Analyse d'un langage de programmation réflexif orienté composants
Etude de COMPO

Anthony Ferrand Lucas Nelaupe Frédéric Verdier

22 février 2015

Remerciements

Nous souhaitons remercier avant tout Monsieur Dony, notre tuteur lors du projet, pour ses très nombreux conseils et son regard critique et exigeant sur notre travail.
Nous remercions aussi Monsieur Petr Spacek, créateur du langage COMPO, qui a bien voulu nous aider en répondant à toutes nos questions.
Nous remercions également Monsieur Tibermacine, chercheur au LIRMM, pour ses nombreuses idées et interventions dans le cadre de notre projet.

Table des matières

Chapitre I

Introduction

I.1 Définitions

Composant

"A run-time entity which provides and requires services through ports" [Définition de composant dans [**?**]].
Un composant est un élément du programme. Celui-ci sera couplé à d'autres pour former un programme.

Port

"Un port est un point de connexion (les composants sont connectés par leurs ports) et de communication (les invocations de services sont transmises via les ports)." [Définition de port dans [**?**]].
Les composants peuvent communiquer entre eux grâce à des envois de messages via leurs ports.

Connexion

"Describes a binding from one to another port" [Définition de connexion dans [**?**]].
Une connexion représente un lien ou une voie de communication entre deux ports. C'est par ce biais que seront effectué les envois de messages.

I.2 Contexte : La réutilisabilité en développement logiciel

Développer un logiciel est très coûteux. Cependant, certains éléments sont redondants entre tous les logiciels. Le but est de ne concevoir qu'une seule fois ce dont on a besoin. Si un programme a besoin d'un service que l'on a déjà développé pour un précédent logiciel, l'architecte n'aura qu'à reprendre le précédent service déjà développé qui fournit ce dont il a besoin. On peut également imaginer une banque de composants considérés comme fiables par la profession. Cette banque serait disponible via Internet. Cela comporte un double avantage. D'une part, le développement logiciel sera d'autant plus rapide et donc moins coûteux. D'autre part, vu que ces composants sont déjà testés, le programme sera moins sujet aux erreurs.

I.2.1 Comment développer un programme pour que ses éléments soit réutilisables ?

Les différents paradigmes de programmation majoritairement utilisés dans les langages ne permettent pas ou très peu la réutilisation. Actuellement, la réutilisabilité se limite surtout aux bibliothèques de logiciels qui ont été faites pour être réutilisables.

I.2.2 Solution : La programmation orientée composants

Il existe un paradigme de programmation qui facilite la réutilisation : le paradigme composant. Chaque partie du programme est cloisonnée et a des tâches précises. Le tout fonctionne par des envois de messages. Cette séparation permet donc facilement de récupérer une partie du programme qui nous servira pour un autre.

I.3 Objectifs : Comprendre et améliorer le langage COMPO

Il existe actuellement, très peu de langages permettant de faire de la programmation par composants qui spécifient la logique métier d'un programme. En effet la majorité des langages de programmation de ce type sont des ADL (*Architecture Description Language*). Ils ne décrivent que l'architecture générale d'un programme et utilisent des exécutables écrits dans d'autres langages pour la logique métier (Java, C++, ...).
Nous avons cependant trouvé le langage COMPO, actuellement à l'état de prototype, et avons décidé de participer à son développement.

Chapitre II

Le langage COMPO

COMPO est un langage réflexif de programmation et de modélisation orienté composants. Ce chapitre propose une approche pour se familiariser avec le langage.
Nous utiliserons les termes présents dans la version actuelle de COMPO.

II.1 Les descripteurs de composants

Les composants sont décrits par un descripteur. Un composant est une instance du descripteur le décrivant. On peut faire le parallèle avec la programmation orientée objets où les objets sont des instances de classes.
Les descripteurs sont eux-mêmes des composants décrits par leur méta-descripteur (`Descriptor` par défaut).

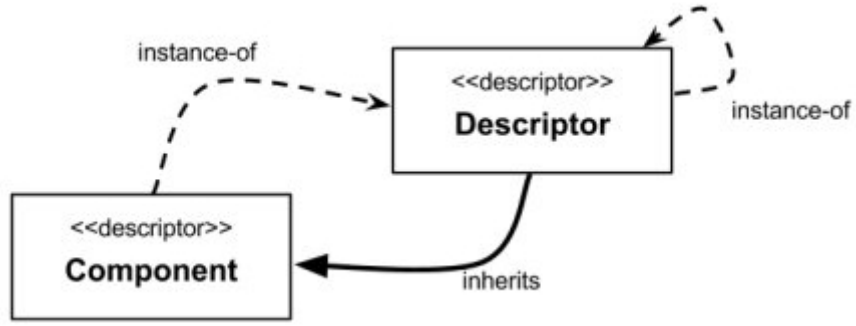


FIGURE II.1 – Métamodèle du langage COMPO extrait de [**?**]

II.1.1 Les informations présentes dans un descripteur

Afin de décrire un composant, il est nécessaire d'expliciter :

Le requis Ce dont un composant a besoin pour fonctionner. Ici ces requis pourront être fournis grâce à d'autres composants.

Le fourni Il s'agit des services que le composant décrit propose. Ces services pourront ainsi être proposés à d'autres composants.

Les services COMPO étant un langage décrivant aussi la logique métier des composants. Cette logique métier se trouve dans les services des composants. Donc le descripteur doit aussi

décrire le contenu des services proposés.

Les connexions entre composants internes Les composants peuvent être eux-mêmes composés d'autres composants. On parle alors de **composites**. L'utilisateur d'un composite ne connaîtra pas ses connexions internes. C'est pourquoi le développeur ayant créé le descripteur doit pouvoir spécifier les connexions internes.

II.1.2 Instanciation d'un composant

Pour créer une nouvelle instance d'un descripteur, il faut utiliser le service `new()`.
Ce service est présent dans tous les composants et renvoie le port fourni par défaut `default` d'une nouvelle instance du descripteur receveur courant.
Voici la syntaxe d'utilisation de `new()` :

Listing II.1 – Syntaxe de création d'un nouveau composant

```
<Descripteur>.new();
```

II.1.3 Exemple de syntaxe d'un descripteur en COMPO

Maintenant que nous avons introduit les concepts de ports, services et descripteurs, nous allons donner la syntaxe générale d'un descripteur.

Listing II.2 – Syntaxe générale d'un descripteur

```
<Meta-descripteur> <Descripteur> extends <Super-descripteur>{
    provides {
        //ports fournis
        <Declaration de port>
    }
    requires {
        //ports requis
        <Declaration de port>
    }
    internally requires {
        //ports internes requis
        <Declaration de port>
    }
    architecture {
        //Connexions interne
        <instructions de connexion>
    }
    //Definition des services
    <Services>
}
```

II.2 Les ports

Un composant peut communiquer avec d'autres composants via ses ports. Un port est défini par un nom, une liste de signature de services proposée, une visibilité, un rôle et une cardinalité.

II.2.1 Rôle

Un port peut avoir deux rôles : fournis (`provides`) ou requis (`requires`).

Fournis

Un port fournit, liste les signatures de services qui sont offerts par un composant.

Requis

Un port requis ; liste les signatures de services que le composant requiert pour fonctionner.

II.2.2 Visibilité

Un port peut être soit interne, soit externe.
Un port interne ne sera pas visible (accessible) depuis l'environnement extérieur du composant. Ces ports servent à réaliser les connexions entre un composite et ses composants internes.
Un port externe est visible depuis l'environnement extérieur du composant. Il peut être connecté à un autre composant.
Par défaut un port est externe. Il faut préciser sa visibilité interne (`internally`) lorsqu'on veut un port interne.

II.2.3 Cardinalité

Un port peut être simple ou multiple.
Un port simple peut être connecté à un autre port tandis qu'un port multiple est une collection de ports simples. Chacun des ports doit avoir la même liste de signatures de services.

II.2.4 Les ports par défaut

Tous les composants possèdent un port fourni externe par défaut nommé **default** qui fourni tous les services publiques du composant.
Ils ont aussi un port interne requis unique nommé **self** listant l'ensemble des services qu'ils possèdent.
Les composants possèdent également un autre port requis interne nommé `super`, connecté au composant descripteur parent.
Enfin ils possèdent un port externe requis multiple nommé `args`. Ce port permet lors d'une invocation de service de stocker les arguments s'il y en a.

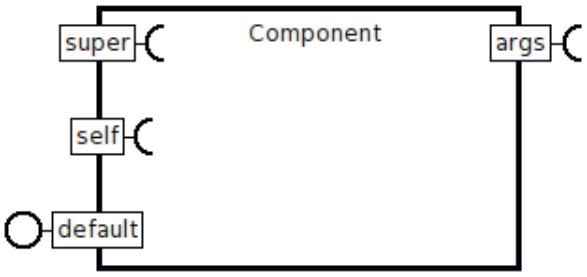


FIGURE II.2 – Schéma d'un composant avec ses ports par défauts

II.2.5 Syntaxe générale de définition d'un port

Voici la syntaxe d'une clause de ports :

Listing II.3 – Syntaxe de la déclaration d'un port

```
<visibilite> <role> {
    //Definition de la liste des services par extension
    <Nom de port>:{ <signature service>; ... };
    //ou par interface/descripteur
    <Nom de port multiple>[]:<Interface||descripteur>;
    ...
}
```

II.3 Les connexions

Les connexions permettent de lier des ports entre eux et ainsi de transmettre des invocations de service entre ports connectés. Les connexions dans COMPO peuvent être de deux types : connexion standard ou délégation.

II.3.1 Standard : requis-fourni

Une connexion classique permet de connecter un port requis avec un port fourni. Le port fourni ne peut être connecté à un port requis que si l'ensemble des signatures de services du port requis est un sous-ensemble de celui des signatures de services du port fourni.

Listing II.4 – Syntaxe de la connexion

```
//Connexion
connect <port>@<composant> to <port>@<composant>;
```

II.3.2 Délégation : requis-requis ou fourni-fourni

Une délégation se fait entre deux ports de même rôle. Elle sert à déléguer l'invocation de service d'un port à l'autre. Cette délégation n'est possible que si l'ensemble des signatures de services du port délégant est au moins un sous-ensemble des signatures de services du port à qui on délégue les invocations.

Listing II.5 – Syntaxe de la délégation

```
//Delegation
delegate <port>@<composant> to <port>@<composant>;
```

II.4 Architecture interne

Un descripteur peut décrire l'architecture interne d'un composite grâce à la clause `architecture`.
Cette clause est une série d'instructions appelées à l'instanciation du composant permettant de créer sa structure interne. On y trouve ainsi l'ensemble des connexions à réaliser pour initialiser le composant dans ses ports internes.

La clause d'architecture interne se définit de la façon suivante.

Listing II.6 – Syntaxe de la définition d'une clause architecture

```
architecture {
    <instructions de connexion>
}
```

II.5 Les services

Les services implémentent la logique métier (ou comportement) du composant. Ils peuvent prendre des paramètres et retourner un composant par un de ses ports fournis.

II.5.1 L'invocation

L'invocation de service se fait à partir d'un port requis connecté à un port fourni passé sous forme d'argument.

Listing II.7 – Syntaxe de l'invocation de service

```
<portRequis>.<service>(<param1>, <param2...>);
```

II.5.2 Les arguments

Tous les composants possèdent un port externe nommé `args`. Lors de l'invocation d'un service, les composants passés en paramètre sont automatiquement connectés à ce port. Lors de l'exécution du service, l'appelé va automatiquement récupérer le composant concerné au port `args`.

II.5.3 Syntaxe générale d'un service

Un service est défini dans un descripteur avec la syntaxe suivante :

Listing II.8 – Syntaxe de la définition d'un service

```
service <Nom>(<Parametre>, ...) {
    <Instructions>
}
```

Chapitre III

Réflexion sur les améliorations possibles de COMPO

III.1 Analyse des connexions

III.1.1 Connexion standard

Elle s'apparente aux passages d'arguments Java. A savoir : le composant passé en argument peut être modifié, mais la variable extérieure gardera un lien sur celle-ci. La variable dans l'appel de la fonction et la variable dans les arguments de la fonction ne sont pas liées. La modification de l'une ne changera pas l'autre. On passe donc une référence vers le composant. Ce comportement est semblable à celui de C++ lorsqu'on utilise un passage d'arguments par &. L'affectation d'une variable de type T& changera la valeur, et non le pointeur vers celle-ci.

