \textit{Dom}(\Pi) \\ \sigma\#\pi_R(x) & \textit{else if } x \in \textit{Dom}(\sigma\#\pi_R) \\ \bot & \textit{else} \end{cases}$$

Creation of a transparent port After a service call, we will need to create a transparent port (cf 2.4.2) if the service has returned a provided port. However, if the service had returned a required port, we do not need to change anything. Here we define how this transparent port is declared when a service called by a component σ returned a port p :

$$\textit{transp}_\sigma(p) =$$
$$\begin{cases} < \epsilon, \sigma, p\#\textit{sigs}, p\#\textit{name}, p\#\textit{self}_c, p\#\textit{ap}, \textit{true} > & \text{if } p\#\textit{name} \in \textit{Dom}(p\#\pi_p) \\ p & \text{else} \end{cases}$$

4.4.2 Localization of ports

As seen in 2.3.2 page 5 we will need to locate the providing port and the final port. To do so, we just have to follow the path and to verify that each step of the path is correct. We first define a function to find the providing port.

$$findProv_{\Gamma,\Pi}(p) =$$
$$\begin{cases} p & \text{if } p\#self_{name} \in Dom(\Gamma(p\#self_c)\#\pi_P) \\ findProv_{\Gamma,\Pi}(p\#cp) & \text{else if } p\#cp\#name \in Dom(p\#cc\#\pi) \\ \otimes & \text{else} \end{cases}$$

Then we can define a miscellaneous function which takes in argument provided port and returns the final port.

$$findFinalFromProv_{\Gamma,\Pi}(p) =$$
$$\begin{cases} p & \text{if } p\#self_{name} = p\#cp_{name},\ p\#self_c = p\#cp_c \\ & \text{and} \quad p\#self_{name} \in Dom(\Gamma(p\#self_c)\#\pi_F) \\ findFinalFromProv_{\Gamma,\Pi}(p\#cp) & \text{else if } p\#cp\#name \in Dom(p\#cc\#\pi_P) \\ \otimes & \text{else} \end{cases}$$

$$findFinal = findFinalFromProv \circ findProv$$

(TODO : in fact we don't have to do any restriction, all the verifications are already done. to prove)

Invariant 2. *All the ports is connected by a connection chain to a final provided port. (TODO preuve de la bonne fondaison de la relation connect.)*

Invariant 3.

$$\forall c \in Dom(\Gamma), Dom(c\#\pi_P) \cap Dom(c\#\pi_R) = \emptyset$$

ie un port requis et un port fourni d'un même composant ne peuvent pas avoir le même nom. (TODO déplacer)

We can define in a very similar way a function $findNonTransp$ which returns the first non transparent port of a connection chain.

4.5 Operational semantics of programs

Explications :

4.5.1 Connect to

During all the execution, a component can only connect one of its own ports or an external of its subcomponent. Note that v_1 could be changed by v_2 but only the connected port (TODO prove)

$$Var\ \frac{}{\Gamma, \Pi, \Omega, \sigma \vdash_e x \implies \Pi^+_\sigma(x); \Gamma, \Omega}$$

$$Call\ \frac{\begin{array}{c}\Gamma, \Pi, \Omega, \sigma \vdash_e expr \implies val; \Gamma', \Omega' \\ \Gamma', \Pi, \Omega', \sigma \vdash_e arg_1 \implies val_1; \Gamma_1, \Omega_1 \\ \vdots \\ \Gamma_{n-1}, \Pi, \Omega_{n-1}, \sigma \vdash_e arg_n \implies val_n; \Gamma_n, \Omega_n\end{array} \qquad \begin{array}{c}\Gamma_n, \Pi', \Omega_n, \sigma' \vdash_p prog \implies \\ val'; \Gamma'', \Omega''\end{array}}{\Gamma, \Pi, \Omega, \sigma \vdash_e expr.serv(arg_1, ...arg_n) \implies transp_\sigma(val'); \Gamma'', \Omega_n \cup \Omega''}$$

with :

