Formalisation de la sémantique d'un langage à composants

Étude bibliographique

Master *Sciences et Technologies*,
Mention *Informatique*,
Parcours AIGLE

Auteur
Jimmy Lopez

Superviseurs
David Delahaye
Christophe Dony
Chouki Tibermacine

Lieu de stage
LIRMM UM5506 - CNRS, Université de Montpellier

version 1 30 mars 2016

Résumé

Le développement par composants est un paradigme important du Génie Logiciel. Il est une solution pour répondre aux problématiques de modularité et de réutilisabilité dans la production de logiciels. Dans la littérature, il existe un certain nombre d'architectures logicielles à base de composants, qui se regroupent en trois grandes familles : les frameworks, les approches génératives et les langages à composants. Cette étude porte sur la spécification la sémantique formelle et l'implantation d'un système de passage de composants en arguments dans le langage "pur Composants" Compo. De plus, ne pas avoir d'ambiguïtés dans la spécification d'un langage est un problème récurrent pour tout concepteur de langage. Dans cette étude bibliographique, nous présentons un état de l'art des différentes familles à composants, les différents modes de passage de paramètres dans les langages, ainsi que l'ensemble des techniques permettant la définition formelle de la sémantique d'un langage. Dans le contexte de la famille des langages à composants, on se place dans le cadre particulier du langage Compo, développé au sein de l'équipe MAREL du LIRMM, afin de fournir une sémantique formelle et de compléter sa conception.

Table des matières

1

Introduction

Le Génie Logiciel (GL) est une discipline de l'informatique visant à proposer des modèles, logiciels, méthodes et outils pour la production de logiciels, reposant sur un développement standardisé, automatisé et organisé dans le but de réduire les coûts de production, permettant aux créateurs de logiciels de maîtriser leurs produits sur toutes leurs facettes : fonctionnalités, fiabilité, qualité, maintenabilité et réutilisé [Don89]. On trouvera une définition du GL dans [Boe76] ; une autre définition de S.Krakowiak :

> *... ensemble des méthodes, techniques, et outils nécessaires à la production de logiciel de qualité industrielle, pour l'ensemble du cycle de vie d'un produit... Il s'agit en fait de passer à terme d'une technique basée sur la réparation, ... à une technique fondée sur la conception sûre et la garantie à priori de la qualité.*

mettant en avant les problématiques de modularité (décomposer un programme en modules – fonctions et/ou méthodes et/ou sous-programmes – afin de réaliser un développement indépendant et de répartir la complexité du programme) et de réutilisabilité (réutiliser tout ou partie d'un logiciel – spécification et/ou conception et/ou programme – pour en faire un autre). Le développement par objets est une partie importante du GL, permettant de répondre à ces problématiques, mais n'est pas le seul paradigme possible. En effet le développement par composants permet lui aussi de répondre à ces problématiques.

Dans ce contexte global nous nous intéressons plus spécifiquement aux architectures logicielles à base de composants. Le concept de composant a été introduit historiquement par [McI68], donnant depuis de multiples recherches, pouvant être regroupées en trois grandes familles : les frameworks, les approches génératives et les langages à composants (LOC), s'unifiant autour d'une idée centrale, qui voit le composant comme une entité boîte noire, communiquant avec le monde extérieur au moyen d'interfaces (requises ou fournies).

Dans ce stage, nous allons nous intéresser à la dernière famille. L'approche des LOC propose la création et la manipulation de composants au sein d'un seul et même langage de programmation. Les langages de cette famille se divisent en deux catégories. Une première dite «mixte», évoluant dans un monde où la notion d'objet et de composant cohabitent, c'est-à-dire où les composants s'échangent des données au travers d'objets.

La deuxième dite «pure» où seule la notion de composant existe (cf. annexe A). Les composants s'échangeant des données par connexion à d'autres composants.

Afin d'enlever toute ambiguïté dans la spécification d'un langage de programmation définie dans le langage naturel, on peut définir formellement la sémantique de ce langage. Cela permet de caractériser mathématiquement les calculs décrits par un programme et les résultats qu'il produit. Les programmes deviennent des objets mathématiques qui doivent s'évaluer selon un certain nombre de règles définies formellement par la sémantique du langage.

La problématique du stage se place dans ce contexte spécifique des langages pur composants. Nous allons nous placer dans le cadre particulier du langage Compo, développé au sein de l'équipe MAREL du LIRMM. L'objectif du stage est de définir la sémantique formelle de ce langage. Cependant, avant de mettre en place cette sémantique, nous allons compléter la conception de Compo en étudiant les différents modes de passage de paramètres dans un monde «tout composant».

Dans le chapitre suivant, nous aborderons un état de l'art autour des trois grandes familles à composants, les différents modes de passage de paramètres dans les langages et la formalisation de la sémantique d'un langage. Le chapitre 3 présentera les différentes problématiques liées au sujet ainsi que les pistes de recherche envisagées dans le cadre de l'évolution de la conception de Compo.

2

État de l'art

La première section introduit le concept des langages de programmation par composants. La deuxième section nous replace dans le contexte des langages de programmation et de leurs différentes techniques permettant le passage d'arguments. Enfin la dernière section présente les techniques de définition de la sémantique des langages de programmation, avec la mise en place d'un mécanisme de preuve.

2.1 Le développement de logiciels à base de composants

2.1.1 Présentation de l'approche à composants

Le terme « composant », défini dans l'approche de l'ingénierie du logiciel basée sur les composants (*Component-Based Software Engineering – CBSE*) étant très générique, en donner une définition exacte et précise paraît difficile car cela dépend fortement du contexte de son utilisation. Cependant on peut se baser sur des définitions faites dans la littérature :

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.* [Szy02]

> *A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces.* [Kru98]

> *A component is a unit of distributed program structure that encapsulates its implementation behind a strict interface comprised of services provided by the component to other components in the system and services required by the component and implemented elsewhere. The explicit declaration of a component's requirements increases reuse by decoupling components from their operating environment.* [PC98]

Ces différentes définitions permettent de faire ressortir des notions qui se retrouvent dans la plupart des approches à composants :

interfaces C. Szyperski définit une interface d'un composant comme étant un point d'accès au service du composant [Szy99] permettant de décrire comment les composants peuvent être assemblés ou utilisés dans une architecture. On parle d'interfaces « requises », les interfaces permettant de décrire les besoins d'un composant et d'interfaces « fournies », les interfaces permettant de définir les fonctionnalités que proposent le composant aux autres composants.

indépendance Il faut voir un composant comme un élément indépendant de tout système. Il doit être assez générique pour pouvoir se connecter à d'autres composants au sein d'une nouvelle application en fournissant un ensemble de services, sans être trop spécifique à un système.

architecture La notion d'architecture permet de représenter le plan de l'application et permet de décrire comment l'application doit être conçue afin de respecter les spécifications mises en place.

composition Le mécanisme de composition permet de créer à partir d'un assemblage de composants, un nouveau composant plus complexe, en encapsulant des composants afin de pouvoir réutiliser directement cet assemblage. Ce nouveau composant est dit composite car constitué de composants qui deviennent des sous-composants (composants interne), pouvant être eux aussi des composants composites ou des composants primitifs [1].

service Les services permettent de représenter la logique métier d'un composant. Généralement présents uniquement dans des composants primitifs, certaines approches essaient de définir des services pour des composants composites.

2.1.2 Les principes du développement par composants

De nombreux concepts permettent la réutilisabilité dans le développement (appel de fonction, importation de module, héritage entre classes, paramétrage de framework, assemblage de composants, etc.). Cependant, même si ces différentes techniques permettant de mettre en avant cette notion de réutilisabilité, les composants ont fait de la réutilisation le fer de lance de leur approche.

