Fédération de composants : une architecture logicielle pour la composition par coordination

Jorge Villalobos

UNIVERSITE JOSEPH FOURIER – GRENOBLE I

THESE

pour obtenir le grade de

DOCTEUR de l'Université Joseph Fourier de Grenoble

(arrêtés ministériels du 5 juillet 1984 et du 30 mars 1992)

Discipline : Informatique

présentée et soutenue publiquement par

Jorge VILLALOBOS

le 15 juillet 2003

Fédération de Composants : une Architecture Logicielle pour la Composition par Coordination

Directeur de thèse :

Jacky ESTUBLIER

JURY

F. Ouabdesselam, Professeur à l'Université Joseph Fourier	(Président)
J. Bezivin, Professeur à l'Université de Nantes	(Rapporteur)
J.M. Geib, Professeur à l'Université de Lille	(Rapporteur)
J. B. Stefani, Directeur de Recherche à l'INRIA	(Examinateur)
R. Balter, ScalAgent Distributed Technologies	(Examinateur)
J. Estublier, Directeur de Recherche au CNRS	(Directeur de thèse)

Thèse préparée au sein du Laboratoire LSR – Equipe ADELE

A Vicky.
elle sait pourquoi…

Je tiens à remercier,

M. F. Ouabdesselam, Professeur à l'Université Joseph Fourier de Grenoble, de me faire l'honneur de présider mon jury de thèse.

M. J. Bezivin, Professeur à l'Université de Nantes, et M. J.M. Geib, Professeur à l'Université de Lille, d'avoir accepté de rapporter cette thèse.

M. J. B. Stefani, Directeur de Recherche à l'INRIA, et M. R. Balter, de la société ScalAgent Distributed Technologies, pour sa participation à ce jury.

M. Jacky Estublier, Directeur de Recherche au CNRS, qui m'a accueilli dans son équipe. Je lui suis très reconnaissant du point de vue personnel et professionnel.

M. Pierre-Yves Cunin, pour la lecture du document, et pour son aide dans tous les aspects administratifs.

Frédéric Duclos, Rémy Sanlaville, German Vega et Sabrina Lefievre pour la lecture du document, et leurs commentaires.

Tous les membres de l'équipe Adèle.

Je voudrais remercier spécialement Anh-Tuyet Le, avec qui j'ai partagé le quotidien de la recherche, avec toutes les petites crises et doutes, qui finalement ont aidé à donner forme aux idées et aux résultats présentés dans ce rapport.

Et finalement, je voudrais dire à Vicky que c'est grâce à elle que je suis là…

Résumé

La construction de logiciels à partir de la composition d'éléments existants de haut niveau est un objectif fondamental, depuis plus de 20 ans. On pensait que les objets permettraient d'atteindre cet objectif mais, avec les années, on a compris que la composition est un problème beaucoup plus compliqué qu'il ne peut le paraître.

Dans cette thèse, nous avons commencé par étudier l'approche orientée objet, pour identifier ses problèmes de composition et d'évolution. A partir de ce résultat, nous avons analysé l'approche des composants. Nous avons identifié et caractérisé trois types de composants : les composants fonctionnels (remontée du niveau d'abstraction et réification du concept de module), les composants de composition (remontée du niveau d'abstraction des objets) et les composants de coordination (éléments qui permettent à un groupe de composants de divers domaines de travailler ensemble). Ce cadre conceptuel nous a permis d'étudier les aspects fondamentaux et structuraux des composants, et de déterminer leurs limitations. Dans ce cadre, nous avons comparé l'approche des composants et de la programmation orientée aspect (POA), et nous avons identifié trois niveaux de composition : l'intégration (approche boîte blanche), la connexion (approche boîte noire et grise) et la coordination.