III.1.2 Délégation

Elle n'existe pas de façon non-programmatique en Java, mais son comportement peux être simulé par la création d'un objet intermédiaire. Idem pour C++. En effet, pour simuler cette connexion, il faudrait simuler un passage de pointeur *immutable*, qui contiendrai une référence vers l'objet. Un exemple est donné en annexe A avec une implémention en Java.

III.1.3 Connexion partagée : {requis}-fourni

Une connexion {requis}-fourni consiste à connecter un ensemble de ports requis à un port fourni.
Si un des ports requis modifie la connexion, alors les autres ports requis seront modifiés de la même manière.
Contrairement à la délégation, tous les ports peuvent modifier cette connexion. La figure III.1 illustre l'idée proposée.

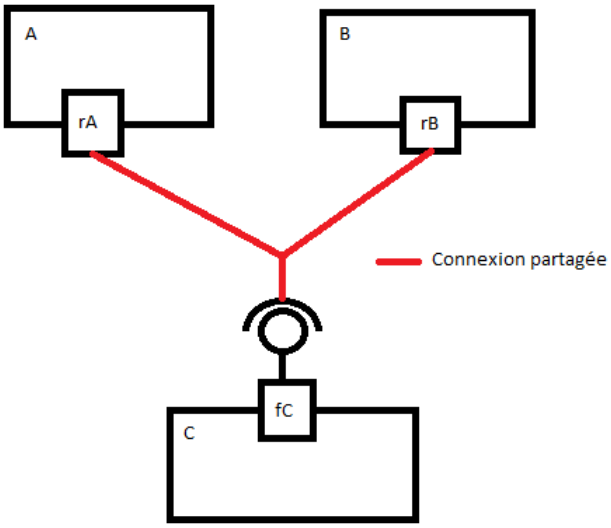
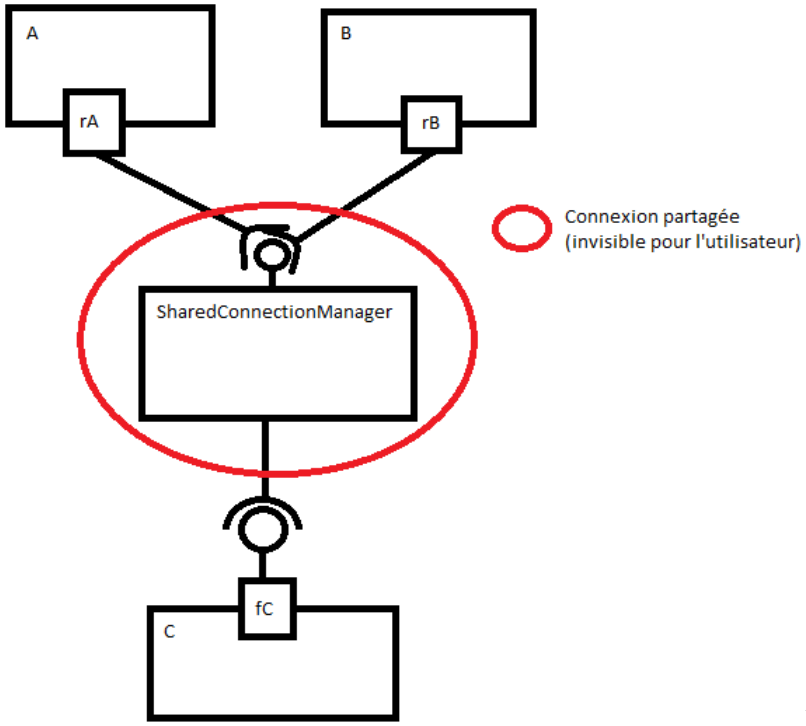


FIGURE III.1 – Schéma d'une connexion partagée

Implémentation

Il n'existe pas de solution simple directement implémentable en COMPO. Voici néanmoins des solutions possibles d'extension du langage.

Solution 1 : Passer par un composant gérant le partage Cette solution implique d'ajouter un composant intermédiaire pour partager une connexion entre plusieurs composants. Voir figure III.2.



R

FIGURE III.2 – Solution 1 : le composant `SharedConnectionManager` gère le partage de la connexion entre `A` et `B` vers `C`

Le composant intermédiaire offre des opérations de connexion, déconnexion de son port requis via son port fourni.
Les composants connectés au composant instance de `SharedConnectionManager` peuvent ainsi modifier par envois de messages la connexion entre le composant intermédiaire et `C` (impactant donc tous les autres composants).
Afin de rendre invisible ce composant intermédiaire, il est nécessaire de modifier les ports requis actuels. Un port requis posséderait un booléen `isSharing`, vrai si et seulement si le port est connecté à un composant de connexion partagée intermédiaire. De plus il faudrait modifier les opérations `connectTo` (pour se connecter à un port fourni) et `disconnect` (pour se déconnecter d'un port fourni) afin de ne pas couper sa connexion avec le composant intermédiaire mais au contraire faire appel à son opération correspondante (si `isSharing` est vrai). A cela devra être rajouté une opération `endSharedConnection` afin de rompre la connexion entre le port et l'instance de `SharedConnexionManager`.
Cette solution peut être actuellement implémentée dans COMPO sans modification du langage si on laisse le composant intermédiaire visible. Cela implique que le contenu des opérations des composants `A` et `B` doit prévoir une telle connexion partagée. C'est pourquoi il est préférable que la gestion de cette connexion partagée soit directement effectuée dans les ports.

Solution 2 : Créer un nouveau type de port requis Cette solution propose d'augmenter le langage COMPO avec un nouveau type de port évitant le passage par un composant intermédiaire. La figure III.3 représente une implémentation de cette solution.

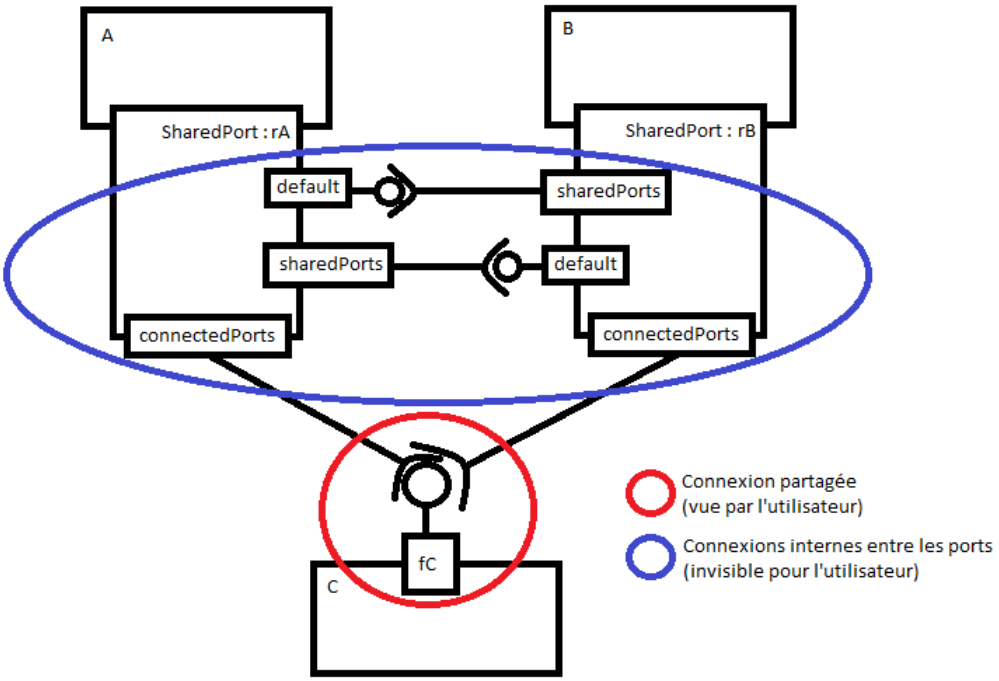


FIGURE III.3 – Solution 2 : on utilise un port de type `SharedPort` pour créer une connexion partagée

Ici on utilise un nouveau type de port nommé `SharedPort`. Celui-ci possède un port requis supplémentaire aux ports classiques `sharedPorts` qui sera connecté au port `default` d'un autre port instance de `SharedPort`.
Ce nouveau port devra redéfinir ses opérations `connectTo` et `disconnect` afin d'appeler l'opération correspondante via son port `sharedPorts`.
Grâce à l'utilisation d'une syntaxe adéquate, il est possible de rendre les connexions entre les ports qui partagent une connexion invisible pour l'utilisateur qui aura l'impression d'utiliser un port classique.

Solution préconisée La solution 2 demande 4 connexions pour chaque paire de connexion. Cela fait donc $2n$ connexions avec n le nombre de composants partagés.
La solution 1 demande une connexion pour chaque composant partagé plus une connexion entre le composant intermédiaire et la cible. On obtient donc $n + 1$ connexions.
La solution 1 est donc plus économe en nombre de connexions dans tous les cas ($n > 1$). Nous préconisons donc l'utilisation de la première solution.

Cas d'utilisation

Un composant ne peut pas anticiper comment il sera utilisé. Cela implique qu'un composant ne peut pas déterminer quel type de connexion sera utilisé sur ses ports requis externes.
Par conséquent, pour une connexion E-fourni où E est un ensemble de ports requis, tout composant ayant un port requis externe appartenant à E ne peut initialiser la connexion (car sinon il anticiperait son utilisation).

Syntaxe proposée

Contrairement à la connexion standard ou à la délégation, la connexion partagée ne peut être implémentée rapidement par un utilisateur du langage COMPO. Il faut donc l'autoriser avec un raccourci syntaxique qui mettra en place automatiquement l'une des solutions proposées.
Par exemple on peut imaginer la syntaxe suivante :

Listing III.1 – Syntaxe de création d'une connexion partagée

```
connect {<portRequis>,<portRequis>} to <portFourni>;
```

Comparaison avec C++

La connexion partagée (voir figure III.1) ne possède pas d'équivalent en Java. Cependant en C/C++ elle s'apparente à l'utilisation de pointeurs. Ces pointeurs pointent sur la même adresse mémoire. Lorsque la valeur à cette adresse est modifiée, elle l'est pour toutes les variables pointant sur cette adresse. On retrouve ce comportement dans la connexion partagée où les ports requis sont des pointeurs pointant sur la même adresse mémoire (ici le port fourni).

Listing III.2 – Exemple en c++

```
int *i = new int;
int *j = i;
*i = 5;
cout << *j << endl; // Affiche : 5
```

III.1.4 Conclusion

L'étude des autres langages a mis le doigt sur le fait que COMPO introduit une nouvelle façon de considérer le "passage de variables". Ce passage inédit est le passage par port requis (délégation). Bien que pouvant être simulé dans Java et C++.
De plus COMPO reprend les deux mécanismes principaux pour récupérer une valeur : par référence Java (connexion requis-fourni) ou par pointeur dé-référencé C++ (connexion {requis}-fourni).

III.2 Passage d'arguments

COMPO est un langage dont la communication entre composants est représentée par l'invocation de service par leurs ports requis.
Un service est très similaire aux méthodes de classe dans un langage orienté objets.
Comme pour une méthode, un service peut prendre des arguments. Actuellement, les arguments des services de COMPO sont des ports requis internes du receveur courant représentés par le port multiple `args`.
A l'invocation du service, on passe alors des ports fournis en arguments qui seront connectés à `args` avant l'exécution du corps. On obtient alors des connexions temporaires entre le receveur courant et les composants passés en argument.
Si on pouvait diversifier le type des paramètres proposés, il pourrait être intéressant de permettre à un même service d'avoir des comportements différents en fonction des types d'arguments proposés (surtout dans le cas où un service utilisera l'argument pour le connecter à un de ses ports).

III.2.1 Par composant

On peut dans un premier temps se demander si un argument représente un port ou un composant.
Dans la programmation orientée composants, on peut communiquer avec un composant que grâce à ses ports. Il ne sert donc à rien de passer en argument un composant directement, car on ne peut que passer par ses ports pour communiquer.
De manière générale, pour référencer une valeur, une "variable" devra toujours passer par un port du composant ciblé. Ainsi un argument ne peut avoir pour valeur qu'un port.