- $val, val_{arg_1}, ..., val_{arg_n} \neq \perp$
- $val \in Dom(\Pi) \cup \sigma\#\pi_R$
- $\sigma' = findFinal_{\Gamma,\Pi,\Omega}(val)\#self_c$
- $\Pi_p = [arg_1 \rightarrow < arg_1, \sigma', val_1\#sigs, val_1\#self_{name}, val_1\#self_c, val\#ap >;$
  $...; arg_n \rightarrow < arg_n, \sigma', val_n\#sigs, val_n\#self_{name}, val_n\#self_c, val\#ap >]$
- $prog = \begin{cases} val_{serv}\#body & \text{if it is defined and if } val_{serv}\#sig =< serv, n > \\ & \quad \text{and } < serv, n > \in val\#sigs \\ \perp & \text{else} \end{cases}$

  where $val_{serv} = \Delta(\Gamma_n(\sigma')\#descr)\#servs(serv)$

Figure 4.1: Semantics of expressions

$$At \; \frac{\Gamma, \Pi, \Omega, \sigma \vdash_e expr \implies val; \Gamma', \Omega'}{\Gamma, \Pi, \Omega, \sigma \vdash_e port@expr \implies val'; \Gamma', \Omega'}$$

with :

$$val' = \begin{cases} \Gamma(pgp\#self_c)\#\pi_X(port) & \text{if it is defined and if } port \in pgp\#ap \\ \bot & \text{else} \end{cases}$$

where pgp is the providing port of val : $pgp = findProv_{\Gamma,\Pi,\Omega}(val)$

$$Path \; \frac{\Gamma, \Pi, \Omega, \sigma \vdash_e expr \implies val; \Gamma', \Omega'}{\Gamma, \Pi, \Omega, \sigma \vdash_e !expr \implies val'; \Gamma', \Omega'}$$

where $val' = findProv(val)$

$$New \; \frac{\Gamma', \Pi, \Omega, max \vdash_e arch \implies (); \Gamma'', \Omega'}{\Gamma, \Pi, \Omega, \sigma \vdash_e descr.new() \implies val; \Gamma'', \Omega'}$$

with :

- $newCompo =< descr\#servs, [\![descr\#\pi_P]\!]_{pp}, [\![descr\#\pi_{ER}]\!]_{erp}, [\![descr\#\pi_{IR}]\!]_{irp} >$
- $max = max(Dom(\Gamma)) + 1$
- $\Gamma' = \Gamma + [max \to newCompo]$
- $val = \Gamma''(max)\#\pi_P(default)$

Figure 4.2: Semantics of expressions

$$\perp\ \frac{}{\Gamma,\Pi,\Omega,\sigma \vdash_p \perp \implies \perp;\Gamma,\Pi,\Omega}$$

$$Seq\ \frac{\Gamma,\Pi,\Omega,\sigma \vdash_p prog1 \implies val1;\Gamma',\Pi',\Omega' \qquad \Gamma',\Pi',\Omega',\sigma \vdash_p prog2 \implies val2;\Gamma'',\Pi'',\Omega''}{\Gamma,\Pi,\Omega,\sigma \vdash_p prog1;prog2 \implies val2;\Gamma'',\Pi'',\Omega''}$$

$$Ret\ \frac{\Gamma,\Pi_{transp},\Omega,\sigma \vdash_e expr \implies val;\Gamma',\Omega'}{\Gamma,\Pi,\Omega,\sigma \vdash_p \textbf{return}\ expr \implies Ret(findNonTransp(val));\Gamma',\Pi',\Omega'}$$

+connect Ω

$$Require\ \frac{}{\Gamma,\Pi,\Omega,\sigma \vdash_p \textbf{require}\ name \implies ();\Gamma,\Pi+[name \rightarrow < name,\sigma,\emptyset,empty,0,\emptyset >],\Omega}$$

$$Connect\ \frac{\Gamma,\Pi,\Omega,\sigma \vdash_e expr_1 \implies v_1;\Gamma',\Omega' \qquad \Gamma,\Pi,\Omega,\sigma \vdash_e expr_2 \implies v_{transp};\Gamma'',\Omega''}{\Gamma,\Pi,\Omega,\sigma \vdash_p \textbf{connect}\ expr_1\ \textbf{to}\ expr_2 \implies v_{res};\Gamma_{res},\Pi_{res},\Omega_{res}}$$

with :