On identifie deux acteurs dans le développement par composants [Fab07] : le développeur de composants et l'architecte d'application (cf figure 2.1). Le rôle du développeur va consister à réaliser des composants indépendants. L'architecte récupère les composants déjà créés par un développeur et réalise une intégration dans son assemblage de composants existant. Il est bien sûr possible pour une personne d'avoir les deux rôles à la fois, mais il est important dans ce paradigme de rendre son composant indépendant de l'application.

Nous avons vu que l'architecture logicielle permettait de définir comment notre application devait être construite. C'est pour cela que nous avons besoin de visibilité sur cette

[1] un composant est dit primitif s'il ne contient pas de sous-composants

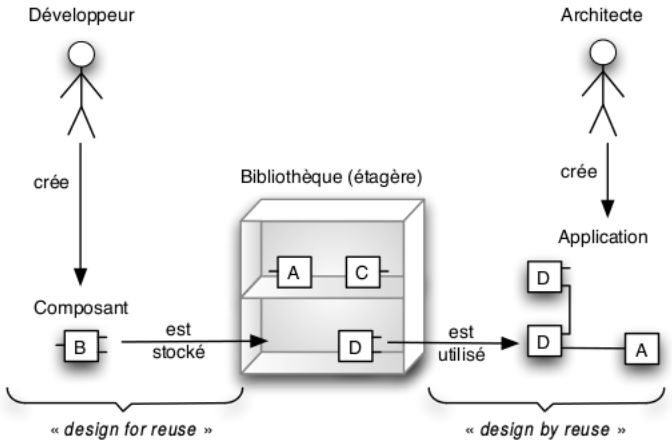

FIGURE 2.1: Vision simplifiée du processus de développement par composants, extrait de [Fab07]

architecture afin de bien appréhender comment notre application est réalisée et de visualiser toutes les intéractions entre les éléments de l'application, permettant ainsi de vérifier que cette architecture respecte bien les spécifications. Dans le développement par objets, la notion d'architecture est utilisée lors de la phase de conception de l'application. Le développeur peut se faire aider par des outils de représentation graphique, comme UML, mais cette représentation n'est que peu maintenue lors de la phase d'implémentation. Les modifications apportées par celle-ci ne sont plus visibles et donc n'assure plus que l'architecture obtenue soit conforme aux spécifications.

Le développement par composants arrive à corriger cela, en rendant explicite cette visibilité sur l'architecture, soit à la manière des ADLs (Architecture Description Languages), qui permettent de décrire la structure et de définir le comportement des composants à travers leurs assemblages, par une représentation abstraite, soit directement dans l'implémentation des composants par l'approche des langages à composants, qui définissent l'architecture au sein même des composants. Par conséquent, même si pendant la phase d'implémentation on décide de modifier l'architecture de nos composants, on ne perdra pas la visibilité sur l'architecture de notre application car elle est écrite explicitement.

2.1.3 Les grandes familles du développement à base de composants

2.1.3.1 Les approches des génératives

Les approches des génératives se situent à un haut niveau d'abstraction afin de modéliser et gérer des systèmes logiciels complexes. Cette représentation se base généralement sur les ADLs, permettant de décrire les architectures à base de composants en y décrivant leurs comportements, leurs intéractions avec les autres composants et leurs configurations ainsi que leurs besoins explicites. La stratégie de cette approche générative consiste donc

en la modélisation d'une architecture de composants « initiale » décrite de manière formelle. Ensuite un « squelette » d'application dans un langage de programmation précis qui respectera les spécifications du système est généré à partir de l'architecture. Cette représentation étant abstraite, elle est indépendante des langages de programmation qui implémentent les composants, permettant ainsi de s'adapter à tous types de plateformes.

Parmi les modèles les plus importants dans cette famille, nous retrouvons UML et Fractal.

UML – un langage de modélisation graphique pour composants

UML (Unified Modeling Language) est un langage de modélisation graphique standardisé par l'OMG (*Objet management groupe*) [2]. Dans sa première version, UML intègre déjà la notion de composant, comme étant une entité indépendante et communiquant avec d'autres composants au travers d'interfaces. La version 2.0 d'UML [Spea] a permis d'améliorer cette vision des composants, en s'inspirant aussi des mécanismes et concepts des ADLs, introduisant la notion de port, de connecteur et de composant composite.

Un composant UML est donc une entité autonome, qui communique avec les autres composants de l'architecture au travers d'interfaces, regroupées dans des ports qui expriment des rôles. Un composant peut disposer de plusieurs ports, où un port regroupe un ensemble d'interfaces requises ou fournies. Le port permet donc de réaliser un point de communication entre l'environnement extérieur du composant et son architecture interne.

Cette connexion entre composants est permise par la notion de connecteur, qui permet de représenter les possibilités de communication entre les composants. Il existe deux types de connecteurs : d'assemblage et de délégation. Un connecteur d'assemblage permet de lier deux interfaces de deux composants. Avec une interface requise pour notre premier composant et une interface fournie pour notre deuxième composant. Cette connexion ne peut se faire que si les deux interfaces ont la même signature. Un connecteur de délégation quant à lui est utilisé afin de réaliser une connexion entre ses interfaces ou ports vers les interfaces ou ports de ses composants internes, permettant ainsi une délégation d'un besoin ou d'un service vers un composant interne.

UML apporte cet aspect visuel à la représentation des architectures à composants qui manquent aux autres approches, aidant ainsi un architecte à mieux visualiser l'architecture qu'il est en train de construire. Par contre, il ne permet pas de réaliser l'implémentation et la manipulation de composants sans les coupler à une autre technique.

Fractal – un framework à composants intégrant un ADL

Fractal [BCL+06] est une spécification d'un modèle à composants proposé par le consortium *ObjectWeb* en 2002. Il propose un framework pour la création d'applications déployables sur un serveur, ainsi que la création et la manipulation de composants, intégrant en plus un ADL pour la description de l'architecture des composants. Il met en avant le concept de composant composite, de composant partagé, un mécanisme d'introspection et la possibilité de configuration dynamique.

[2] http ://www.omg.org/spec/UML

Un composant Fractal est composé de deux parties. La partie *contenu* qui est constituée des fonctionnalités que propose le composant, en étant soit un composant primitif qui dans son *contenu* va implémenter la logique métier, soit un composant composite qui encapsule un assemblage de composants qui réalise les fonctionnalités attendues. La deuxième partie d'un composant est appelé la *membrane*, qui contient les parties non-fonctionnelles du composant, intégrant les différentes interfaces qui communiquent avec le monde extérieur, ainsi que les interfaces qui permettent le contrôle du composant. Parmi les modèles les plus importants dans cette famille, nous retrouvons UML et Fractal.

Une interface a deux types de visibilité : externe qui est visible depuis l'extérieur du composant et permettra une liaison avec d'autre composants et interne permettant une liaison avec son contenu. Une interface peut avoir différents rôles :

1. Le rôle de serveur, qui est une interface fournie proposant les différents services du composant
2. Le rôle de cliente, qui est une interface requise représentant les besoins du composant
3. Le rôle de contrôle, permettant la configuration dynamique du composants avec la possibilité de modifier ces liaisons avec les composants externe, de modifier son assemblage interne (modification des liaisons internes) et la gestion du cycle de vie du composant

Afin de pouvoir connecter des composants entre eux, il faut connecter une interface de l'un avec une interface de l'autre. Cette connexion est appelée liaison (*binding*) et peut être de trois types : une liaison normale (*normal binding*) entre une interface serveur externe d'un composant et une interface cliente externe d'un autre composant, une liaison d'export (*export binding*) entre une interface cliente d'un composite et une interface serveur d'un composant interne, permettant donc la délégation d'un service fourni par notre composant composite qui sera fourni par son composant interne, et une liaison d'import (*import binding*) entre une interface serveur interne d'un composant composite et une interface cliente externe d'un composant interne permettant la délégation d'un besoin de notre composant interne au composant composite.