Nous nous sommes intéressés dans cette thèse au problème de la composition de composants par coordination, dans lequel un groupe de composants de domaines distincts doivent travailler ensemble. Sachant que toute connexion directe entre deux composants de deux domaines différents engendre des problèmes de pollution sémantique, nous avons exploré la coordination comme une façon de composer. Pour aborder ce problème, nous avons caractérisé le problème de la coordination entre mondes, et montré les modèles de conteneurs comme une solution rudimentaire à ce problème. Nous avons étudié le monde de la coordination, et nous avons introduit la "fédération", comme une architecture logicielle qui matérialise la coordination, et qui permet de structurer les applications comme un ensemble de mondes qui coopèrent pour atteindre un but commun. Nous avons défini l'univers commun (la réification des relations entre les composants de domaines distincts) et les contrats de coordination (la matérialisation des règles de jeu), comme les structures de base d'une fédération.

Finalement, nous avons validé les idées de manière pratique. Nous avons construit une plate-forme de support pour une fédération de composants, et nous avons implémenté une application d'évaluation sur cette plate-forme. Bien qu'il faille aller plus loin dans les tests, nous considérons que les résultats sont très satisfaisants, et que nous avons pu démontrer la viabilité de l'approche.

Mots clés : Programmation orientée objet, Modèles de composants, Fédération logicielle, Programmation orientée aspect, Composition, Coordination, Composition par coordination, Contrats de coordination, Développement orienté par les modèles.

Components Federation: a Software Architecture for Composing by Coordination

Abstract

Initially we studied the object-oriented approach and identified problems related to object composition and evolution. Since component models were developed to help solve these problems, we then went on to analyze these as well. We identified and characterized three types of components: functional components (which increase of the level of abstraction and reification of the concept of module), composition components (which change the abstraction level in the object model) and coordination components (elements which allow a group of components of various domains to work together). This conceptual framework allowed us to study the fundamental and structural aspects of the components, and to determine their limitations. Within this framework, we compared the component approach with the aspect-oriented programming (AOP), and identified three levels of composition: integration (white box approach), connection (block box and gray approach) and coordination.

We studied the problem of composition by coordination, in which a group of components belonging to different domains must work together in an application. Considering that any direct connection between two components of two different domains generates evolution problems, we explored coordination as a composing mechanism. We characterized the problem of coordination between domains, and showed that container models are a simplistic solution to this problem. We studied the coordination world, and we introduced the "federation", as a software architecture that makes it possible to structure an application as a set of components, which cooperate (without any direct connection) to achieve a common goal. We defined the common universe (relationships between the components of distinct domains) and the coordination contracts (materialization of the game rules) like the basic structures of a federation. We implemented a framework and an application to validate the proposed approach.

Keywords: Object-Oriented Programming, Component Models, Software Federations, Aspect-Oriented Programming, Software Composition, Software Coordination, Composition by Coordination, Coordination Contracts, Model-Oriented Development.

Table des matières

Chapitre I – Introduction

Chapitre II – Objets

Chapitre III – Composants

Chapitre IV – Les approches non-composants

Chapitre V – Composition de composants par coordination

Chapitre VI – Fédérations de composants

Chapitre VII – Expérimentation et résultats pratiques

Chapitre I
Introduction

1. Objectif du chapitre

L'objectif de ce chapitre est de présenter l'idée générale du travail développé : le sujet, les objectifs concrets et la méthodologie.

Le chapitre est structuré comme suit : dans la section 2, nous présentons les objectifs de la thèse. Dans la section 3, nous énonçons les lignes méthodologiques principales. Et enfin, dans la section 4 nous décrivons la structure du reste du document.

2. Objectifs de la thèse

2.1. Le problème

La construction et la structuration de logiciels à partir de la composition d'éléments de haut niveau est le rêve depuis plus de 20 ans des informaticiens. On pensait qu'avec les objets, ce rêve serait une réalité, mais avec les anées on a compris que la composition est un problème beaucoup plus compliqué qu'il ne peut le paraître.

Actuellement, nous écoutons souvent que les composants sont la réponse à nos attentes, mais, malheureusement, nous sommes encore très loin d'une vraie solution. Le problème est, surtout, que pour composer, il faut bien comprendre les éléments de composition, et non seulement se contenter de les connecter. Pour l'instant, les composants continuent à être un concept flou, dans lequel on a du mal à distinguer la technologie de l'essence, et les idées importantes, des conseils aux programmeurs.