III.2.2 Par port fourni

Dans ce cas on passe en argument un port fourni (mode de passage d'arguments actuellement utilisé dans COMPO), on obtient une architecture représentée par la figure III.4.

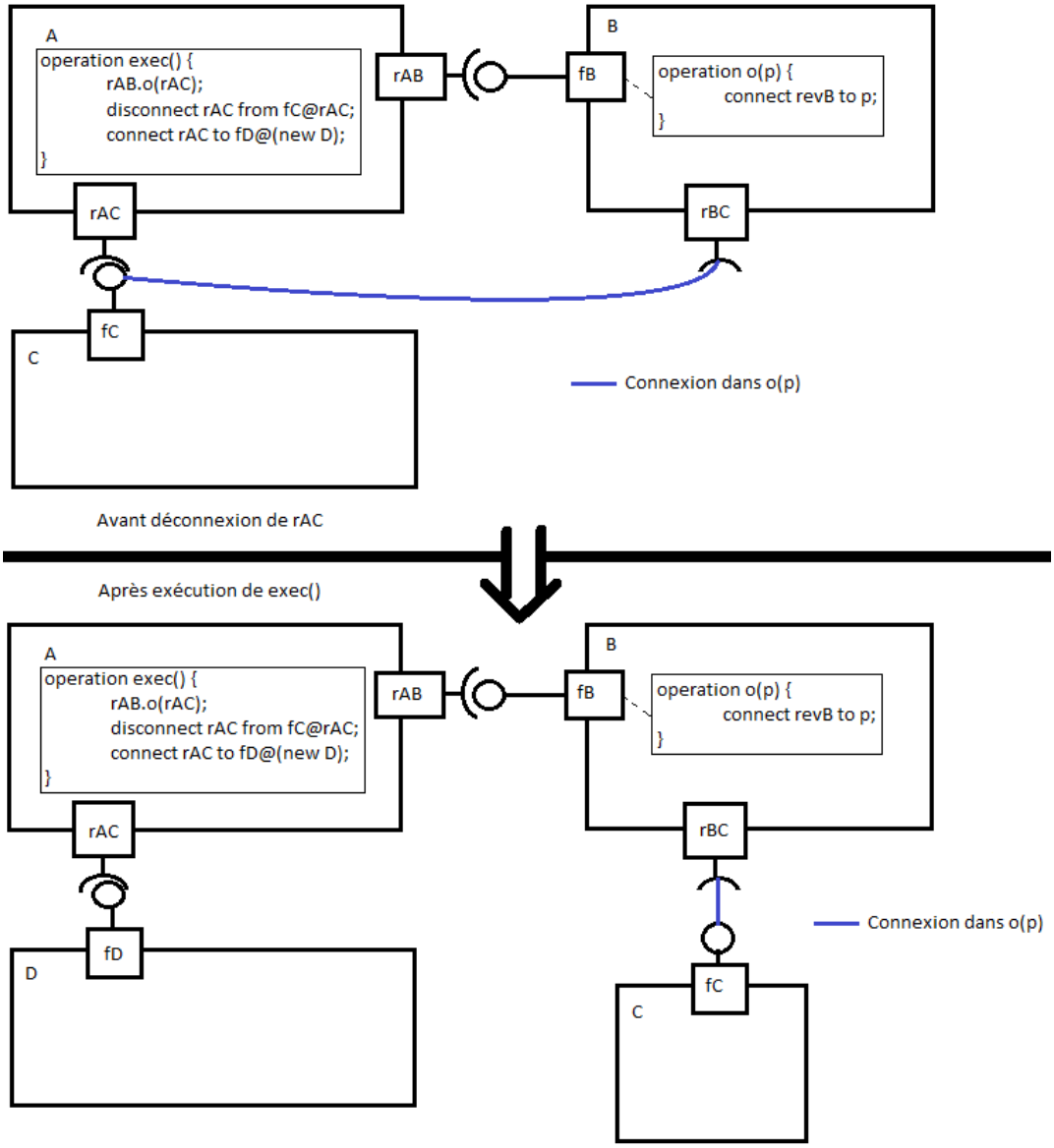


FIGURE III.4 – Exemple lors d'un passage d'un port fourni en argument à `o`

On remarque que la connexion entre `rBC` et `fC` est conservée après la connexion de `rAC` à `fD`. Le passage d'un port fourni est comparable au passage d'arguments par références de Java.

Listing III.3 – Equivalent en Java d'un passage de port fourni en paramètre de `o`

```
1 public void o(Port p) {
2     this.rBC = p;
3 }
```

III.2.3 Par port requis

On suppose pouvoir passer en paramètre de `o` le port requis et non le port fourni auquel il est connecté.
On suppose également que l'instruction "`connect <portRequis> to <portRequis>;`" crée une délégation du premier port vers le second car sinon on ne pourrait pas exécuter `exec()` sans lever une exception.
La figure III.5 montre ce que l'on obtient lors de l'exécution de `exec()`.

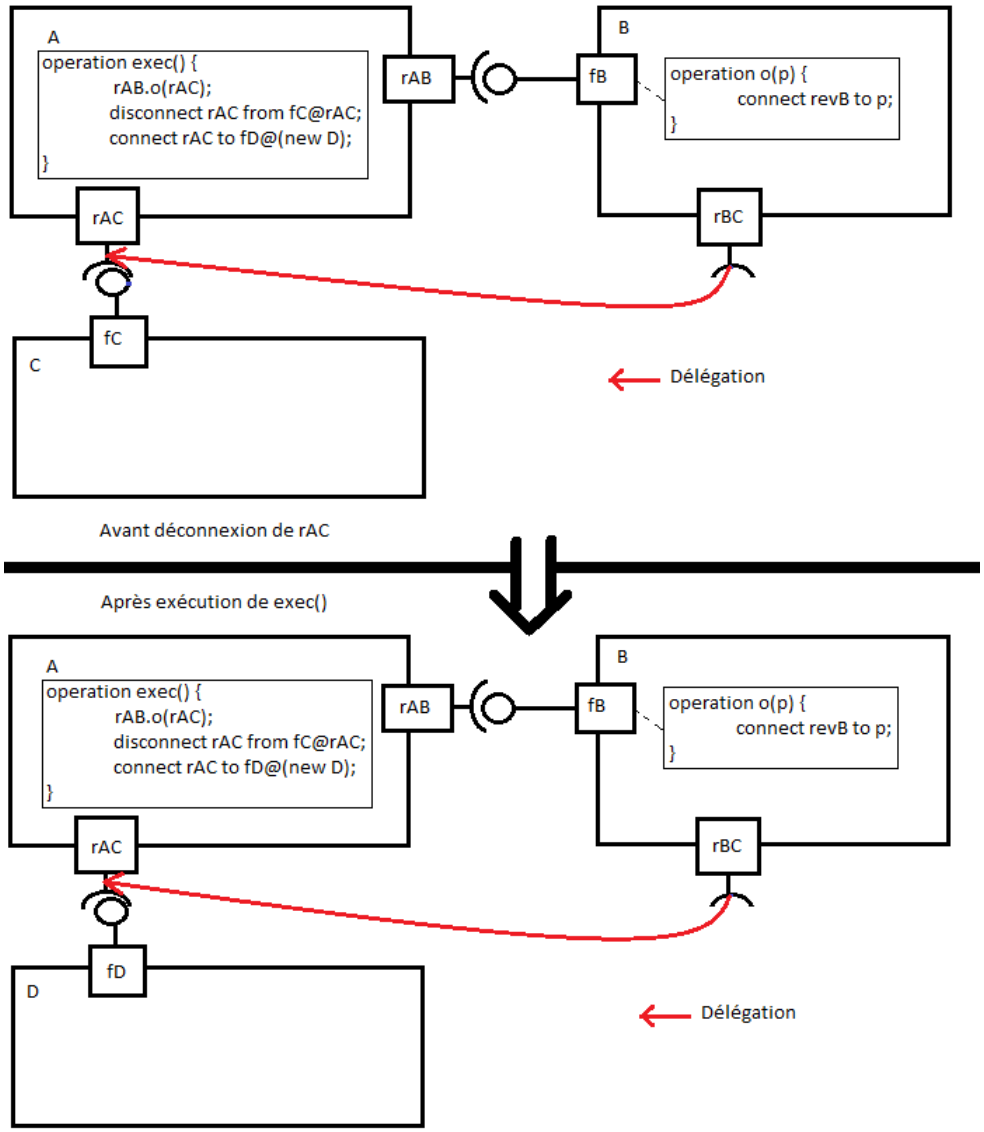


FIGURE III.5 – Exemple lors d'un passage d'un port requis en argument à `o`

On remarque que lors de la déconnexion de `rAC` à `fC`, `rBC` a aussi perdu son accès à `fC`. `rBC` obtient l'accès à `fD` grâce à la délégation.
Le passage d'un port requis en argument est comparable au type de paramètre suivant en C++.

Listing III.4 – Equivalent en C++ d'un passage de port requis en paramètre de o

```
1 void B::o(Port& p) {
2     this->rBC = p;
3 }
```

Cependant la délégation ne possède pas d'équivalent direct en C++. Dans la figure III.5, `rBC` ne peut pas modifier la connexion entre `rAC` et `fC`. Il peut seulement se déconnecter de `rAC` (`this->rBC=NULL` en C++).

III.2.4 Qui choisit le mode de passage ?

Un composant ne peut pas anticiper son utilisation. Il ne peut donc pas anticiper comment il sera connecté à d'autres composants. Il est donc préférable que les composants ne choisissent que leurs connexions lors de l'initialisation et laissent leur conteneur choisir comment ils seront connectés à l'extérieur.
De plus si un composant `A` connecté à `C` donne accès à `C` à un composant `B`, `A` devrait pouvoir choisir comment la connexion doit s'effectuer.
Cependant, il peut exister des cas particuliers dans lesquels l'exécution de l'opération de `B` ne peut pas fonctionner dans tous les cas. Il faut donc laisser un moyen à `B` de contraindre le type des arguments.

Le choix du type de connexions créées par une opération `o` impliquant ses arguments est laissé à l'appelant de `o`. Néanmoins, dans la signature de l'opération, il est possible de restreindre le type des arguments.

III.2.5 Conséquence : Il faut les deux modes de passages

Le passage d'un port fourni en paramètre possède l'inconvénient de créer des architectures peu réactives aux changements. En effet dans l'exemple de la figure III.4, lors du remplacement du composant C par un composant `D` par `A`, la modification n'est pas prise en compte par `B`.
Au contraire, dans le passage d'un port requis, `rBC` reste toujours connecté au port connecté à `rAC` par délégation. Cela crée donc une architecture qui peut être modifiée plus facilement.
Il existe des cas pour lesquels le second mode de passage d'argument ne convient pas. La figure III.6 propose un exemple de cas où la délégation possède un comportement incompatible avec ce que l'on veut faire. Dans cet exemple, `A` invoque deux fois l'opération `add` de `B` en passant soit `value`, soit le port fourni connecté à `value` en paramètre. Entre ces invocations, A change la connexion de `value`. Les connexions en rouge sont créées si on appelle l'opération `add` en passant en paramètre un port requis. Les connexions en bleu sont créées si on appelle l'opération en passant en paramètre un port fourni.