Étant un modèle abstrait, il permet de s'abstraire de tous langages et de plateformes pour la gestion de composants. Différentes implémentations ont été réalisées (par exemple en Java avec Julia ou en C++ avec Cecilia). En effet l'ADL qu'intègre Fractal est décrit dans une syntaxe XML et spécifié de manière indépendante des langages de programmation. Les différentes implémentations proposent cependant un mécanisme de génération de code à partir de l'ADL pour réaliser l'implémentation des composants.

2.1.3.2 La famille des frameworks

La famille des frameworks à composants utilise les langages de programmation à objets connus, pour la création et la manipulation de composants afin de modéliser un

système. Cette approche permet au développeur d'utiliser les composants afin de bien séparer les besoins fonctionnels et les besoins non-fonctionnels. Principalement utilisé dans le monde professionnel, les frameworks à composants proposent aux développeurs de créer, déployer et configurer une application.

Parmi les frameworks les plus importants dans cette famille, nous retrouvons Enterprise JavaBeans et Spring.

Enterprise JavaBeans – un framework à composants pour la création d'applications distribuées

L'approche de composant mise en place par Enterprise JavaBeans (EJB), développée au sein de la plateforme Java Enterprise Edition (JEE), propose un modèle à composants serveur permettant la réalisation d'applications distribuées [Speb].

Un composant EJB s'exécute dans un conteneur au sein d'un serveur JEE. Les conteneurs gèrent les instances des EJB, tandis que le serveur fournit au conteneur des besoins non-fonctionnels.

Un composants EJB est regroupé en trois catégories : session, entité et message. Une instance d'un composant entité permet de représenter un concept métier, ayant pour but de représenter des données pouvant être stockées dans une base de données. Une instance d'un composant message permet la gestion de messages asynchrones reçus par le serveur depuis le client. Une instance d'un composant session permet de fournir des services définis par des méthodes, pouvant être sans état (stateless) ou avec état (stateful).

Un composant EJB communique ses services par une interface *remote*, spécifiant ainsi les méthodes que fournit une instance d'un EJB au monde extérieur. Il permet sa configuration et la gestion de son cycle de vie par une interface *home* qui est fournie par son conteneur. Cependant on ne peut définir le besoin d'un EJB au moyen d'une interface requise. Les propriétés d'un EJB sont définies dans un descripteur de déploiement permettant au conteneur EJB de savoir comment gérer les composants pendant leur exécution.

L'approche des EJB ne permet pas la réalisation de composant composite. Ainsi une architecture sera dite « à plat », chaque composant sera au même niveau que les autres, permettant une vision horizontale de l'architecture. Mais cela peut s'avérer répétitif de réaliser l'ensemble de toutes les différentes connexions entre composants sans pouvoir utiliser la composition pour réutiliser un assemblage de composants.

Spring – un framework à composants pour la création d'applications modulaires

Spring [3] est un framework utilisé afin de fournir une infrastructure compréhensive pour la création d'applications Java. Comme la grande majorité des frameworks, il fonctionne sur le principe de l'inversion de contrôle (IoC – patron de conception où le processus d'exécution d'une application est géré par le framework), qui est assurée par la recherche de dépendances et l'injection de dépendances automatique, tout en offrant un ensemble de fonctionnalités facilitant l'accès aux données, le développement web, les tests automatisés, etc., qui sont organisés en modules.

[3] https ://spring.io/

Un module est une unité de code indépendante d'un système logiciel. Il possède deux caractéristiques fondamentales : une forte cohésion (le module se focalise sur une tâche unique), et un couplage minimal (le module a de faibles dépendances avec les autres modules). Un module est décrit par une spécification, qui définit les signatures des fonctions publiques fournies par le module et une implémentation, qui implémente les fonctions fournies. Il permet de répondre aux problématiques de maintenabilité, réutilisabilité et testabilité.

Le cœur de Spring est géré par le *Spring Core Container*, qui est un ensemble de modules permettant la réalisation du conteneur Spring de l'application, qui a un comportement similaire au conteneur EJB. Il implémente l'IoC et permet de gérer la création, le cycle de vie et les dépendances des *bean* qu'il contient (un composant Spring (bean) étant une classe Java qui implémente une ou plusieurs interfaces fournies). La description des beans est principalement décrite dans des fichiers XML mais peut se faire dans le code source avec l'utilisation d'annotations Java ou l'utilisation de code source de manière spécifique par le framework.

Cette description décrit la classe du bean, ainsi que son identifiant. On peut injecter à cette description des constructeurs, avec comme argument des valeurs primitives ou des références vers d'autre beans (grâce à par leur identifiant). On peut aussi rajouter un ensemble de propriétés (i.e. attribut privé ayant un getter et un setter), définis par un nom et une valeur d'un type primitif ou une référence vers un autre bean. La reconfiguration d'un bean (par exemple, modifier la référence vers un bean d'une propriété), se réalise par la modification textuelle de la référence et la relance de l'exécution de l'application. Un bean composite est obtenu par la déclaration d'un bean interne (inner bean), qui se comporte comme une classe interne.

2.1.3.3 La famille des langages à composants

L'approche des langages à composants s'articule autour d'un seul langage de programmation pour permettre de définir une architecture de composants et l'implémentation de ses composants. Ce regroupement de ses deux principes permet de faciliter le développement, car le programmeur ne doit connaître qu'un seul langage. Cette approche autour d'un seul langage permet de ne pas avoir de violation de contraintes architecturales, qui pouvait se produire avec les ADLs, qui ont une description dans un langage et l'implémentation dans un autre.

Parmi les langages à composants les plus importants dans cette famille, nous retrouvons ArchJava et ComponentJ.

ArchJava – un langage à composants

ArchJava [4] créé en 2001 [ACN02], s'inspire de l'approche composant que proposent les ADLs pour la description d'architecture, tout en permettant l'implémentation de ces composants au sein d'un nouveau langage qui est une extension du langage Java.

Un composant est dans ce monde un «objet particulier», qui s'obtient par instanciation de la classe `component class`. Cette classe de composant permet de regrouper

[4] http ://www.archjava.org/

les informations du composant : ses ports, ses méthodes, ses attributs, ainsi que ma description de son assemblage interne à travers des connexions.

Deux composants ne peuvent communiquer entre eux que s'ils sont explicitement connectés par la primitive `connect`. Cette primitive introduit la connexion entre composants par la liaison de leurs ports respectifs pour la première fois dans l'histoire des approches à composants

Un port définit une liste de méthodes fournies et requises, différenciées respectivement par les primitives `provides` et `requires`. Une méthode fournie permet d'introduire une fonctionnalité proposée par le composant au monde extérieur. Les autres composants n'ont accès à cette méthode que par le biais du port. Une méthode requise permet de décrire une fonctionnalité attendue par notre composant. Deux ports ne peuvent se lier entre eux que si les signatures des méthodes requises et fournies qu'ils définissent sont compatibles entre eux.

ComponentJ – un langage à composants

ComponentJ [5] présenté par [SSP08], propose un modèle à composants, explicitant les dépendances entre composants, la création dynamique de nouveaux composants, la reconfiguration au runtime et un typage fort pour les composants. Il ne faut pas le confondre avec son cousin ArchJava, qui comme lui est une extension du langage Java, car ils sont tous les deux conceptuellement différents.