Nous nous sommes intéressés dans cette thèse au problème de la composition de composants, et particulièrement au problème de composition sur des domaines distincts, dans lequel, plutôt qu'une simple connexion entre composants, une coordination est nécessaire pour les faire travailler ensemble. C'est ce que nous appellerons une composition par coordination.

Nous pensons que pour composer, il est indispensable de pouvoir réfléchir et travailler au niveau du modèle du domaine du problème, en gardant à tout moment une vision globale abstraite. Dans ce sens, notre travail est très proche de l'approche MDA (*Model Driven Architecture*) [AK02b, Bez01, DSo01] proposé par l'OMG (*Object Management Group*).

2.2. Les objectifs spécifiques

Dans ce travail, nous nous sommes fixé quatre objectifs concrets :

- Définir un cadre conceptuel de base pour identifier les éléments fondamentaux du problème et les approches actuelles. Nous voulons analyser en profondeur les objets, les composants et les aspects, trois des lignes les plus prometteuses de l'actualité dans le génie logiciel. Nous considérons cette étape d'importance vitale, car elle va nous permettre d'introduire une terminologie commune aux trois domaines, en facilitant ainsi leur comparaison et l'étude de leurs limitations.

- Étudier le problème de composition par coordination, et proposer un modèle général pour la coordination d'un ensemble de composants. A partir de ce modèle, l'objectif est d'introduire une fédération comme une architecture logicielle pour construire des applications comme un ensemble de composants travaillant ensemble pour atteindre un but commun. Considérée de cette façon, une fédération peut être utilisée, en particulier, pour faire inter-opérer un ensemble d'applications.

- Développer une plate-forme complète pour supporter une fédération. Cette plate-forme doit fournir les outils de définition des éléments d'une fédération, aussi bien que les outils d'interprétation du modèle de coordination. Étant donné que nous voulons que cette implémentation puisse s'utiliser comme base pour des applications industrielles, elle doit supporter tous les aspects pratiques importants (i.e. distribution, sécurité, hétérogénéité, etc.), en plus d'être performante et flexible.

- Développer quelques applications réelles (avec des partenaires industriels) pour évaluer l'approche. Ceci va nous permettre de valider la plate-forme de base développée, mais surtout, d'étudier les aspects méthodologiques et pratiques liés à la construction d'applications avec une architecture de fédération.

3. Méthodologie de travail

Nous pouvons résumer la méthodologie par rapport aux aspects suivants :

- Étant donnée la confusion actuelle concernant la définition de la plupart des concepts clés en génie logiciel (comme "objet", "composant" ou même "architecture"), nous considérons important que le document soit auto-contenu dans la mesure du possible. Ceci veut dire que nous ferons l'effort de donner une définition à chaque terme du domaine que nous allons utiliser.

- Nous pensons qu'il est aussi important de montrer la construction du résultat que le résultat lui-même. Pour cette raison nous allons laisser une trace de la réflexion et des principales décisions prises tout au long de la thèse. Cette manière de travailler va permettre de rendre explicite l'immense quantité d'options et de possibilités qui n'ont pas été explorées, et qui peuvent inspirer et guider des travaux futurs dans la même direction.

- Nous donnons beaucoup d'importance au fait de pouvoir expliquer chaque partie de la solution en termes du problème que nous voulons résoudre : il doit exister une raison claire à la présence de chaque élément dans la solution, et cette raison doit remonter jusqu'au problème. Il faut éviter la tendance de mélanger dans la solution tous les mécanismes et techniques qui peuvent paraître utiles pour résoudre le problème.