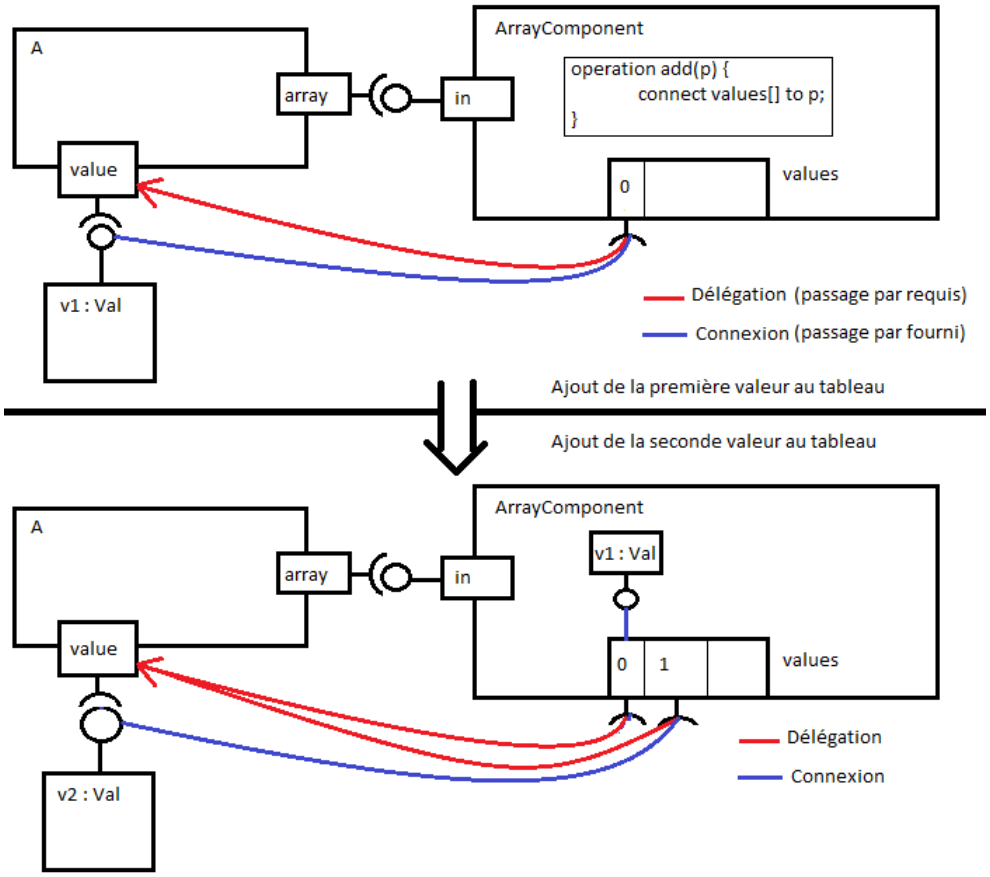


FIGURE III.6 – Cas d'ajout de valeur à un tableau selon que l'on passe un port fourni ou requis en argument de `add`

On remarque que dans le cas où on passe un port requis en argument, la première valeur ajoutée au tableau n'est pas conservée. On obtient à la place deux fois la dernière valeur ajoutée. Il est donc nécessaire de conserver le passage d'argument par port fourni.

On doit donc pouvoir passer en arguments d'une opération des ports fournis ou requis.

III.3 Type de retour d'opération

III.3.1 Du point de vue de l'appelant

Cas de renvoi d'un composant

On peut supposer que le retour d'une opération est un composant.
Dans ce cas, il n'est pas possible d'enchaîner les opérations. Par exemple, on souhaite exécuter : "`portRequis.foo().bar();`". On obtiendra une erreur car `portRequis.foo()` renverra un composant. Or on ne peut invoquer des opérations sur les composants que par un port requis.
De plus l'instruction "`connect monPortRequis to monAutrePortRequis.foo();`" ne fonctionnerait pas également. En effet on ne peut connecter un port qu'à un autre port et non un composant. Ce cas n'est donc pas approprié.

Cas de renvoi d'un port fourni

On peut supposer que le retour d'une opération renvoie un port fourni (par exemple `default`). Dans ce cas, il n'est pas possible d'enchaîner les opérations. Par exemple, on souhaite exécuter : "`portRequis.foo().bar();`".
On obtiendra une erreur car `portRequis.foo()` renverra un port fourni. Or on ne peut invoquer des opérations que sur des ports requis.
Si l'on veut faire cela, il ne faut pas renvoyer directement un port fourni.

Cas de renvoi d'un port requis

Le dernier cas de type de retour d'opération possible est de renvoyer un port requis.
Dans ce cas l'appelant peut enchaîner les invocations d'opérations car on peut appeler des opérations sur les ports requis.
En supposant que l'on accepte la syntaxe "`connect <portRequis> to <portRequis>`", on peut alors écrire : "`connect monPortRequis to monAutrePortRequis.foo();`". Cela créera une délégation de `monPortRequis` vers le résultat de `monAutrePortRequis.foo()`.
L'appel à `new()` ne pose également plus de problème. L'exécution de "`connect monPortRequis to portCible@(ComposantCible.new());`" est possible car :

- `ComposantCible.new()` renvoie un port requis.
- `portCible@portRequis` renvoie le port fourni nommé `portCible` du composant connecté au port requis `portRequis`.
- `connect monPortRequis to portFourni` crée une connexion classique.

L'invocation d'une opération peut donc renvoyer un port requis.

Les retours d'opérations (vus par l'appelant) sont des ports requis.

III.3.2 Du point de vue du receveur courant

Nous avons vu que les opérations renvoyaient des ports requis à l'appelant. Maintenant nous devons nous demander ce que renvoie l'exécution d'une opération. C'est à dire de quel type peut être `x` dans l'instruction "`return x;`".

Cas d'un composant

Dans le cas où on cherche à renvoyer un composant, on se retrouve confronté à un problème. L'appelant attend un port requis. Il faut donc renvoyer un port requis que possède le composant. Or on ne peut pas déterminer quel port choisir automatiquement car il peut exister des composants sans port requis externe.
Il n'est donc pas possible de renvoyer un composant.

Cas d'un port fourni

Dans le cas où on cherche à renvoyer un port fourni, l'appelant attend un port requis. Il est donc nécessaire de faire une instruction implicite supplémentaire.
On propose de connecter le port fourni à renvoyer à un nouveau port requis sans nom, possédé par aucun composant. C'est ce port requis **anonyme** qui serait reçu par l'appelant.
Cette solution peut donc fonctionner mais reste complexe.
On peut aussi supposer que dans le cas où on écrit "`return <portRequis>;`", on renvoie en réalité le port fourni au port requis spécifié. Cela permet de renvoyer tout port fourni connecté à un

port requis (ou à un retour d'opération).
Dans le cas où on stocke la valeur retournée par une opération, on obtient le cas représenté par la figure III.7.

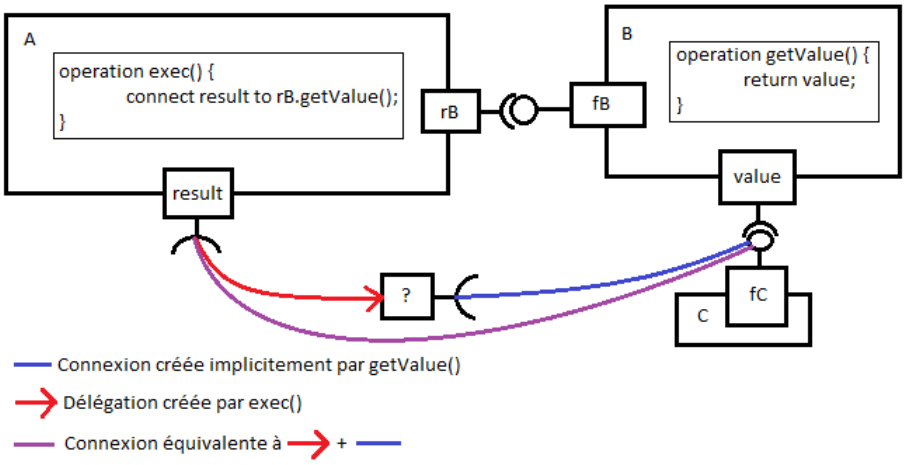


FIGURE III.7 – Cas de renvoi d'un port fourni connecté à un port requis

Lors de l'exécution de "`return value`" dans `getValue()`, l'interpréteur crée un port requis anonyme (sans nom et attaché à aucun composant) connecté à `fC` et le renvoie à `A` (connexion bleue). L'instruction de `exec()` consiste alors en une connexion requis-requis, donc une délégation de `result` vers le port anonyme (délégation rouge).
On peut réduire le nombre de connexions en simplifiant par une seule connexion entre `result` et `fC` (connexion violette). Pour cela il faut modifier l'implémentation de l'opération `connect` dans les ports requis dans le cas où on cherche à créer une délégation vers un port requis anonyme.
Ce cas de figure est équivalent au cas expliqué dans la section III.2.2. En effet on a les propriétés présentes dans les connexions classiques.
Cette solution est très similaire au retour de méthode en Java.

Cas d'un port requis

Nous avons vu que le retour d'une opération était un port requis pour l'appelant. La solution la plus simple est donc de pouvoir renvoyer un port requis côté receveur courant.
Ainsi contrairement au cas précédent, on n'a plus besoin de passer par un port requis anonyme puisqu'on renvoie un port requis.
Si on reprend l'exemple proposé dans le cas précédent avec un retour par port requis, on obtient la figure III.8.

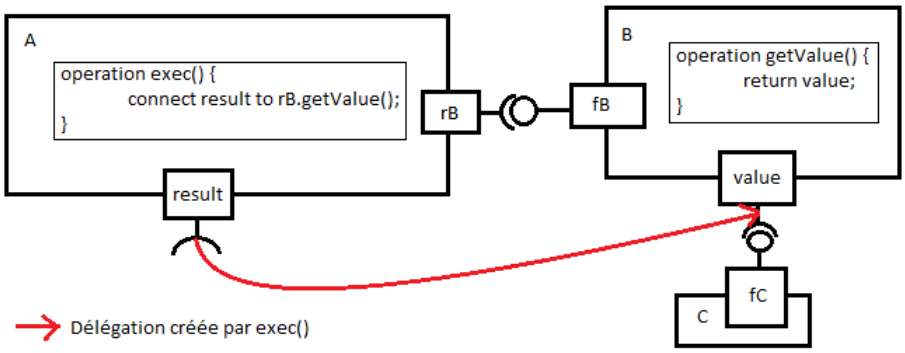


FIGURE III.8 – Cas de renvoi d'un port requis connecté à un port requis

On remarque que la connexion entre `result` et `fC` est similaire, et donc possède les mêmes propriétés, que celle présente dans la section III.2.3. En effet la délégation permet d'obtenir les mêmes propriétés.
Cette solution pose un problème. Car c'est `B` qui choisit ce qu'il renvoie. Il peut donc modifier la connexion impliquant `value`, qui modifiera par la délégation la connexion de `result`. Or `B` ne doit pas pouvoir anticiper comment il sera utilisé. Il y a donc une contradiction.
C'est pourquoi nous pensons que ce mode de connexion n'est pas utilisable.

Renvoyer un port fourni

D'après les paragraphes précédents, on peut dire qu'il est préférable de renvoyer un port fourni. Celui-ci sera alors connecté à un port requis anonyme et passé à l'appelant.

Il peut être intéressant pour des questions de performances d'éviter les délégations d'un port requis vers un port anonyme. Lorsqu'un cas pareil se présente (voir exemples III.7 et III.8), la connexion est effectuée entre le port requis et le port fourni connecté au port anonyme. Ce dernier est ensuite détruit.

III.3.3 Cas particulier : l'opération new()

Afin de créer une nouvelle instance de composant, on invoque l'opération `new` sur son descripteur.
D'après III.3.1 , `new()` renvoie un port requis anonyme connecté à un port fourni `default` du composant nouvellement créé. Ce port anonyme possède les mêmes propriétés qu'un port requis nommé. Il peut, par exemple, servir comme tout autre port à identifier le composant auquel il est connecté. Cela est notamment nécessaire pour connecter un port fourni spécifique d'un nouveau composant à un de ses ports requis.

```
1 connect monPortRequis to portFourniCible@(DescripteurCible.new());
```

III.4 Les services et l'encapsulation

III.4.1 Modification de la connexion entre deux composants distants

Dans un service, il est actuellement possible de modifier une connexion d'un composant distant au travers d'autres composants. Cette utilisation constitue un abus du langage COMPO et ne devrait pas être permis. En effet, cela constitue une infraction au principe d'encapsulation.

Il est possible pour un composant d'utiliser la couche réflexive pour connaître les composants qui sont connectés à un port et ainsi pouvoir modifier des connexions à l'extérieur de lui-même.

Sinon, il faut connaître l'architecture du programme, un exemple de cette déconnexion distante est donnée en annexe C.

III.4.2 Modification de connexion d'un port externe

Prenons un compteur simple représenté par la figure III.9. Ajoutons lui un service `foo` qui modifie la connexion d'un port externe comme dans III.5.