Par exemple, un composant est un élément instance d'une classe dans ArchJava, alors qu'un composant est un descripteur instanciable d'objets en ComponentJ. Ce descripteur permet de définir la structure et le comportement de chacune de leurs instances en décrivant les ports et en définissant les services fournis par les objets. En effet, un objet en ComponentJ est une instance de composant, qui a un comportement comparable à un objet Java. Cependant il doit être typé par une interface (**object interface**) qui définit l'ensemble des services que fournira notre objet, à travers la définition d'un seul port fourni.

Les ports sont des interfaces (**port interface**) qui définissent un ensembles de signatures de méthodes. Ces méthodes représentent les services des composants et seront dites requises ou fournies en fonction du rôle du port qui les définit. En effet si un port a un rôle de fourni dans un composant alors ce composant devra implémenter ces méthodes, identifiées par la primitive `methods`. Tandis que si un port a un rôle de requis, alors le composants pourra invoquer ces méthodes afin de les utiliser.

Les configurateurs sont des blocs d'instructions présents dans les composants. Ils permettent le mécanisme de composition ou de connexion. Ils peuvent être utilisés pour la création au runtime de nouveaux composants, ainsi que la reconfiguration des comportements et la restructuration des composants.

ComponentJ a fait le choix de mettre en avant l'assemblage de composants, qui ne sont pas ici des instances mais des descripteurs, leur instanciation étant possible uniquement si les dépendances requises sont satisfaites ce qui permet d'avoir une forte sûreté à l'exécution, mais rend plus contraignante l'utilisation des composants. En effet, un développeur doit satisfaire l'ensemble des besoins de chaque composant, même ceux qui

[5] https ://projectos.fct.unl.pt/projects/di-componentj/wiki

seraient inutiles dans l'application.

2.2 Les différents mécanismes du passage d'arguments en programmation

Dans la majorité des langages de programmation, l'appel à un sous-programme (procédure, fonction, méthode, etc.) peut prendre des paramètres (arguments). On ne s'intéresse pas ici au détail du fonctionnement de cet appel, mais on se concentre sur comment un paramètre effectif est transmis au sous-programme et comment son utilisation dans le corps du sous-programme peut dans certains cas impacter l'environnement. Il existe un grand nombre de mécanismes de passage d'argument mais nous allons voir les plus connus et les plus utilisés, présentés dans [Hor84].

2.2.1 Passage par valeur

Sûrement le mécanisme de passage d'arguments le plus connu et utilisé est le passage d'argument « par valeur ». Les modifications sont apportées aux paramètres formels et n'ont aucun impact sur l'environnement extérieur. Elle ne modifient en aucun cas l'état du paramètre effectif. Dans ce mode de passage, le paramètre effectif est déréférencé donnant une *r-value* [6] qui est copiée dans une nouvelle zone mémoire, liée à notre paramètre formel. Cependant cette copie consomme de l'espace mémoire et du temps de calcul pour la copie. Un grand nombre de langage utilise ce mécanisme, comme mécanisme par défaut, notamment C, C++ [Str00], ou encore Java.

2.2.2 Passage par référence

Dans le mécanisme de passage par référence, le paramètre formel fait référence à l'adresse mémoire de notre paramètre effectif. Dans ce mécanisme aucune copie n'est réalisée, ce qui permet de ne pas avoir les contraintes du passage par valeur, car aucune nouvelle zone mémoire n'a besoin d'être allouée (sauf celle du paramètre formel, bien sûr). Toutes les modifications auront un impact sur l'environnement local à la fonction mais aussi sur l'environnement extérieur à celle-ci.

Java [AGH00] utilise le passage par valeur pour les objets. En effet, même si certains affirmeront que Java fait un passage par référence, c'est en fait un abus de langage, c'est-à-dire que techniquement c'est un passage par valeur qui entraîne une copie de la référence de notre paramètre effectif. On se retrouve alors avec deux références qui pointent vers la zone mémoire de notre objet, ce qui a pour conséquence que si on réalise un appel qui modifie un attribut d'un objet passé en paramètre cela entraînera une modification sur le paramètre effectif. Mais si on réalise une affection sur cet objet alors le paramètre formel référencera une nouvelle zone mémoire, et les nouvelles modifications de l'état de notre objet ne modifieront plus l'état de notre paramètre effectif, car feront référence à la nouvelle zone mémoire créée.

[6] contenant une variable ou le résultat d'une expression

2.2.3 Passage par valeur-résultat

E. Horowitz définit le passage par « value-result » [Hor84] comme une méthode incluant les fonctionnalités du passage par valeur et du passage par référence. Dans ce type de passage, dans le corps du sous-programme le paramètre formel a le même mécanisme que le passage par valeur, avec déréférencement du paramètre effectif et la copie dans une nouvelle zone mémoire. Durant toute l'exécution du sous-programme les modifications n'auront un impact que sur l'environnement local de la fonction, mais une fois celle-ci terminé, la valeur du paramètre formel est copiée vers celle du paramètre effectif. Ce mécanisme est utilisé par le langage ALGOL-W [Sit72] (une variante d'Algol60).

2.2.4 Passage par nom

Dans ce mécanisme de passage de paramètres, c'est une substitution textuelle qui est effectuée par le paramètre d'effectif dans le corps de la fonction à chaque fois que le paramètre formel apparaît. Ce comportement est introduit comme comportement par défaut en ALGOL-60. Il réalise un effet semblable au passage par référence, dans le sens où aucune copie n'est réalisée et son champ d'action impacte aussi l'environnement extérieur. Il est aussi comparable aux macros en C.

2.2.5 ADA et ses trois nouveaux modes de passage

Le passage de paramètres en ADA est un peu particulier. En effet, comme mentionné par J-L. Nebut [Neb], le programmeur ne peut pas choisir entre un passage par valeur ou référence (les deux mécanismes présents dans ADA), mais a la possibilité de choisir un « mode » de passage. Le mode permet au développeur de définir chaque argument comme étant en entrée (IN), sortie (OUT) ou entrée/sortie (IN OUT).

En mode `IN`, le paramètre formel est vraiment en « read-only », c'est-à-dire que dans le corps de la fonction on ne peut pas modifier la valeur du paramètre formel. C'est le comportement par défaut utilisé par ADA. En mode `OUT`, le paramètre formel peut être modifié dans la procédure mais ne peut être lu. Si le paramètre effectif est affecté dans la procédure, sa valeur pourra être utilisée dans la suite du programme avec cette nouvelle valeur affectée. En mode `IN OUT`, les deux mécanismes précédents sont combinés, avec comme contrainte que le paramètre effectif doit être initialisé avant l'appel.

2.3 La sémantique formelle des langages de programmation

2.3.1 Présentation

La sémantique d'un langage de programmation permet de définir le comportement d'un programme écrit dans ce langage, à la différence de la syntaxe d'un langage de programmation qui permet de décrire la représentation d'un programme dans ce langage. Ces deux notions sont complémentaires mais la frontière entre la sémantique et la syntaxe n'a pas toujours été claire (notamment dans les années 70). Les travaux de C. Strachey et D. Scott ont été fondateurs pour la séparation de ces deux domaines. On peut citer notamment C. Strachey qui disait :

> *La sémantique est là pour ce que nous voulons dire et la syntaxe pour comment nous avons à le dire.* C. Strachey

De manière générale, les concepteurs de langage de programmation définissent la sémantique de leur langage de manière informelle, au moyen de blocs de texte, écrits en langage naturel, ce qui laisse la place à l'ambiguïté, lors de l'interprétation d'un comportement décrit dans cette sémantique informelle. Pour éviter toute ambiguïté, les sémantiques formelles reposent sur l'utilisation des mathématiques afin d'exprimer rigoureusement le comportement des programmes.