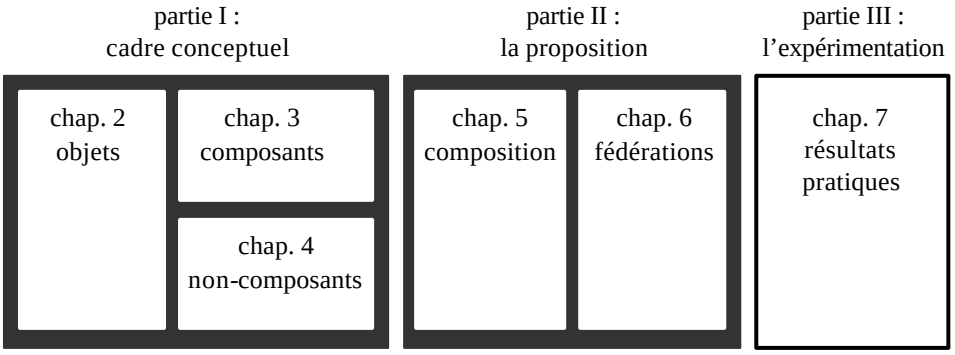
- Dans notre travail nous allons retrouver plusieurs niveaux d'abstractions différents. Au fur et a mesure que nous avançons dans la thèse et que nous passons de l'abstrait vers le concret, nous allons rencontrer des éléments de plusieurs domaines qu'il faut maintenir à l'écart. Il faut éviter tout mélange de niveau d'abstraction.

- Pour nous c'est très important que l'expérimentation des résultats se fasse sur des problèmes réels, avec des clients réels et des besoins réels. C'est ainsi qu'on peut valider vraiment les résultats et raffiner les idées.

- Etant donné le haut niveau d'abstraction dans lequel nous voulons exprimer quelques idées, nous avons décidé de nous appuyer souvent sur une syntaxe graphique pour faciliter les explications. Cette syntaxe est intuitive et informelle, mais aussi précise que possible. Pour cette raison, nous considérons les figures comme une partie fondamentale du discours, et non seulement une illustration.

4. Plan de la thèse

Ce rapport de thèse est divisé en trois grandes parties :



- Partie I : Définition de la problématique et cadre conceptuel de travail.

  Cette partie est composée de trois chapitres. Un chapitre de base (chapitre 2), qui part de la problématique de développement d'un programme, qui montre le modèle d'objets, et qui termine en montrant les limitations de ce modèle. Un deuxième chapitre (chapitre 3) qui étudie les modèles de composants et les diverses significations du mot "composer". Dans ce deuxième chapitre on introduit une classification des modèles de composants : les composants de composition, les composants fonctionnels et les composants de coordination. Le troisième chapitre (chapitre 4) montre des solutions au problème de la structuration d'un logiciel, différentes de celles des composants. Il s'agit notamment de la programmation par aspects. A la fin de ce dernier chapitre nous discutons sur la différence entre "intégrer", "connecter" et "coordonner" comme mécanismes de composition.

  Il est important de dire que les résultats de cette partie peuvent être réutilisés hors du contexte de cette thèse. Nous les considérons comme un apport important du travail, et non seulement comme une description de l'état de l'art.

- Partie II : Étude du problème de composition par coordination.

  Cette deuxième partie du document comporte deux chapitres. Au début du premier chapitre (chapitre 5) nous fixons clairement les objectifs du travail, en termes des éléments introduits dans la première partie de la thèse. Ensuite, nous étudions le problème de la composition par coordination, en le séparant en facettes indépendantes pour faciliter sa présentation. Dans cette partie nous avons décidé d'étudier le problème de composition par coordination, en modélisant les éléments et les structures d'un monde imaginaire chargé de faire travailler un ensemble de mondes. Dans le deuxième chapitre de cette partie (chapitre 6) nous présentons la fédération de composants, une architecture pour matérialiser le monde de la coordination.

- Partie III : Expérimentation et résultats pratiques.

  Cette dernière partie du document est composée d'un seul chapitre (chapitre 7). Ce chapitre montre tous les résultats pratiques du travail, et les leçons apprises. Ce chapitre a deux parties : (1) la description de l'implémentation d'une plate-forme pour exécuter une fédération, et (2) la description des applications développées sur cette plate-forme pour évaluer la viabilité de l'approche proposée dans la thèse.

  Dans ce chapitre nous ne prétendons pas montrer les détails des implémentations, mais plutôt donner une idée du type d'expérimentation faite. Nous allons utiliser aussi ce chapitre pour illustrer quelques-unes des idées développées tout au long du document.

Finalement, il y a un chapitre (chapitre 8) qui présente les conclusions et le travail futur.

Bonne lecture…

Chapitre II
Objets

1. Introduction

Ce chapitre décrit le modèle d'objets et les limitations inhérentes à ce modèle.