- $v_2 = findNonTransp(v_{transp})$
- $v_{res} = \begin{cases} () & \text{if } v_1\#sigs \subset v_2\#sigs\ \textbf{and}\Big(\\ & \quad \big(v_1\#self_c = \sigma\ \textbf{and}\ v_1\#name \in v_1\#self_c\#\pi_I\ \big)\textbf{or} \\ & \quad \big(v_1\#self_c \in \{irp\#cp_c | irp \in Im(\sigma\#\pi_{IR})\}\ \textbf{and} \\ & \quad\ v_1\#name \in Dom(v_1\#self_c\#\pi_X)\ \big)\ \Big) \\ \perp & \text{else} \end{cases}$
- $(\Gamma_{res},\Pi_{res}) = \begin{cases} (\Gamma'',\Pi+[v_1\#name \rightarrow newPort])) & \text{if } v1\#name \in Dom(\Pi) \\ (\Gamma''+[v_1\#self_c \rightarrow newCompo],\Pi) & \text{else} \end{cases}$
- $\Omega_{res} = \begin{cases} \Omega'' \cup \{v_1\} & \text{if } v_1 \notin Dom(\Pi) \text{ and } v_2 \in Dom(\Pi) \\ \Omega'' & \text{else} \end{cases}$
- $newPort = \begin{cases} < v_1\#name, v_1\#self_c, v_2\#sigs, \\ \qquad v_2\#cp_{name}, v_2\#cp_c, v_2\#ap > & \text{if } v_1\#name \in Dom(\Pi) \\ < v_1\#name, v_1\#self_c, v_1\#sigs, \\ \qquad v_2\#self_{name}, v_2\#self_c, v_1\#ap > & \text{else} \end{cases}$
- $newCompo =< \Gamma''(v_1\#self_c)\#servs, \pi_P, \pi_{ER}, \pi_{IR} >$ where

$$\pi_P\ (resp\ \pi_{ER},\pi_{IR}) = \begin{cases} \Gamma''(v_1\#self_c)\#\pi_P\ (resp\ \pi_{ER},\pi_{IR}) + [name \rightarrow newPort] \\ \quad \text{if } name \in Dom(\Gamma'(v_1\#self_c)\#\pi_P)\ (resp\ \pi_{ER},\pi_{IR}) \\ \Gamma''(v_1\#self_c))\#\pi_P\ (resp\ \pi_{ER},\pi_{IR}) \\ \quad \text{else} \end{cases}$$

Figure 4.3: Sémantique des programmes

Chapter 5

Puissance de calcul

Problématique : comment se passer des primitives?

Il est possible de simuler le $\lambda-$calcul dans Compo:

Listing 5.1: Variable

```
freshVar(x) =
(Descriptor Var extends Component
{
   provides{
      default : {run()}
   }
   requires{
      x : {run()}
   }

   architecture{
      connect x to default;
   }

   service run(){
      "return par terminal"
      return !x;
   }
}).new()
```

Listing 5.2: $\lambda-$abstraction

```
lambda(x,u) =
(Descriptor Lam extends Component
{
   provides{
      default : {run();app(v)}
   }
   requires{
      u#requires - x
   }
   internally requires{
      x :{run()}
```

```
  }

  architecture{
    "si u a un port requis x"
    connect u@x to x;

    "pour tous les autres ports"
    connect u@p to p;
  }

  service run(){
    return default;
  }

  service app(v){
    connect x to v;
    return u.run();
  }
}).new()
```

Listing 5.3: Application

```
app(u,v)=
(Descriptor App extends Component
{
  provides{
    default : {run()}
  }
  requires{
    u#requires + v#requires
  }

  architecture{
    "pour tous les ports de u"
    connect u@p to p;
    "pour tous les ports de v"
    connect v@p to p;
  }

  service run(){
    return u.app(v);
  }

}).new()
```

Appendices

Appendix A

Notations

Ensembles

- Nous confondrons les noms des variables dans la syntaxe abstraite de Compo et les ensembles engendrés par ceux-ci.
- Parties d'un ensemble E : $\mathfrak{P}(E)$
- Nous distinguerons les valeurs de la sémantique et les chaînes de caractère de la syntaxe en utilisant d'un côté des lettres (TODO: trouver bon mot) normales et de l'autre des lettre rondes.
- On pose $\mathscr{L}(E)$ l'ensemble des listes d'éléments d'un ensemble E défini par induction comme suit :
  - $\mathbf{nil} \in \mathscr{L}(E)$
  - $x \in E \implies l \in \mathscr{L}(E) \implies x :: l \in \mathscr{L}(E)$

Application partielles

- $Dom(f)$: domaine de f.
- $f = [i_1 \to v_1; i_2 \to v_2; ...; i_n \to v_n]$: application partielle de domaine $\{i_1, ..., i_n\}$ telle que $\forall k \in [\![1, n]\!] f(i_k) = v_k$.
- $f = [i_1 \to v_1; i_2 \to v_2; ...; i_n \to v_n] \ + \ i_{n+1} \to v_{n+1}$: application partielle de domaine $\{i_1, ..., i_n, i_{n+1}\}$ telle que $\forall k \in [\![1, n+1]\!] f(i_k) = v_k$.