Les motivations derrière la formalisation de la sémantique d'un langage sont nombreuses. Cela permet de visualiser tout comportement possible à l'exécution (même les cas d'erreurs), de lever toute ambiguïté pour un utilisateur du langage (car décrit formellement), de démontrer différentes sémantiques pour un même langage, l'équivalence de programmes (syntaxiquement différents), la validation de transformations de programme, des propriétés relatives au typage (comme la correction du typage par rapport à la sémantique ou la préservation du typage lors de l'exécution). Cela permet aussi le développement d'outils certifiés (interprètes, compilateurs, analyseurs statiques, etc.).

2.3.2 Les grandes approches de la sémantique formelle

Dans la littérature on remarque trois grandes approches de la sémantique formelle [Win93] , décrites par Xavier Leroy [7].

1. La sémantique axiomatique, reposant sur la logique de Floyd-Hoare, permet de décrire le comportement des programmes impératifs annotés par des assertions (formule logique). Elle permet d'assurer la validité des valeurs des variables avant et après l'exécution, si les instructions terminent.

2. La sémantique dénotationnelle est la première sémantique à avoir été introduit (début 70) par C. Strachey et D. Scott. Elle met en relation une instruction du programme avec sa dénotation. La dénotation est généralement une fonction, au sens mathématique du terme, c'est-à-dire associe des entrées à des sorties (plus généralement un domaine à un autre – théorie des domaines). Permettant de décrire de façon précise la sémantique du langage mais de façon la plus abstraite possible, elle reste toutefois difficile à manipuler et à implanter.

3. La sémantique opérationnelle met en relation un programme avec son résultat. Cette relation peut être de deux natures : «à grands pas» (dite «à la Kahn») ou «à petits pas» (dite «à la Plotkin»). Cette sémantique opérationnelle est donnée par un ensemble de règles d'inférence, reposant sur des structures inductives (théorie des types). La sémantique à grands pas met directement en relation un programme avec son résultat, ne montrant pas les étapes intermédiaires de calculs. Elle est sans doute la sémantique la plus proche d'une implantation. En revanche, elle parle plus difficilement des programmes qui ne terminent pas. Ce type de programmes

[7] http ://cristal.inria.fr/∼xleroy/mpri/prog/cours.pdf

peut être caractérisé plus facilement avec une sémantique à petits pas, qui décrit toutes les étapes intermédiaires de calcul entre le programme et le résultat final. Toutefois, les sémantiques à petits pas restent plus loin d'une implantation et sont moins efficaces en pratique.

2.3.3 Mécanisation des sémantiques formelles

Pour nous aider dans la formalisation des sémantiques, différents outils d'aide à la preuve existent. Par exemple, Coq [8] est un outil développé au sein de l'équipe INRIA πr^2. Il est fondé sur la théorie des types (calcul des constructions inductives). Coq est particulièrement approprié pour formaliser les sémantiques car il propose un très bon support pour les types inductifs. En effet, les types inductifs sont un moyen idiomatique de formaliser en Coq, et permettent en particulier de traduire presque directement les sémantiques opérationnelles exprimées sous la forme de règles d'inférence. Par ailleurs, il existe également des outils en Coq [TDD12] permettant d'extraire des interprètes à partir de la formalisation de sémantiques exprimées au moyen de types inductifs, ainsi que les preuves de correction correspondantes (c'est-à-dire que les interprètes extraits sont conformes aux sémantiques formalisées). Il existe également d'autres outils d'aide à la preuve qui possèdent un bon support pour l'induction et qui sont aussi appropriés pour la formalisation de sémantiques. On pourra citer en particulier Lego [9] ou Isabell [10].

2.3.4 La sémantique formelle des langages à objets et des langages à composants

Il existe assez peu de travaux sur la formalisation de la sémantique des langages à objets et la référence en la matière sont les travaux de M. Abadi et L. Cardelli [AC96], qui ont proposé une nouvelle approche dans la compréhension des langages objets, en présentant des calculs d'objets en développant une théorie autour d'eux, en couvrant leur sémantique et leurs règles de typage.

Cependant il n'existe pas, à notre connaissance, de travaux autour de la formalisation de la sémantique des langages à composants, sauf quelques rares exceptions comme [PQB01].

[8] https ://coq.inria.fr/
[9] http ://www.dcs.ed.ac.uk/home/lego/
[10] https ://isabelle.in.tum.de/

3

Problématique du stage de recherche

3.1 Le langage à composants, Compo

Développé au sein de l'équipe MAREL du LIRMM, le projet Compo[1] est un travail de collaboration et de réflexion autour du développement d'un langage pur composants, promu par son créateur C.Dony avec l'aide de C.Tibermacine et D.Delahaye, ainsi que des travaux de recherche réalisés dans les thèses de P.Spacek [SDT14] et L.Fabresse [Fab07].

3.1.1 Compo – un langage tout composant

L'idée du langage Compo est d'unifier les concepts des précédentes approches à composants autour d'un langage de programmation. Il s'inspire des ADLs, pour une description explicite des architectures, ainsi que l'idée de pouvoir définir explicitement le requis d'un composant. Il reprend le concept des LOC de mettre au même niveau une description de l'architecture et l'implémentation des composants. De la même manière que l'approche UML, il permet une visualisation graphique de l'architecture interne d'un composant. Avec la même philosophie que Smalltalk pour un monde «tout objet», Compo prône un monde «tout composant». Dans sa deuxième version de développement, Compo a intégré la réflexivité au sein de son méta-modèle, réalisée par les travaux de P.Spacek [SDT14], décrivant les descripteurs, ports et services comme étant des composants.

3.1.1.1 Composants et descripteurs

Un composant Compo est décrit par son descripteur, dont il est instance. Un descripteur permet de définir la structure et le comportement de chacune de ses instances en décrivant les ports et l'architecture et en définissant les services.

3.1.1.2 Des ports comme seul moyen de communication entre composants

Un port regroupe un ensemble d'interfaces (fournies ou requises). Comme l'approche de Fractal ou ComponentJ, les ports de Compo sont unidirectionnels, c'est-à-dire qu'un port est soit un ensemble d'interfaces fournies, on parle alors de port fourni (`provides`), soit un ensemble d'interfaces requises, on parle alors de port requis (`requires`). Contrairement aux ports bidirectionnels, les ports permettent un découplage du requis et du

[1] http ://www.lirmm.fr/compo/

besoin entre composants. En effet les approches comme Archjava ou en UML, les ports sont un ensemble d'interfaces fournies et/ou requises, laissant plus de souplesse dans le développement mais introduisant une perte de sécurité au niveau de l'architecture. En effet, un composant fournissant un service n'a pas forcement besoin d'avoir accès au service du composant auquel il se connecte. Il possède aussi une visibilité, interne ou externe. Par défaut un port est externe, visible depuis l'environnement extérieur. Un port peut devenir interne, visible uniquement dans l'environnement interne du composant, grâce à l'ajout de la primitive `internally`.

3.1.1.3 Les services

Un service est représenté par un nom et un ensemble de paramètres. Il représente une fonctionnalité que peut fournir chaque composant qui est instance du descripteur qui le définit. L'invocation du service d'un composant se réalise au travers d'un port requis du composant. Il peut aussi être défini dans des descripteurs de composants composites. Ainsi un composant composite encapsule un assemblage de composants et peut y rajouter des comportements grâce à des services. Ceci n'est pas possible dans les approches comme Fractal par exemple, où la définition de service ne peut se faire que par des composants primitifs.