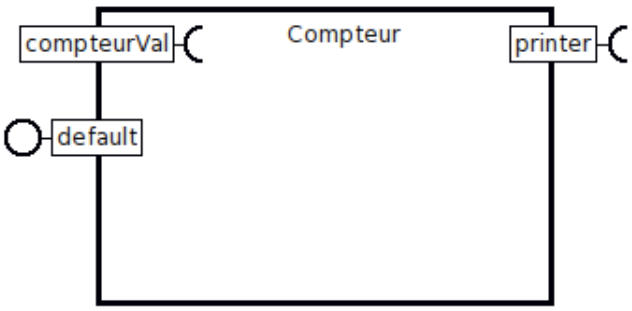


FIGURE III.9 – Schéma d'un compteur extrait de `Compobrowser2`

Listing III.5 – Exemple d'un service dans un composant compteur

```
1 service foo {
2     connect printer@self to default@(MorphPrinter.new());
3 }
```

Cet exécution permet de connecter le port requis externe `printer` à un nouveau `printer`, dans ce cas un `MorphPrinter`

Quand on exécute le programme, le résultat ne sera pas celui attendu car le `printer` utilisé ne sera pas celui défini par l'architecte lors de la création du programme. En effet, le `printer` connecté sera modifié lors de l'exécution du service `foo`.

Nous pensons qu'il ne devrait pas être possible pour un composant de modifier une connexion sur un port externe dans un service

Pour cela, on pourrait interdire l'utilisation du mot clé `@` dans les services si le port est un port externe.

III.5 Le typage en COMPO

Actuellement, COMPO est un langage à typage dynamique. Il peut être intéressant de voir les points forts et points faibles des typages possibles en COMPO afin d'en préconiser celui que nous estimerons le plus intéressant.

III.5.1 Le typage dynamique

Le typage dynamique permet de programmer rapidement avec une syntaxe minimale des programmes complexes.
Dans COMPO ces points forts sont

- Syntaxe minimale : ne pas préciser les types pour les variables ou les signatures d'opérations permet de garder une syntaxe minimale et lisible.
- Retrait des problèmes du polymorphisme : grâce au typage dynamique, il est possible de connecter un port requis à un port fourni peu importe d'où il vient.

Néanmoins les inconvénients sont similaires à ceux que l'on peut retrouver en langage à objets à typage dynamique :

- Code peu lisible : l'absence de typage implique qu'il est difficile de reprendre un code créé par une autre personne. Il est difficile de savoir quel type d'argument est demandé par une opération par exemple.
- Vérification de la cohérence du code à l'exécution : on ne peut anticiper les erreurs liées au typage à la compilation. On est obligé de faire des tests afin de vérifier que le code est cohérent.

III.5.2 Le typage statique

Le typage statique propose des codes plus complexes mais aussi plus précis.
L'utilisation du typage statique en COMPO permettrait

- Vérification de la cohérence du code à la compilation : le typage statique permettrait de vérifier à la compilation d'un descripteur si les connexions sont cohérentes par exemple.
- Signatures d'opérations plus lisibles : ajouter des types aux arguments et au retour des opérations, permettrait d'expliciter ce qu'attend l'opération pour fonctionner.

Cependant cette solution possède des inconvénients :

- Typage très contraignant sur l'origine des composants : le fait de typer statiquement les arguments ou les ports, restreint l'origine des composants. En effet en typage statique il faut que le type d'une variable et ce qu'elle contient possède une relation d'héritage (polymorphisme). Cette relation d'héritage implique qu'il est difficile de mettre en relation des composants d'origines diverses.
- Code plus lourd : COMPO cherche à rester minimaliste et l'ajout du typage statique rendrait la syntaxe beaucoup plus lourde (surtout sur les signatures d'opérations).

III.5.3 Notre solution : le typage par interface

Typer les arguments d'opérations par une liste d'opérations comme pour les ports. On peut retrouver ce type de vérification de compatibilité dans certains ADL comme DEDAL [**?**].
On obtiendrait ainsi :

- Un code lisible pour les opérations : l'ajout de ce typage permet de mieux comprendre ce dont a besoin l'opération pour fonctionner.
- Une vérification de la cohérence du code à la compilation : il est possible de vérifier la compatibilité des connexions et des passages d'arguments à la compilation en comparant les interfaces (anonymes ou nommées).
- Retrait des problèmes liés au polymorphisme : il n'y a plus de problème lié aux relations d'héritage dans le typage par interface puisqu'on compare la liste des signatures d'opérations.

Chapitre IV

Nos contributions au langage COMPO

Nous présentons dans ce chapitre une partie de nos contributions.

IV.1 Connexion à des littéraux

Il n'était pas possible de connecter un port directement à un littéral. Nous souhaitons donc rajouter cette possibilité. Cela permet au développeur de facilement connecter un port à un nombre (entier ou à virgule). On peut également se connecter directement à une chaîne de caractère.

Pour faciliter l'implémentation, nous avons ajouté de manière temporaire le mot clef `rockonnect`. Le littéral *parsé* est connecté au port par l'intermédiaire de la création d'un `LiteralAtomicPort`, une spécialisation de `AtomicPort`

Par la suite, nous avons fusionner le mot clef `rockonnect` et `connect`. Il est donc désormais possible pour un développeur d'utiliser le mot clé `connect` pour se connecter aux littéraux. Le mot clé `rockonnect` a depuis été supprimé.

Listing IV.1 – Syntaxe d'une connexion à un littéral

```
1 connect compteurVal@self to 1;
2 connect compteurVal@self to 3.14;
3 connect compteurVal@self to 'Hello World';
```

IV.2 Enchainement des `connect` sur un même port

Il n'était jusqu'à lors pas possible d'enchaîner les `connect` sur un même port. Seul le premier `connect` était prit en compte. Il fallait utiliser entre chaque modification le mot clé `disconnect`.

Implémentation du `disconnect`

Le mot cléf `disconnect` existait déjà dans COMPO mais n'était pas implémenté. Nous avons donc ajouté la logique métier. Celui-ci permet de couper une connexion entre deux ports.

Listing IV.2 – Syntaxe d'une déconnexion

```
1 disconnect compteurValSyndicat@self from default@compteurValSyndicat
```

Cet ajout va nous permettre de pouvoir par exemple modifier la valeur d'un compteur.

L'utilisation de ce mot clé est problématique dans la mesure où il nécessaire de savoir si un port est connecté à quelque chose ou pas avant de pouvoir le connecter à un autre composant.

Ajout du `disconnect` automatique lors d'une connexion

Nous souhaitons que l'utilisateur puisse modifier une connection sans utiliser le mot clé `disconnect`. Nous avons donc rajouté l'appel au `disconnect` de manière automatique avant chaque connexion.

Listing IV.3 – Syntaxe d'une connexion entre un port et un littéral

```
1 connect compteurVal@self to 1;
2 connect compteurVal@self to 2;
3 connect compteurVal@self to 3;
```

Cet enchaînement permet de se connecter au port `1`, `2`, puis `3`. Cet abstraction permet à l'utilisateur de ne pas utiliser le mot clé `disconnect` entre chaque opération. Celui-ci sera fait automatiquement par COMPO.

IV.3 Nouvelle syntaxe d'une connexion

A la manière de l'affectation SmallTalk, nous avons implémenté une nouvelle syntaxe de connexion. Nous avons choisi >>> qui peut donc remplacer le traditionnel `connect to`. Cette syntaxe moins verbeuse permet également de s'abstraire du `@self`.

Listing IV.4 – Comparaison entre l'ancienne et la nouvelle syntaxe d'une connexion

```
1 "Nouvelle syntaxe"
2 compteurVal >>> 1;
3
4 "Ancienne syntaxe"
5 connect compteurVal@self to 1;
```

Les deux syntaxes sont actuellement valides pour définir une affectation.

IV.4 Ajout de primitives SmallTalk en COMPO

COMPO étant basé sur SmallTalk, il peut être intéressant pour un développeur de pouvoir utiliser des primitives. Nous avons donc rajouté une syntaxe qui permet d'indiquer à l'analyseur syntaxique que ce qui suit est une primitive SmallTalk.

Listing IV.5 – Syntaxe d'une primitive smalltalk en COMPO

```
1 <!
2     Transcript crShow: 'Hello'.
3 >
```

IV.5 Définir un `service` avec le mot clé `operation`

Lors de discussions avec des non initiés au langage COMPO, l'utilisation du mot clé `service` comportait des ambiguïtés. Pour faciliter la compréhension du langage, nous avons rajouté le mot clé `operation` pour définir un service. Ce dernier permet de faire le paralléle plus facilement avec les autres langages.

Listing IV.6 – Syntaxe d'une opération en COMPO

```
operation foo() {
    "TODO"
}
```

Il est donc possible d'utiliser les deux mots clés dans le langage pour définir un service.

IV.6 Vérification de compatibilité lors des connexions

Nous avons donc rajouté une vérification des compatibilités de connexions.

A la compilation

Quand le développeur fait un *accept*, une erreur lui indique s'il a fait une connexion entre deux composants incompatibles.

Listing IV.7 – Exemple de connexion incorrecte

```
internally requires {
    myPort : { foo() };
}
architecture {
    connect myPort@self to default@(MyComponent.new());
}
```

Si `MyComponent` n'implémente pas de méthode `foo`, lors du `accept`, le développeur aura le message d'erreur suivant :

Listing IV.8 – Message d'erreur lorsque les interfaces sont incompatibles

```
Incompatible interfaces between myPort@self and default@(MyComponent.new());
```

A l'exécution

Une seconde vérification de comptabilité est faite à l'exécution car il est possible d'effectuer un `connect` dans un service.

Listing IV.9 – Extrait d'un descripteur Compteur

```
Descriptor Compteur extends Component
{
   requires {
      printer: { foo(); };
   }
}
```

Le compteur a besoin d'un `printer` qui possède une méthode `foo`.

Listing IV.10 – Service qui lancera une exception lors de son invocation

```
Descriptor Program extends Component
{
   service init() {
      connect printer@self to default@(Printer.new());
   }
}
```

L'exécution de ce service *init* plantera à l'exécution si `Printer` ne possède pas de service `foo`. L'invocation du service `foo` aura donc pour conséquence d'arrêter l'exécution et de déclencher l'erreur suivante.

Listing IV.11 – Message d'erreur lorsque les interfaces sont incompatibles

```
Incompatible interfaces between printer and printing
```

IV.7 Retour par port requis ou fourni

Par port requis

Nous souhaitons que le développeur puisse chaîner les opérations. Or, les invocations d'opérations ne peuvent se faire que sur des ports requis. Il faut donc qu'une opération rende un port requis.

Le code suivant permet de récupérer le composant `Printer` connecté au compteur et d'invoquer le service `print()` directement.

Listing IV.12 – Syntaxe d'un chaînage d'invocation en COMPO

```
compteur.getPrinter().print('Hello');
```

Listing IV.13 – Syntaxe de retour d'un port requis en COMPO

```
service foo() {
    return !portInterne;
}
```

Cette syntaxe permet de retourner un port requis directement.

Par port fourni

Il peut être intéressant de retourner un port fournis dans le cas où l'on souhaite réutiliser le résultat d'un service.

Listing IV.14 – Syntaxe de retour d'un port founis en COMPO

```
service foo() {
    return 1;
}
```

Lors de la définition d'un service, un composant peut choisir de retourner directement un port requis. Nous donnons en Annexe D un exemple pour montrer la différence de comportement entre ces deux modes de retour.

IV.8 Précisions du type de retour dans la signature d'une opération

Nous avons ajouté une syntaxe qui garanti le retour d'un port requis ou fourni par les mots clés `:req` et `:pro`.
Cette syntaxe a, avant tout pour but, de spécifier le comportement d'une opération sous la forme d'une assertion en précisant si elle renvoie un port requis ou fourni. Si l'assertion n'est pas vérifiée, une exception est levée.

Listing IV.15 – Syntaxe du typage de retour d'opération

```
operation foo() :pro {
    return 1;
}
operation bar() :req {
    return !aRequiredPort;
}
```

IV.9 Passage d'arguments par fourni ou requis

Nous avons implémenté la possibilité de choisir le mode de passage. Si l'on précède la variable par un `!`, cela revient à passer le port directement. Le choix du mode de passage est totalement transparent pour le receveur.