Nous ne prétendons pas montrer une formalisation du modèle d'objets, ni comparer le modèle d'objets avec les autres modèles existants. Le seul objectif de ce chapitre est de fournir un ensemble de définitions précises et une terminologie qui va permettre d'exprimer de manière claire la problématique dans laquelle se situe ce travail, ainsi que les solutions proposées. Etant donné qu'actuellement les mêmes termes (objet et composant, pour n'en citer que deux) sont utilisés dans des contextes différents pour exprimer des choses distinctes, nous avons considéré important de commencer de cette façon.

Dans certains cas, nous avons sacrifié la rigueur à la simplicité de la présentation, ce qui est aussi l'un de nos objectifs principaux.

2. Mondes, problèmes et solutions

Dans cette partie nous introduisons les concepts de base, présents dans plusieurs disciplines, concernant la définition et la résolution de problèmes. De façon informelle nous pouvons dire qu'un problème se déroule dans un monde, et qu'une solution décrit la façon de changer l'état de ce monde pour arriver à une situation où le problème a été résolu.

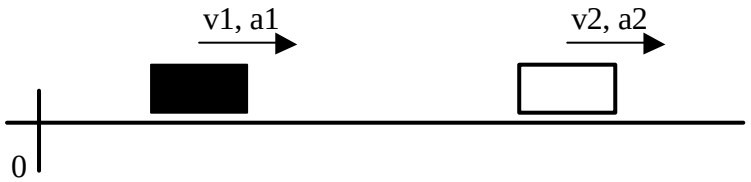
De manière plus précise, nous pouvons dire qu'un monde M est un système fermé, composé d'un ensemble d'éléments, un ensemble de relations entre ces éléments et un ensemble de règles de fonctionnement (dit son comportement) qui lui permettent d'évoluer dans le temps.

A un moment donné, un monde a un état, qui correspond à l'ensemble des états de ses éléments, incluant dans le concept d'état d'un élément les relations avec les autres éléments du monde.

Un monde a un ensemble d'états possibles associés, qui inclut toutes les situations imaginables du monde, et qui est défini par ses règles de fonctionnement.

Nous proposons l'exemple suivant, hors du domaine de l'informatique, pour illustrer les concepts introduits : supposez qu'on a un monde correspondant à un espace linéaire infini, avec un point de départ (dit le point zéro), dans lequel il y a deux mobiles (le mobile blanc

et le mobile noir) en mouvement. L'état d'un mobile est défini par sa position (par rapport au point zéro du monde), la vitesse (positive ou négative) et son accélération. L'état du monde est défini par l'état des deux mobiles. Les règles du monde disent que les deux mobiles ne peuvent pas occuper la même position au même moment (une contrainte), que la relation entre position et vitesse est définie par l'équation XX, que la vitesse varie avec l'accélération selon l'équation YY et que, en cas de choc, les mobiles vont rebondir en suivant la loi ZZ de la physique. Ces règles décrivent le comportement du monde. Les états possibles de M sont toutes les situations imaginables du monde.



Dans le cas le plus simple, on peut dire qu'un problème P dans un monde M est défini par deux états possibles de M. Le problème consiste alors à passer de l'état dit initial à l'état dit final, où le problème a été résolu. Dans ce cas, on dit que M est le monde du problème P.

Dans notre exemple, un problème simple pourrait être défini de la façon suivante :

- état initial : le mobile noir est arrêté dans la position 0, le mobile blanc est arrêté dans la position 10
- état final : le mobile 1 est arrêté dans la position 10, le mobile blanc est arrêté dans la position 11

Dans le cas général, un problème peut être défini comme un couple d'ensembles d'états de M : les situations initiales et les situations finales possibles. Le problème consiste, dans ce cas, à passer de n'importe quelle situation initiale, à n'importe quelle situation finale.

Typiquement, les ensembles de situations initiales et de situations finales ne sont pas définis par extension, mais caractérisés par deux assertions, une précondition et une post-condition. La précondition définit l'ensemble des états initiaux possibles et la post-condition, l'ensemble des états finaux.