3.1.1.4 Une description d'architecture explicite

Cette description d'architecture est contenue dans un bloc d'instructions, identifié par la primitive `architecture`. Elle permet de représenter les connexions entre les composants internes de notre composant avec l'instruction `connect` et de permettre la délégation entre un port externe fourni (resp. requis) de notre composant à un port externe fourni (resp. requis) d'un composant interne, avec l'instruction `delegate`.

3.1.1.5 Des composants composites

Compo intègre lui aussi la notion de composant composite, comme le font certaines des autres approches. Cependant, contrairement à ArchJava qui encapsule les composants internes de la même façon qu'une classe le fait avec ses attributs, Compo stocke ses composants internes à l'aide de ports internes requis. Une instance d'un composant interne est liée à ce port requis interne par son port fournis externe.

3.1.2 Compo – une architecture à composants d'un compteur

L'annexe A permet de décrire une architecture de composant qui simule le comportement d'un compteur. Le premier composant est décrit par le descripteur `Printer`. Il décrit le service `print`, qui permet d'afficher une chaîne de caractères. Le deuxième composant est décrit par le descripteur `Counter` qui permet de simuler l'incrémentation d'un compteur au travers de son service `increment`. Cette même valeur est encapsulée dans notre composant par un composant `Integer`. Notre composant a aussi besoin d'être connecté à un autre composant lui fournissant le service `print`.Le dernier composant est décrit par le descripteur `PrintableCounter` qui décrit notre architecture de composants. Il

encapsule nos deux composants et leurs connexions (ligne 62). Le service *main* qu'il fournit, aura comme comportement d'incrémenter le compteur et d'afficher sa valeur.

3.2 La problèmatique du passage des arguments dans les modèles à composants

3.2.1 La signification du passage d'un composant en argument

Les composants peuvent avoir besoin de s'échanger des données. Dans les autres approches et notamment les autres LOC ces données sont des objets, il est donc normal de se demander comment un composant passe à un autre composant une donnée, sachant que cette donnée est aussi un composant. Le composant A qui voudrait passer en argument un composant C à un service du composant B qu'il invoque, permettrait une connexion entre le composant C et le composant B (représenté dans figure 3.1). Le mécanisme de cette connexion étant décrit par [SDT14] :

> ***Choice 17 :*** *Arguments passing is made by the automatic establishment and removal of connections. Any component has a required collection port named args . During a service invocation, the arguments, i.e. ports, a1, a2, ..., an are each respectively connected to args[1], args[2], ..., args[n]. The identifiers of the parameters are actually alias identifiers of ports args . At the end of the execution of the service, all connections to args ports are removed. In the case when a service invocation is delegated, because the receiving port is delegated, the appropriate delegation connection for the args port is also automatically established and removed*

3.2.2 Les premières pistes du passage d'arguments

3.2.2.1 Le passage par fourni

Une connexion par port fourni (ou passage par fourni), figure 3.1, est représenté par la liaison *L1* entre le port *args[1]* du composant B et le port fourni *FC1* du composant C. Cette connexion entre B et C permet au composant B d'avoir accès au service de C, cependant on remarque que le composant A n'a plus de contrôle sur cette connexion car elle ne passe plus par lui. De plus si jamais le composant A décide de modifier sa connexion à C vers un autre composant (ici D par exemple), alors B gardera la connexion vers C, créant ainsi un comportement parallèle. Ce comportement peut s'avérer utile dans certain cas, par exemple si A modifie sa connexion plusieurs fois, il permettrait à B de conserver des connexions vers les anciens arguments de A (permettant ainsi d'avoir un historique des connexions).

3.2.2.2 Le passage par requis

Une connexion par port requis (ou passage par requis), figure 3.1, est représenté par la liaison *L2* entre le port *args[1]* du composant B et le port requis *RA1* du composant A. Cette liaison met en place un mécanisme de délégation entre le port de A et le port de B.

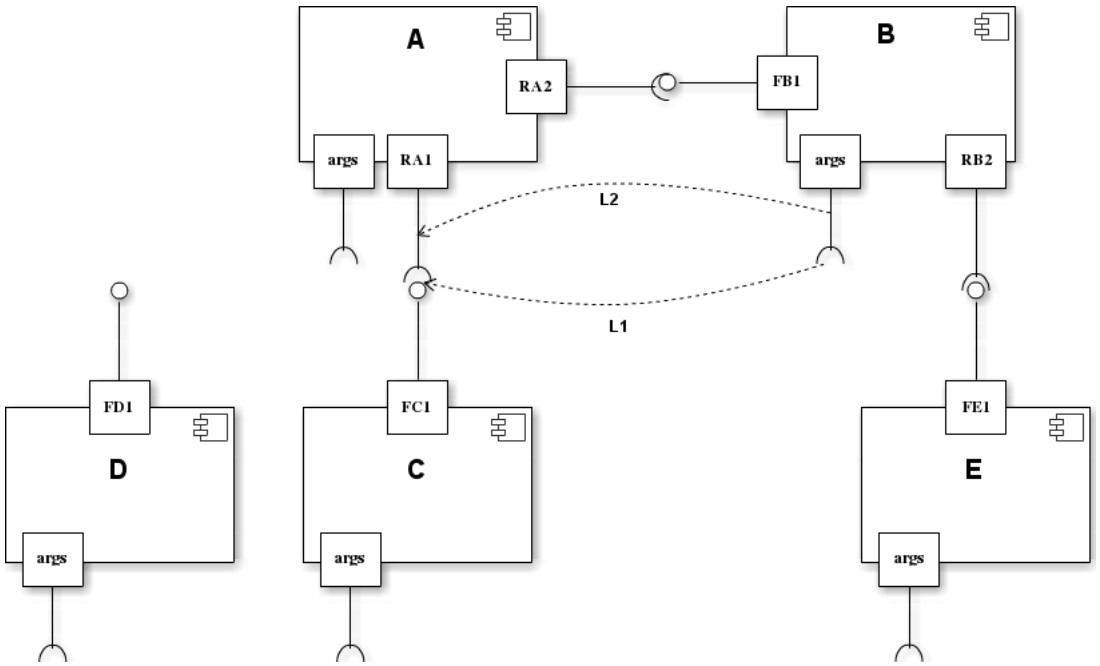

FIGURE 3.1: Exemple du passage en arguments d'un composant

Cela permet à B d'invoquer les services de C, tout en permettant à A d'avoir un possible contrôle sur cette connexion, car elle passe par son port. De plus si jamais A modifie sa connexion à C vers un nouveau composant, alors B aura lui aussi cette connexion vers D. Cela permet donc la création d'architectures très réactives aux changements de composants, ce qui fait défaut au passage par fourni.

3.2.3 Comparaison avec les passages d'arguments classiques

Nous avons vu précédemment les différents mécanismes de passage d'argument qui pouvaient exister dans les langages de programmation. Il est intéressant de se demander comment ces différents mécanismes pourraient se traduire dans un passage de composant en arguments.

On peut faire un premier parallèle du passage par fourni, qui semble permettre un mécanisme similaire au passage de référence en Java. En effet les modifications de l'état du composant C dans le service de B auront un impact sur le composant C. De plus si A modifie sa connexion vers C vers un autre composant et que B a conservé sa connexion vers C alors comme en Java B n'aura pas une référence vers D mais toujours vers C.