Listing IV.16 – Syntaxe du passage par founis en COMPO

```
a.foo(var);
```

Listing IV.17 – Syntaxe du passage par requis en COMPO

```
a.foo(!var);
```

Imaginons que le composant `A` enregistre cette `var` grâce à la méthode `foo`. Si on change ce qui est connecté à mon port `var` à l'extérieure de `A` il sera également modifié dans `A`.

Nous donnons en Annexe E un exemple pour montrer la différence de comportement entre ces deux modes de passages.

Chapitre V

Conclusion

V.1 Le langage COMPO

COMPO est actuellement l'un des seuls langages de programmation orienté composants qui soit réflexif. Cette spécificité nous a amené à étudier son fonctionnement.
COMPO étant à l'état de prototype, nous avons proposé des améliorations possibles du langage. Nous en sommes arrivés aux conclusions suivantes.

Analyse des connexions

Il existe trois modes de connexion possibles en COMPO :

Connexion standard C'est une connexion entre un port requis et un port fourni. Ces connexions ne sont pas impactées par des modifications de connexions extérieures.

Délégation C'est une connexion entre deux ports requis ou deux ports fournis. Une délégation peut être impactée par des modifications de connexions extérieures. On peut rendre une architecture plus simple à maintenir par des délégations.

Connexion partagée Cette connexion n'est pas implémentée actuellement dans COMPO. Il s'agit d'une connexion entre plusieurs ports requis vers un port fourni. Lorsque le possesseur d'un des ports requis modifie sa connexion, la modification est transmise aux autres connexions.

Passage d'arguments

Nous avons vu qu'il était possible de passer en argument un port fourni ou un port requis. Selon le mode de passage choisi (par l'appelant), le comportement de l'opération côté appelé change.

Type de retour d'opération

Nous avons vu qu'il existait deux modes de retour possible pour les opérations du point de vu du receveur courant : retour d'un port requis ou d'un port fourni.
Du côté de l'appelant, ce dernier ne reçoit que des ports requis afin de pouvoir enchaîner les invocations d'opérations.

Les services et l'encapsulation

Nous avons mis en évidence un problème lié à l'accès à un port distant par l'opérateur `@` de COMPO dans les opérations.
Cela brise l'encapsulation des composants. Cependant nous avons précisé qu'il s'agit d'un cas particulier. En effet pour accéder à un port distant, le composant doit avoir conscience de l'architecture globale du programme. Il peut en avoir conscience par introspection.

Le typage en COMPO

Nous avons réfléchi sur le mode de typage que devrait adopter COMPO. Actuellement en typage dynamique, nous avons conclu qu'il était préférable d'opter pour un typage "hybride" entre le typage statique et le typage dynamique. Ce mode intermédiaire ne vérifierait la compatibilité qu'au niveau des signatures d'opérations que doit fournir (ou requiert) un port fourni (ou un port requis).

V.2 Nos contributions

Nous avons souhaité contribuer au langage COMPO pour améliorer sa syntaxe, mais également ses possibilités. Voici un rapide résumé des apports développés au sein de ce projet.

Connexion à des littéraux

Nous avons rajouté la possibilité de connecter un port directement à un littéral SmallTalk (nombre ou chaîne de caractère).
Pour cela nous avons créé un nouveau type de port atomique fourni nommé `LiteralAtomicPort` héritant de `AtomicPort`.

Enchaînement des connexions sur un même port

Nous avons permis l'enchaînement des connexions sur un même port. Auparavant, lorsqu'un port était déjà connecté, il ne se connectait pas au nouveau port demandé tant qu'il n'était pas au préalable déconnecté.

Nouvelle syntaxe d'une connexion

Nous avons pu remarquer qu'une connexion était très similaire à une affectation dans d'autres langages comme le C. Nous avons donc augmenté la syntaxe des connexions afin de montrer visuellement ce lien tout en les distinguant par l'opérateur >>>.

Ajout de primitives SmallTalk en COMPO

Afin de pouvoir ultérieurement améliorer les opérations de la couche réflexive de COMPO, il était nécessaire de pouvoir utiliser du code SmallTalk directement en COMPO. Nous avons donc augmenté la syntaxe afin de créer de telles sections de code.

Définir un `service` avec le mot clé `operation`

Nous pensons qu'il est nécessaire de changer le mot clé `service` par le mot clé `operation` pour éviter les confusions avec les *webservices* (complètement différents).

Vérification de compatibilité lors des connexions

Afin d'implémenter le typage "hybride" proposée dans nos réflexions sur le typage de COMPO, nous avons implémenté une vérification de compatibilité des connexions.
On a ainsi implémenté une vérification à la compilation pour la clause `architecture` du programme. Une autre vérification est réalisée à l'exécution pour les connexions effectuées dans les opérations.

Retour par port requis ou fourni

Nous avons implémenté la possibilité de renvoyer un port fourni connecté à un port requis ou directement un port requis grâce à l'utilisation du mot clé "`!`".
L'appelant reçoit dans tous les cas un port requis (pouvant être anonyme si le receveur courant de l'opération invoquée renvoie un port fourni) afin de pouvoir enchaîner les invocations d'opérations.

Précisions du type de retour dans la signature d'une opération

Nous avons ajouté une syntaxe optionnelle faisant office d'assertion sur le type de retour d'une opération. En utilisant les mots clés `:req` ou `:pro` dans la signature d'une opération, on précise ce qu'elle renvoie.

Passage d'arguments par fourni ou requis

De la même manière que pour le retour par port requis ou fourni, l'opérateur "`!`" nous a permis de pouvoir passer en argument des ports fournis ou des ports requis. Selon le mode de passage choisi, le comportement de l'opération en est modifiée.

V.3 Difficultés rencontrées

Le code généré

Une partie du langage COMPO est basé sur du code généré. Lors de nos ajouts, cette spécificité nous a très vite posé problème. En effet, lorsque l'on fait une modification, le code est à nouveau généré. Or, certaines portions du code généré avait été modifiées "à la main" pour effectuer des correctifs. Lorsque nous avons essayé de modifier le code de base, celui généré ne comportait plus les correctifs.

Le retour par requis

Pour renvoyer un port requis dans une opération et non ce à quoi il est connecté, nous avions pensé dans un premier temps à rajouter un opérateur `:req` dans la signature du service.
Ci-dessous, on suppose que le composant `Cpt` possède un port requis `A`, connecté à un autre composant. Cet exemple renvoie le port requis `A` et non ce à quoi il est connecté.

Listing V.1 – Syntaxe d'une opération avec `:req` en COMPO

```
1 operation foo() :req {
2     return A;
3 }
```

On a remarqué que l'ajout du `!` par la suite posait problème car les deux opérateurs peuvent répondre à la même demande. Nous avons donc modifié `:req` pour le transformer en une assertion. Si `foo()` ne retourne pas un port requis, le programme lève une exception.

V.4 Perspectives

Notre problématique étant complexe, la solution qui y répond est complexe aussi. Notre étude nous a permis de comprendre les difficultés en question. Ainsi, les perspectives d'avenir de notre projet sont multiples et variées.

Augmenter la vérification de compatibilité

Actuellement, la vérification de compatibilité ne fonctionne que sur les interfaces anonymes. Le port `default` généré sur tous les composants est toujours attribué avec une interface "*". Hors une interface nommée est l'interface du port `default` du composant de même nom (donc "*").
Il faudrait donc modifier comment est générée l'interface du port `default` afin de pouvoir avoir une vérification de compatibilité dans tous les cas.

Implémenter la connexion partagée

Nous avons vu dans III.1.3 qu'il pourrait exister un troisième type de connexion : la connexion partagée.
Elle n'est pas implémentée dans COMPO actuellement. On peut la simuler mais il n'y a pas de raccourci syntaxique pour en créer une automatiquement.
L'ajout d'une telle connexion demanderait d'augmenter la syntaxe du `parseur`, et d'ajouter de nouveaux types de ports requis.

Implémenter les opérations de la couche réflexive

COMPO est un langage réflexif. Cependant les opérations de la couche réflexive ne sont pas toutes implémentées. On effectue actuellement l'introspection directement en SmallTalk en suivant les relations créées par COMPO.
Ces opérations ne poseraient pas de problèmes pour être implémentées grâce au fait qu'on puisse écrire du code SmallTalk directement dans une opération par la syntaxe `<! ... >`.

Interdire l'accès à un port distant auquel on n'est pas connecté

On a vu dans III.4.1 que l'accès à un port distant auquel on n'est pas connecté pouvait poser problème dans une opération. Il peut être intéressant d'interdire ces accès par le mot-clef `@` et forcer l'utilisation d'opérations servant d'accesseurs.

Extraire COMPO de Pharo

Actuellement, COMPO fonctionne en environnement fermé dans Pharo. Pouvoir s'extraire de cet environnement pourrait permettre une exportation plus simple et une utilisation d'outils externes plus élaborés. Par exemple on pourrait utiliser COMPO dans Eclipse et obtenir toute la puissance d'un tel environnement de développement.

Appendices

Annexe A

Parallèle entre COMPO et JAVA pour une délégation

Nous allons réaliser en COMPO puis en Java un exemple de délégation. On prend un cameraman, un interviewer et une personne qui est interviewée. Le but est que le cameraman filme en permanence la personne que l'interviewer est en train d'interroger. L'interviewer va donc commencer à discuter avec une `ExpressivePerson` puis une `SadPerson`". Le but est que le cameraman et l'interviewer aient toujours le *focus* sur la même personne.

COMPO

Listing A.1 – Cameraman

```
Descriptor Cameraman extends Component {

    provides {
        listen: { listen(txt) };
    }

    requires {
        chat: { chat(txt); };
    }

    architecture {
        connect chat@self to chat@(Interviewer.new());
    }

    service listen() {
        Transcript.show('The cameraman is listening... ');
        Transcript.crShow(chat@self.chat());
    }
}
```

Listing A.2 – Interviewer

```
Descriptor Interviewer extends Component {

    provides {
        interview: { interview(txt) };
    }

    requires {
        chat: { chat(txt); };
    }

    architecture {
        connect chat@self to chat@(SadPerson.new());
    }

    service interview() {
        Transcript.show('Hello, what do you think? ');
        Transcript.crShow(chat@self.chat());
    }
}
```

Listing A.3 – SadPerson

```
Descriptor SadPerson extends Component {

    provides {
        chat: { chat(txt); };
    }

    service chat(txt) {
        return 'You will fail your life.';
    }
}
```

Listing A.4 – ExpressivePerson

```
Descriptor ExpressivePerson extends Component {

    provides {
        chat: { chat(txt); };
    }

    service chat(txt) {
        return 'I am in love with you.';
    }
}
```

Java

Listing A.5 – IChatty

```java
package compo;

public interface IChatty {
    String chat();
}
```

Listing A.6 – SadPerson

```java
package compo;

public class SadPerson implements IChatty {
    @Override
    public String chat() {
        return "You will fail your life.";
    }
}
```

Listing A.7 – ExpressivePerson

```java
package compo;

public class ExpressivePerson implements IChatty {
    @Override
    public String chat() {
        return "I'm in love with you.";
    }
}
```

Listing A.8 – ChattyPointer

```java
package compo;

public class ChattyPointer<T extends IChatty> {
    private T m_data;

    public T data() {
        return this.m_data;
    }

    public void define(T data) {
        this.m_data = data;
    }
}
```

Listing A.9 – Cameraman

```java
package compo;

public class Cameraman {
    private IChatty chater;
    private ChattyPointer<IChatty> target;

    public Cameraman() {
        this.chater = null;
        this.target = null;
    }

    public void setChater(IChatty chater) {
        this.chater = chater;
    }

    public void setTarget(ChattyPointer<IChatty> target) {
        this.target = target;
    }

    public void listen() {
        System.out.print("The cameraman is listening... ");
        System.out.println(this.chater.chat());
    }

    public void smartListen() {
        System.out.print("The cameraman is listening... ");
        System.out.println(this.target.data().chat());
    }
}
```

Listing A.10 – Interviewer

```java
package compo;

public class Interviewer {
    private IChatty chater;
    private ChattyPointer<IChatty> target;

    public Interviewer() {
        this.chater = null;
        this.target = new ChattyPointer<IChatty>();
    }

    public void setChater(IChatty chater) {
        this.chater = chater;
    }

    public void setTarget(IChatty chater) {
        this.target.define(chater);
    }

    public IChatty getChater() {
        return this.chater;
    }

    public ChattyPointer<IChatty> getTarget() {
        return this.target;
    }

    public void interview() {
        System.out.print("Hello, what do you think? ");
        System.out.println(this.chater.chat());
    }

    public void smartInterview() {
        System.out.print("Hello, what do you think? ");
        System.out.println(this.target.data().chat());
    }
}
```

Listing A.11 – Program

```java
package compo;

public class Program {
    public static void main(String[] args) {
        IChatty sadPerson = new SadPerson();
        IChatty expressivePerson = new ExpressivePerson();

        Interviewer interviewer = new Interviewer();
        Cameraman cameraman = new Cameraman();

        interviewer.setChater(sadPerson);
        cameraman.setChater(interviewer.getChater());

        interviewer.interview();
        cameraman.listen();

        interviewer.setChater(expressivePerson);

        interviewer.interview();
        cameraman.listen();

        System.out.println("Simulation of required");
        interviewer.setTarget(sadPerson);
        cameraman.setTarget(interviewer.getTarget());

        interviewer.smartInterview();
        cameraman.smartListen();

        interviewer.setTarget(expressivePerson);

        interviewer.smartInterview();
        cameraman.smartListen();
    }
}
```

Explication

L'application Java donnera à l'execution dans la console :

Listing A.12 – output

```
Hello, what do you think? You will fail your life.
The cameraman is listening... You will fail your life.
Hello, what do you think? I'm in love with you.
The cameraman is listening... You will fail your life.
Simulation of required
Hello, what do you think? You will fail your life.
The cameraman is listening... You will fail your life.
Hello, what do you think? I'm in love with you.
The cameraman is listening... I'm in love with you.
```

On utilise en Java un objet qui va servir à garder le lien sur la personne en interview.