Domaines de valeurs Chaque domaine est donné sous la forme d'un produit d'ensembles. Cependant, afin de distinguer une liste d'arguments et une valeur, nous écrirons une valeur du domaine $A \times B \times C$ avec une notation de la forme $< a, b, c >$ avec $(a, b, c) \in A \times B \times C$.

De plus, nous aurons souvent besoin de désigner les différents champs de ces valeur. Recourir à des projections serait alors illisible. Nous allons donc donner un nom aux champs des valeurs et nous utiliserons la notation $foo\#bar$ pour désigner le champ bar de la valeur foo.

Notation	Value type	Field name	Value domain
$\mathscr{S}_{ig}$	Signatures	$< name, nb_{args} >$	$Name \times \mathbb{N}$
$\mathscr{S}_{erv}$	Services	$< sig, args, body >$	$\mathscr{S}_{ig} \times \mathscr{L}(Name) \times Prog$
$\mathscr{P}$	Ports	$< self_{name}, self_c, sigs,$ $cp_{name}, cp_c, ap >$	$Name \times \mathscr{C}_{ref} \times \mathfrak{P}(\mathscr{S}_{ig}) \times$ $Name \times \mathscr{C}_{ref} \times \mathfrak{P}(Name)$
$\mathscr{C}$	Components	$< descr, \pi_P, \pi_{ER}, \pi_{IR} >$	$Name \times (Name \to \mathscr{P})^3$
$\mathscr{D}$	Descriptors	$< name, super, pp, irp,$ $erp, arch, servs >$	$Name^2 \times List(Port)^3$ $\times Prog \times Name \to \mathscr{S}_{erv}$

Table A.1: Value domains and notations

Voici un tableau récapitulatif des noms des champs pour les différentes valeurs apparaissant dans la sémantique :

Some additional notations : for a component c we will note $c\#\pi = \pi_P + \pi_{ER} + \pi_{IR}$, $c\#\pi_R = \pi_{ER} + \pi_{IR}$, $c\#\pi_X = \pi_{ER} + \pi_P$ and $c\#\pi_I = \pi_{IR} + \pi_{IP}$. All these partial functions are defined because the domains of π_P, π_{ER} and π_{IR} are pairwise disjoints.

Environnements

Appendix B

Vocabulaire

Syntax directed À chaque étape, on ne peut choisir qu'une règle.

Propriété de la sous-formule Toutes les formules qui apparaissent dans une preuve sont des sous-formules de la formule prouvée.

Subcomponent A component A is a subcomponent of a component B iff there is an internally required port of B connected to a provided port of A.

Appendix C

TODO

Expliquer pourquoi la réflexivité aide à écrire une sémantique. (mouais...)

parallèle entre sémantique formelle et dessins

expliciter où sont les primitives

vérifier les dead locks

Non treated in the semantics : ofkind, * , named interfaces, descritor name to specify a signature list, inheritance, collection ports (see Spa13 p; 159).

Define more clearly seq and return

WE : finish inheritance, Define more clearly seq and return, definitive version!!

reflexivity

remove type verifications from the semantics

choice between type connexions or not

choice between Omega or not

Π cannot be changed in expression. This can alleviate the notations. [3] [2]

Bibliography

[1] Arthur Charguéraud. Pretty-big-step semantics. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 41–60, 2013.

[2] Petr Spacek. Design and implementation of a reflective component-oriented programming and modeling language, 2013.

[3] Petr Spacek, Christophe Dony, and Chouki Tibermacine. A component-based meta-level architecture and prototypical implementation of a reflective component-based programming and modeling language. In *CBSE'14, Proceedings of the 17th International ACM SIGSOFT Symposium on Component-Based Software Engineering (part of CompArch 2014), Marcq-en-Baroeul, Lille, France, June 30 - July 4, 2014*, pages 13–22, 2014.