Le passage par requis ne peut pas se réaliser en Java, mais on peut avoir un comportement qui s'en rapproche en C++ par un référencement vers un pointeur. Si un objet A passe par référence un pointeur c à une méthode foo de B, et que B conserve cette référence dans un pointeur var, alors var aura comme référence l'adresse du pointeur c. Ainsi si A modifie la référence du pointeur c, alors B aura bien une référence vers un pointeur qui fera référence à ce nouvel objet.

3.2.4 Les potentialités découlant du passage par requis comme support au passage d'arguments

Le passage par requis permet déjà des potentialités sur cette connexion pour le passage en arguments, et qui demanderait peu de modifications afin d'être implémenté.

3.2.4.1 Un passage en lecture seule

Une connexion en mode «read-only» permettrait en effet de pouvoir passer un composant C à un composant B, où B ne pourrait pas modifier l'état du composant C, c'est-à-dire invoquer seulement des services de C qui n'ont pas un impact sur l'état de C, comparable à l'utilisation du mot clé const de C++. On peut donc imaginer la mise en place d'une nouvelle propriété sur les ports qui indiquerait s'ils sont en mode «read-only» ou non et de pouvoir modifier cette propriété selon le besoin. On peut aussi imaginer un nouveau type de port qui ne permettrait que le passage en «read-only» et le composant A devrait choisir quel type de port il veut utiliser. Ce mécanisme «read-only» peut cependant être réalisé manuellement par la mise en place d'un composant placé entre nos deux composants, qui filtrerait les services que peut invoquer notre composant.

3.2.4.2 Un passage révocable

Nous avons vu que le passage par requis permettait au composant A d'exercer un contrôle sur la connexion de B vers C. On peut donc imaginer là aussi une nouvelle propriété ou un nouveau type de port permettant de «couper» la connexion entre B et C mais aussi de pouvoir la rétablir selon le besoin. La difficulté de la propagation d'un passage révocable par le passage par requis est minime, si B passe C en argument au composant E, alors supprimer la délégation entre le port de A et le port B, supprimera la délégation implicite de E vers C.

3.2.5 De nouvelles pistes ouvertes par les résultats de recherches récentes

3.2.5.1 Les références comme entité de première classe

Faire des références une entité de première classe permet d'exercer un contrôle sur les références des objets, permettant de construire des systèmes objet avec une plus grande sécurité. En apportant des comportements comme des références en lecture seule, révocables. On pourra notamment regarder les travaux portant sur les langages à typage statique [CPN98] ou encore les travaux de [ADDT15] portant sur les langages à typage dynamique.

Cependant on pourra discuter de la propagation du contrôle des références, en regardant un exemple inspiré de [ADDT15]. Un objet A passe une référence révocable vers un objet C à un objet B. Cependant B a aussi une autre référence indirecte vers C en passant par D. Alors si A révoque la référence révocable vers C, cela a-t-il un impact sur la référence indirecte vers B ?

On pourra aussi regrouper les travaux de [AA06], qui essaye de lutter contre l'aliasing, en contrôlant les échanges de références, en regroupant les objets dans des domaines et

en explicitant les lien entre eux. Mettant aussi en place des mots clés permettant de gérer les droits d'accès des objets, comme **unique** (ne peut être pointé que par une seule référence), **owned** (ne peut être accessible que par des objets du même domaine), **shared** (partagé entre les domaines).

On remarque cependant que ces différents mécanismes se retrouvent conceptuellement dans les ports de Compo, et pourraient être implémentés sans demander un grand effort de conception, par l'ajout de nouveaux types de ports par exemple.

3.2.5.2 La réification des connecteurs

Un connecteur permet de représenter la sémantique d'une connexion entre deux ou plusieurs instances. Cependant les langages à objets ne permettent pas de fournir une représentation explicite des connexions entre instances, car étant implicitement comprises dans leurs références. Dans le monde composant cette connexion est explicitement décrite et représente une liaison entre deux ou plusieurs éléments connectables (e.g. ports ou interfaces) [ICG+04].

[ASCN03] propose donc d'intégrer les connecteurs au LOC ArchJava, afin de permettre au développeur de pouvoir modifier la sémantique d'un connecteur à l'exécution et de modifier le comportement de vérification de type entre les ports. Par défaut la compatibilité de deux interfaces est la même que dans Java, cependant ajouter un nouveau connecteur qui hérite de la classe *Connector* et surcharger sa méthode *typecheck*, permet de modifier la vérification de cette compatibilité. Par exemple, dans [ASCN03], un nouveau connecteur $TCPConnector$ est défini, ne supportant une connexion qu'entre exactement deux ports et l'ensemble des arguments doivent être des sous-types de l'interface *Serializable* de Java. Le développeur peut ensuite s'il le désire lors de la définition d'une connexion (**connect**), spécifier un connecteur particulier (**with new** [*connecteur*])

Réifier les connecteurs dans Compo ne serait que redondant car leur mécanisme est déjà incorporé dans les ports. Cependant la possibilité de pouvoir ajouter un nouveau comportement de vérification peut être utile dans certains cas d'utilisation et se réaliserait par l'utilisation d'un nouveau type de port.

3.2.6 La signification du retour d'un composant à une invocation de service

En complément des interrogations sur le passage de composants en arguments, il est intéressant de se demander ce que signifie retourner un composant suite à une invocation de service. Une connexion est aussi réalisée entre le composant retourné et le composant qui réalise l'invocation. On prendra comme exemple, le composant E qui serait le retour d'un invocation de service du composant B par le composant A, figure 3.1.

Dans le cas d'un retour par port fourni (ou retour par fourni), le même mécanisme de connexion est utilisé entre les ports de nos composants. Alors, une connexion entre un port requis de A et le port fournis $FE1$ de E est mise en place, donnant ainsi une connexion directe vers le composant E au composant A sans avoir à passer par B.

Dans le cas d'un retour par port requis (ou retour par requis), le même mécanisme de délégation est utilisé entre les deux ports requis. Ainsi, une délégation entre un port

requis de A et le port requis $RB2$ est mise en place, donnant ainsi à B le contrôle sur la connexion entre A et E.

3.3 Formaliser la sémantique de Compo

3.3.1 Évolution du noyau sémantique existant et formalisation des nouveaux mécanismes

La première étape de la formalisation de la sémantique du langage Compo, consiste en la définition des règles d'inférence. Partant d'une base de règles déjà définie par D.Delahaye, nous devrons continuer la définition des règles d'inférence pour prendre en compte l'ensemble des mécanismes de Compo, qui se réaliseront sur papier au cours de cette première étape. Cette définition portera cependant sur un noyau de Compo non réflexif, qui sera intégré dans de futurs travaux.

Nous avons vu précédemment que la réflexion autour de la problématique du passage en argument d'un composant nous suggère différentes pistes. Ainsi la formalisation des différents modes de passage nous apportera un regard neuf dans le choix du mode et de pouvoir comparer les modes entre eux en définissant formellement leur comportement.

3.3.2 Mécanisation en Coq et génération d'un interprète certifié

La deuxième étape sera effectuée avec l'outil Coq, qui mécanisera les règles d'inférence au moyen de types inductifs. Le choix de Coq comme outil pour la mécanisation se justifie par le fait qu'il est particulièrement efficace pour formaliser une sémantique opérationnelle exprimée sous la forme de règles d'inférence. De plus au cours de ma formation j'ai eu l'occasion de l'utiliser afin de mécaniser la sémantique d'un mini-langage fonctionnel.

Le but ultime à atteindre serait l'extraction d'un interprète certifié, qui peut se réaliser à l'aide d'un plugin développé par D.Delahaye et un doctorant [TDD12].