Annexe B

Compteur de manifestants en COMPO

Nous avons implémenté en COMPO un exemple de compteur de manifestants. Celui-ci comporte un composant instance de `Compteur` qui implémente la logique métier et un composant instance de `Printer` qui affiche la valeur du compteur. Dans un premier temps on met un compteur de la police, on remplace ensuite ce compteur par celui des syndicats.

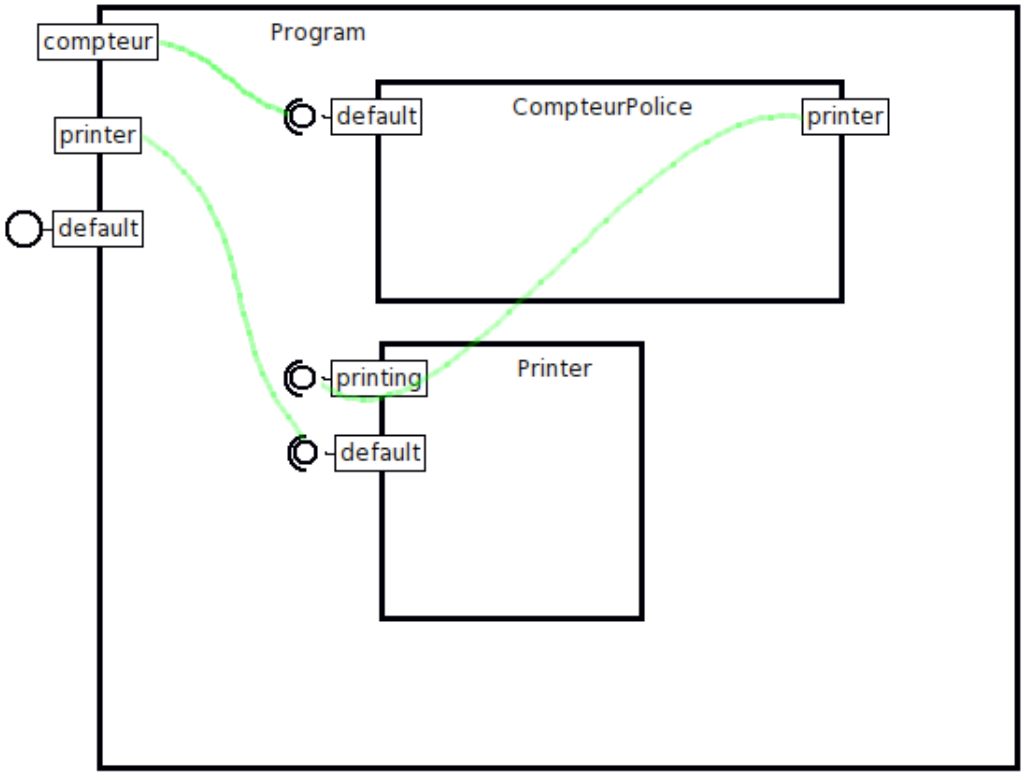


FIGURE B.1 – Schéma du programme extrait de `Compobrowser2`

Listing B.1 – worspace

```
|compteur|
compteur := Program.new();
compteur.print();
compteur.incrementer();
compteur.print();
```

Listing B.2 – output (Avec le `CompteurPolice`)

```
0
1
```

On remplace le compteur de la police par celui des syndicats. Pour cela, on met en commentaire la ligne 15 et on dé-commente la ligne 16 de B.4. On exécute à nouveau le programme.

Listing B.3 – output (Avec le `CompteurSyndicat`)

```
0
5
```

Listing B.4 – Programme utilisant un compteur

```
1  Descriptor Program extends Component
2  {
3     provides {
4        default: {
5           incrementer();
6           print();
7         };
8     }
9     internally requires {
10       compteur : Compteur;
11       printer: Printer;
12    }
13    architecture {
14       connect compteur@self to default@(CompteurPolice.new());
15      "connect compteur@self to default@(CompteurSyndicat.new());"
16       connect printer@self to default@(Printer.new());
17       connect printer@compteur to printing@printer;
18    }
19
20    service incrementer() {
21       compteur.incrementer();
22    }
23
24    service print() {
25       compteur.showResult();
26    }
27 }
```

Listing B.5 – Compteur de la Police

```
Descriptor CompteurPolice extends Component
{
   provides {
      default: {
          incrementer();
          incrementer(val);
          showResult();
      };
   }

   internally requires {
         compteurVal : SmallInteger;
   }

   requires {
      printer: { print(txt); };
   }

   architecture {
       connect compteurVal@self to 0;
   }

   service incrementer(val) {
       | t |
       t := compteurVal.asString().asInteger() + val;
       connect compteurVal@self to t;
   }

   service incrementer() {
       | t |
       t := compteurVal.asString().asInteger() + 1;
       connect compteurVal@self to t;
   }

   service showResult() {
      printer.print(compteurVal.asString());
   }
}
```

Annexe C

Exemple de suppression d'une connexion distante

Dans cet exemple, on prend 3 composants. `A` est relié à `B` qui est relié à `C`. Ces 3 composants sont inclus dans un composant programme nommé `ABC` dont la représentation est donnée en C.1. Le but de cet exemple est de montrer qu'un service de `A` peut couper la connexion entre `B` et `C`. Pour cela, on utilise une fonction `fooA` dans `A` qui va couper la connexion entre `B` et `C`. On met également une méthode `barA` dans `A` qui appelle une méthode `barB` dans `B` qui appelle une méthode `barC` dans `C` qui affiche *Hello* dans le `Transcript`.

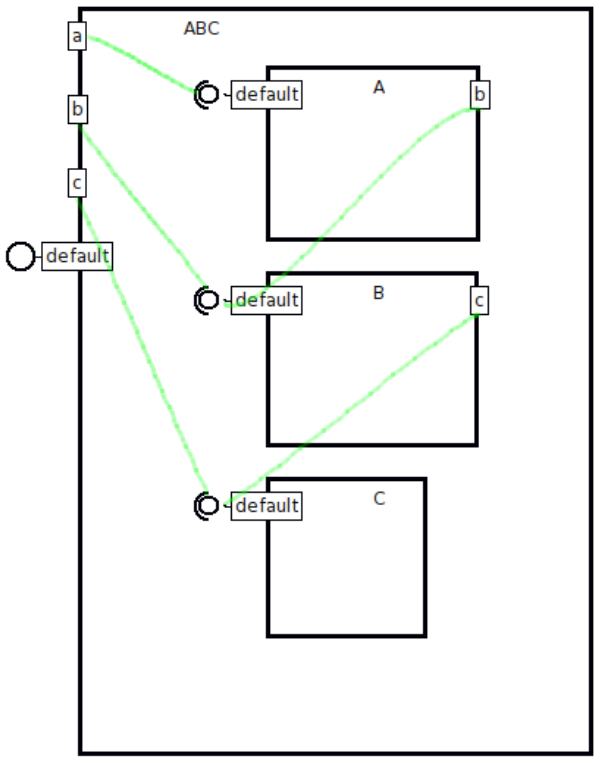


FIGURE C.1 – Schéma du programme extrait de `Compobrowser2`

Listing C.1 – Workspace

```
1 | prog |
2 prog := ABC.new();
3 prog.bar(); "Affiche Hello dans le Transcript"
4 prog.foo(); "Supprime la connexion entre B et C"
5 prog.bar(); "Provoque une erreur"
```

L'invocation du service `bar` ligne 5 lancera l'exception suivante car foo aura coupé la connexion.

Listing C.2 – Erreur lors de l'exécution du 2e bar()

```
1 Port is not connected
```

Listing C.3 – Exemple d'une déconnexion à distance

```
1  Descriptor ABC extends Component
2  {
3     provides {
4        default: { foo(); bar(); };
5     }
6
7     internally requires {
8        a : A;
9        b : B;
10       c : C;
11    }
12
13    architecture {
14       connect a@self to default@(A.new());
15       connect b@self to default@(B.new());
16       connect c@self to default@(C.new());
17
18       connect b@a to default@b;
19       connect c@b to default@c;
20
21    }
22
23    service foo (){
24       a.fooA();
25    }
26
27    service bar() {
28       a.barA();
29    }
30 }
```

Listing C.4 – Programme A

```
Descriptor A extends Component
{
   provides {
      default: {
         fooA(); barA();
       };
   }

   requires {
      b : B;
   }

   service fooA() {
      disconnect c@(b@self) from default@(c@(b@self));
   }

   service barA() {
      b.barB();
   }
}
```

Listing C.5 – Programme B

```
Descriptor B extends Component
{
   provides {
      default: { barB(); };
   }

   requires {
      c : C;
   }

   service barB() {
      c.barC();
   }
}
```

Listing C.6 – Programme C

```
Descriptor C extends Component
{
   provides {
      default: { barC(); };
   }

   service barC(){
      <! "Primitive SmallTalk"
          Transcript crShow: 'Hello'. >
   }
}
```

Annexe D

Comparaison entre les deux modes de retour en COMPO

Le but de cette exemple est de montrer la différence entre les deux modes de retours.

Pour cela, on utilise un compteur classique. Un second composant qui est `suiveur` possède un port `compteurVal`. Nous allons montrer que l'on peut synchroniser les deux compteurs pour que lorsque l'on incrémente le compteur, le port interne `compteurVal` du suiveur se met à jour automatiquement.

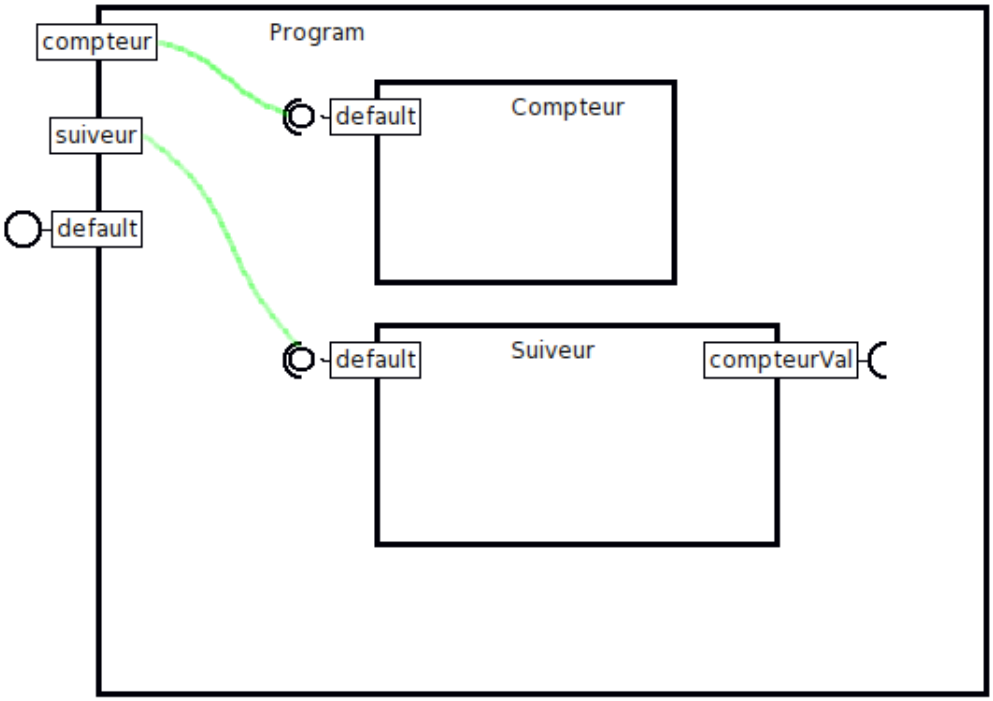


FIGURE D.1 – Schéma du programme extrait de `Compobrowser2`

Listing D.1 – Workspace (Retour classique)

```
|prog|
prog := Program.new();
prog.init1();

prog.printCompteurs();
prog.inrementer();
prog.printCompteurs();
```

Listing D.2 – Output (Retour classique)

```
compteur : 1
suiveur : 1
Increntation du compteur
compteur : 2
suiveur : 1
```

On remarque que lorsque l'on incrémente le compteur, le suiveur n'est pas modifié. En effet, le suiveur s'est connecté directement sur la valeur que possédait le compteur c'est à dire 1. Lorsque l'on incrémente le compteur est modifié. Cependant le suiveur reste connecté à la valeur que on lui avait donnée, autrement dit 1.

Listing D.3 – Output (Retour par requis)

```
|prog|
prog := Program.new();
prog.init2();

prog.printCompteurs();
prog.inrementer();
prog.printCompteurs();
```

Listing D.4 – Output (Retour par requis)

```
compteur : 1
suiveur : 1
Incrementation du compteur
compteur : 2
suiveur : 2
```

Dans cet exemple, le suiveur est bien modifié car on a connecté la valeur du suiveur directement au port qui contient la valeur du compteur et pas à la valeur comme précédemment.

Listing D.5 – Le programme

```
Descriptor Program extends Component
{
   provides {
      default: {init1(); init2(); incrementer();};
   }
   internally requires {
      compteur : {incrementer();getCompteurVal();getCompteurValReq();printValue();};
      suiveur: {getCompteurVal(); setCompteurVal(cpt);printValue();};
   }
   architecture {
       compteur >>> default@(Compteur.new());
       suiveur >>> default@(Suiveur.new());
   }
   operation init1() {
      suiveur.setCompteurVal(compteur.getCompteurVal());
   }
   operation init2() {
        suiveur.setCompteurVal(compteur.getCompteurValReq());
   }
   operation printCompteurs() {
        compteur.printValue();
        suiveur.printValue();
   }
   operation incrementer(){
        Transcript.crShow('Incrementation du compteur');
        compteur.incrementer();
   }
}
```

Listing D.6 – Programme Suiveur

```
Descriptor Suiveur extends Component
{
   provides {
      default: {getCompteurVal(); setCompteurVal(cptVal);};
   }

   requires {
      compteurVal: SmallInteger;
   }

   operation getCompteurVal() {
      return compteurVal;
   }
   operation setCompteurVal(cptVal) {
      compteurVal >>> cptVal;
   }
   operation printValue() {
       Transcript.crShow(compteurVal.asString());
   }
}
```

Listing D.7 – Programme Compteur

```
Descriptor Compteur extends Component
{
   provides {
      default: {incrementer(); getCompteurVal(); getCompteurValReq(); printValue();};
   }

   internally requires {
      compteurVal: SmallInteger;
   }

   architecture {
       compteurVal >>> 1;
   }

   operation getCompteurVal() {
       "retourne la valeur connectee"
      return compteurVal;
   }

   operation getCompteurValReq() {
       "retourne le port directement"
      return !compteurVal;
   }

   operation incrementer() {
       | t |
       t := compteurVal.asString().asInteger() + 1;
       compteurVal >>> t;
   }

   operation printValue() {
       Transcript.crShow(compteurVal.asString());
   }
}
```

Annexe E

Comparaison entre les deux modes de passages en COMPO

Prenons un composant `A` et un composant `B`. Tous deux possédent un port `value`. Le composant `A` va passer au composant `B` son port `value`.

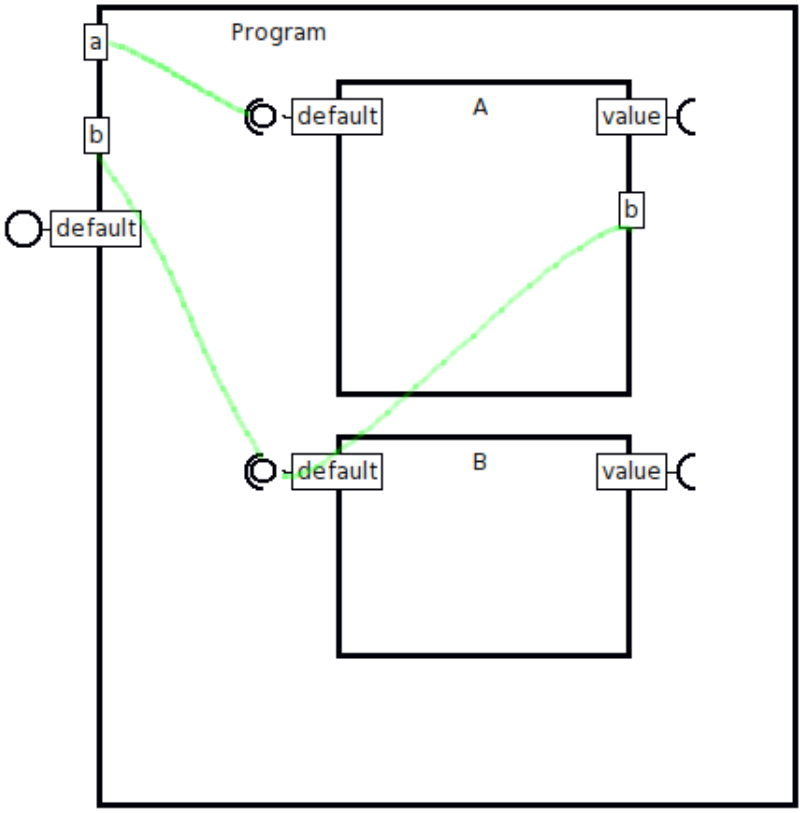


FIGURE E.1 – Schéma du programme extrait de `Compobrowser2`

Listing E.1 – Workspace (Passage par Fournis)

```
|prog|
prog := Program.new();
prog.initProv();

prog.printValue();
prog.changeValA();
prog.printValue();
```

Listing E.2 – Output

```
A : 1
B : 1

A : 2
B : 1
```

On remarquera que B n'a pas été modifié quand on a modifié ce qui était connecté au port `value` du composant `A` avec le mode de passage par défaut.

Listing E.3 – Workspace (Passage par Requis)

```
|prog|
prog := Program.new();
prog.initReq();

prog.printValue();
prog.changeValA();
prog.printValue();
```

Listing E.4 – Output

```
A : 1
B : 1

A : 2
B : 2
```

Avec le mode de passage par requis, cette fois le composant `B` a bien été modifié en même temps que `A`

Listing E.5 – Composant A

```
Descriptor A extends Component
{
   provides {
      default : { initProv(); initReq(); print(); setValue(val); };
   }

   requires {
      value : SmallInteger;
      b : B;
   }

   operation setValue(val){
       value >>> val;
   }

   operation initProv() {
      b.setValue(value);
   }
   operation initReq() {
      b.setValue(!value);
   }

   operation print() {
      Transcript.crShow(value);
   }
}
```

Listing E.6 – Composant B

```
Descriptor B extends Component
{
   provides {
      default : { setValue(v); print();};
   }

   requires {
      value: SmallInteger;
   }

   operation setValue(v) {
      value >>> v;
   }

   operation print() {
      Transcript.crShow('' value.asString());
   }
}
```

Listing E.7 – Programme

```
Descriptor Program extends Component
{
   provides {
      default : {printValues(); changeValA(); initProv(); initReq(); };
   }

   internally requires {
      a: A;
      b: B;
   }

   architecture {
      a >>> default@(A.new());
      b >>> default@(B.new());
      b@a >>> default@b;
      value@a >>> 1;
   }

   operation printValues(){
      a.print();
      b.print();
   }

   operation changeValA(){
       a.setVal(2);
   }

   operation initProv(){
       a.initProv();
   }

   operation initReq(){
       a.initReq();
   }

}
```

Annexe F

Création d'une règle pour le *parser*

COMPO utilise le *framework* `PetitParser` pour l'analyse syntaxique. Il permet la création d'un *AST* grâce à des règles de constructions. Nous allons étudier comment COMPO utilise `PetitParser`.

Les différents *packages*

Compo parser core

Ce *package* contient principalement deux classes à savoir :
— `PPCompoGrammar` : contient les règles syntaxique
— `PPCompoParser` : création des "nodes"

Compo parser node

Ce package contient une série de nodes, les nodes ont leurs propres attributs mais surtout implémentent une méthode nommée `visitNode`. Par la suite, un autre élément va parcourir ces nodes grâce à l'implémentation du design pattern "visiteur" afin de générer du code.

Compo parser visitor

Ce package est l'élément de l'implémentation qui va s'occuper de parcourir les nodes, il contient une classe très importante nommé CompoToSmalltalk, cette classe a pour but de généré du code Smalltalk.

Fonctionnement du parseur

Le point d'entrée du *parseur* est à la méthode `start` de `PPCompoGrammar`. Elle est appelée lorsque l'on fait *accept* dans `CompoBrowser`.

Création d'une règle

Il faut d'abord créer un attribut de classe du nom de la règle qu'on va créer, ensuite il suffit de faire une méthode du même nom, et renvoyer un token que PetitParser va comprendre. Ce token ressemble à une expression régulière, voici quelques exemple de règles extrait de [**?**]

— Création d'un literal
  — (<nom du literal> asParser)
  — Ex : ('connect' asParser) ;
— Pour la détection de nombre, chaine de caractere, ... des symboles ont été crée : *#letter*, *#word*, ... ;
— Détection d'un *token* [0..n]
  — <token> star
  — Ex : (('connect' asParser) star) ;
— Détection d'un *token* [1..n]
  — <token> plus ;
  — Ex : (('connect' asParser) plus) ;
— Detection de la suite token1 puis token2 : <token1>, <token2> ;
— Detection token2 si token1 echoue : (<token1> / <token2>) ;

Ensuite, il suffit de rajouter une méthode du même nom dans PPCompoParser et de la définir ainsi :

Listing F.1 – Création d'une régle

```
1 <ma_nouvelle_regle>
2   ^ super <ma_nouvelle_regle>
```

PPCompoParser herite de PPCompoGrammar, ainsi la règle est implémentée et prête à l'utilisation.

Création d'un node

Là, il n'y a pas grande chose à faire, il suffit de faire une sous classe de `PPCompoNode`, et d'ajouter à cette classe la méthode `visitNode`, qui va appeler une méthode qu'on définit dans `CompoToSmalltalk` avec comme argument, lui même.

Génération de code

Maintenant, on écrit la méthode dans `CompoToSmalltalk`. Cette méthode prend en argument le node. Afin de générer du code, on utilise l'objet CodeStream, qui a une méthode NextPutAll qui prend le code en argument et qui génère le code.

Problème des littéraux (rock bottom)

COMPO est fait de façon à ce que lors que l'on écrit

```
1 connect <lport>@<lcomp> to <rport>@<rcomp>
```

`lcomp` et `rcomp` sont des chaines de caractères (du moins dans la clause architecture). La déduction des chaines de caractère est faite lors de la création des connexions dans `ComponentBuilder`. Ainsi, pour un littéral, en plus d'ajouter la règle syntaxique, il faut corriger le comportement de ce `ComponentBuilder` afin qu'il detecte les litéraux.