4

Conclusion

Nous avons vu au cours de cette étude les différentes familles à composants qui se regroupent autour d'une idée commune, voyant un composant comme étant une entité boîte noire, communiquant avec le monde extérieur au moyen d'interfaces (requises ou fournies). Cette idée de composant apporte au Génie Logiciel une nouvelle approche pour répondre aux problématiques de modularité et de réutilisabilité dans la production de logiciels.

Nous avons aussi vu que la formalisation de la sémantique d'un langage permet de définir le comportement d'un programme de manière mathématique. Les motivations sont nombreuses, allant de la visualisation de tout comportement possible à l'exécution, en passant par la levée de toute ambiguïté pour un utilisateur du langage, jusqu'à des propriétés relatives au typage. Cependant cette formalisation de la sémantique d'un langage à composants est à l'heure actuelle et à notre connaissance inexistante.

Nous avons présenté le langage Compo qui se place dans le contexte d'un langage pur composants qui a pour but d'unifier les concepts des différentes familles à composants, autour d'un seul langage de programmation qui met en avant la notion de «tout composant». Cette notion apporte de nouvelles problématiques, notamment le passage de paramètres. En effet, nous avons vu que les autres familles à composants et plus particulièrement les LOCs, permettent aux composants de s'échanger des données entre eux qui sont des objets, qui dans le langage Compo permet aux composants de s'échanger des données par connexion à d'autres composants.

Cela pose donc la question de savoir la nature de cette connexion et les contrôles qu'ont les composants sur elle. Les premières idées présentes pour l'instant mettent en avant un passage par requis et un passage par fournis. Cependant nous avons vu que ces passages n'ont pas encore atteint leur plein potentiel, notamment des potentialités mis en avant par le passage par requis (passage en lecture seule et passage révocable). Les passages de paramètres classiques sont aussi une source d'inspiration, avec des premières similarités avec le passage par référence et la correspondance avec les autres passages dans le monde des composants est aussi une bonne piste pour avancer dans cette problématique.

L'objectif durant le stage sera donc une approche originale autour de la formalisation de la sémantique d'un langage à composants, mécanisé avec l'outil Coq, tout en proposant une implémentation d'un ou plusieurs modes de passage de paramètres dans un monde «tout composant», par l'évolution d'un passage existant et/ou la création d'un ou plusieurs nouveaux modes de passage de paramètres.

A

Une architecture à composants d'un compteur en Compo

```
1  Descriptor Printer extends Component
2  {
3    provides {
4      default : {
5        print(txt);
6      }
7    }
8
9    service print(txt) {
10     Transcript.crShow(txt);
11   }
12 }
13
14 Descriptor Counter extends Component
15 {
16   provides {
17     default: {
18       increment();
19       show()
20     };
21   }
22
23   internally requires {
24     val : Integer;
25   }
26
27   requires {
28     printer : { print(txt); };
29   }
30
31   architecture {
32     connect val@self to 0;
33   }
34
35   service increment() {
36     | t |
37     t := val + 1;
38     connect val to t;
39   }
40
41   service show() {
```

```
42      printer.print(val.asString());
43    }
44  }
45
46  Descriptor PrintableCounter extends Component
47  {
48    provides {
49      default: {
50        main();
51      }
52    }
53
54    internally requires {
55      c : Counter;
56      p : Printer;
57    }
58
59    architecture {
60      connect c to default@(Counter.new());
61      connect p to default@(Printer.new());
62      connect printer@c to default@p;
63    }
64
65    service main(){
66      c.incrementer();
67      c.incrementer();
68      c.incrementer();
69      c.show();
70    }
71  }
```

Listing A.1: Une architecture composant d'un compteur en Compo

Bibliographie

[AA06] Marwan Abi-Antoun and Jonathan Aldrich. Bringing ownership domains to mainstream java. In *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 702–703, 2006.

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1996.

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava : connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pages 187–197, 2002.

[ADDT15] Jean-Baptiste Arnaud, Stéphane Ducasse, Marcus Denker, and Camille Teruel. Handles : Behavior-propagating first class references for dynamically-typed languages. *Sci. Comput. Program.*, 98 :318–338, 2015.

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language, Third Edition*. Addison-Wesley, 2000.

[ASCN03] Jonathan Aldrich, Vibha Sazawal, Craig Chambers, and David Notkin. Language support for connector abstractions. In *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings*, pages 74–102, 2003.

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in java. *Softw., Pract. Exper.*, 36(11-12) :1257–1284, 2006.

[Boe76] B. W. Boehm. Software engineering. *IEEE Trans. Comput.*, 25(12) :1226–1241, December 1976.

[CPN98] David G. Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, British Columbia, Canada, October 18-22, 1998.*, pages 48–64, 1998.

[Don89] Christophe Dony. *Langages a objets et genie logiciel, application a la gestion des exceptions et a l'environnementde mise au point.* PhD thesis, Paris 6, 1989.

[Fab07] Luc Fabresse. *Du découpage à l'assemblage non anticipé de composants : Conception et mise en oeuvre du langage à composants Scl.* PhD thesis, Université Montpellier II-Sciences et Techniques du Languedoc, 2007.

[Hor84] Ellis Horowitz. *Fundamentals of programming languages (2. ed.).* Computer software engineering series. Computer Science Press, 1984.

[ICG+04] James Ivers, Paul Clements, David Garlan, Robert Nord, Bradley Schmerl, and Oviedo Silva. Documenting component and connector views with uml 2.0. Technical Report CMU/SEI-2004-TR-008, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2004.

[Kru98] Philippe Kruchten. Modeling component systems with the unified modeling language. Int. Workshop on Component-Based Software Eng., 1998.

[McI68] M. D. McIlroy. Mass-produced software components. *Proc. NATO Conf. on Software Engineering, Garmisch, Germany*, 1968.

[Neb] Jean-Louis Nebut. *Ada pour l'écriture de composants logiciels : projet de norme Ada 9X.* Collection Informatique. Ed. Technip, 1994 (05-Gap), Paris.

[PC98] Nat Pryce and Steve Crane. Component interaction in distributed systems. In *Fourth International Conference on Configurable Distributed Systems, 1998, Proceedings, Annapolis, MA, USA, 6 May, 1998*, pages 71–78, 1998.

[PQB01] Frédéric Peschanski, Christian Queinnec, and Jean-Pierre Briot. A typeful composition model for dynamic software architectures, Technical Report , Univ. of Paris VI, 2001.

[SDT14] 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 *Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '14, pages 13–22, New York, NY, USA, 2014. ACM.

[Sit72] Richard L Sites. Algol w reference manual. Stanford University, 1972.

[Spea] OMG Adopted Specification. Uml 2.0 superstructure specification.

[Speb] Sun Microsystems Specification. Ejb 3.1 core contracts and requirements.

[SSP08] João Costa Seco, Ricardo Silva, and Margarida Piriquito. Componentj : A component-based programming language with dynamic reconfiguration. *Computer Science and Information Systems*, 5(2) :63–86, 2008.

[Str00] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[Szy99] Clemens Szyperski. Components vs. objects vs. component objects. In *Proceedings of OOP*, 1999.

[Szy02] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[TDD12] Pierre-Nicolas Tollitte, David Delahaye, and Catherine Dubois. Producing Certified Functional Code from Inductive Specifications. In Chris Hawblitzel and Dale Miller, editors, *Certified Programs and Proofs (CPP)*, volume 7679 of *Lecture Notes in Computer Science (LNCS)*, pages 76–91, Kyoto (Japan), December 2012. Springer.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages : An Introduction*. MIT Press, Cambridge, MA, USA, 1993.