Dans l'exemple des mobiles, un problème pourrait être défini de la façon suivante :

- précondition : le mobile noir avance vers la droite, le mobile blanc est arrêté
- post-condition : le mobile noir est arrêté juste avant la position du mobile blanc, qui a la même position initiale

Chaque assertion caractérise un ensemble d'états du monde, et ces ensembles définissent le problème.

Une solution S est capable de résoudre un problème P, si elle est capable de faire passer le monde M de chaque état initial de P à un état final de P. D'un point de vue purement fonctionnel, on peut voir, alors, une solution S comme une fonction :

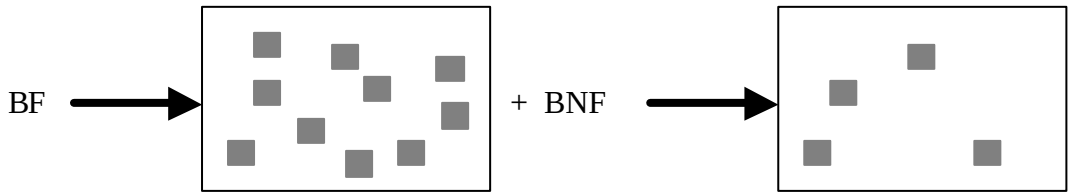
S(P, M) : états initiaux ? états finaux

Dans cette optique, P est nommé aussi un besoin fonctionnel (BF) dans le monde M, parce qu'il caractérise une fonction S, qui correspond à une fonctionnalité attendue (on explique ce qu'on veut faire, mais pas comment le faire).

D'un point de vue structurel, une solution S est un algorithme exprimé en termes de stimulus vers le monde. L'état final étant le résultat de la réaction du monde aux stimulus en suivant ses règles de comportement. Si pour tout état initial du problème, l'exécution de la solution amène le monde à un état final, on dit que S est une solution de P.

Dans le problème des mobiles, une solution est constituée d'une séquence de stimulus sur les mobiles (accélérations positives et négatives) qui permet dans toutes les situations initiales possibles du problème d'arriver à une situation finale. Pour cela, il est indispensable que l'algorithme de la solution puisse consulter l'état de chaque mobile, et réagir en conséquence.

Un problème peut avoir zéro ou plusieurs solutions. D'un point de vue fonctionnel on ne peut pas distinguer une solution d'une autre. Toutes les caractéristiques qui différencient une solution d'une autre, s'appellent des caractéristiques non-fonctionnelles (CNF). Un besoin non fonctionnel (BNF) est alors une contrainte ou propriété exigée à une solution pour être acceptable ou valide. A la différence des BF qui sont toujours exprimés en termes du monde du problème, les BNF sont exprimés en termes du monde de la solution, comme un ensemble de conditions sur les caractéristiques non-fonctionnelles.



Pour le problème des mobiles, on peut trouver plusieurs solutions distinctes. Par exemple, d'abord arrêter le mobile noir appliquant l'accélération négative maximale, et ensuite avancer doucement vers le mobile blanc. Une autre solution est de laisser d'abord les mobiles se choquer et, après, freiner et reculer doucement. Les caractéristiques, comme par exemple le temps d'arriver à une situation finale, que la solution soit simple à comprendre ou qu'elle puisse évoluer, sont des caractéristiques non-fonctionnelles.

3. Modèles et programmes

Jusqu'à maintenant, les définitions de monde, problème et solution sont générales, et peuvent s'interpréter sur plusieurs domaines. Lors de l'utilisation d'une machine pour résoudre un problème, nous entrons dans le domaine de l'informatique et de la modélisation.

Pour résoudre un problème avec une machine, il faut d'abord représenter le monde du problème dans la machine, et, ensuite, traduire la solution en termes des éléments d'un

langage de programmation, de telle façon que l'exécution du programme ainsi généré puisse nous amener à une représentation du monde où le problème a été résolu.

Un modèle d'un monde M est une abstraction de M qui contient tous les éléments qui participent au problème, et qui se comporte comme le monde qu'il est censé représenter. Ceci implique que le modèle et le monde réagissent de la même manière aux stimulus externes, et que le modèle a la capacité de représenter tous les états possibles de l'abstraction du monde.

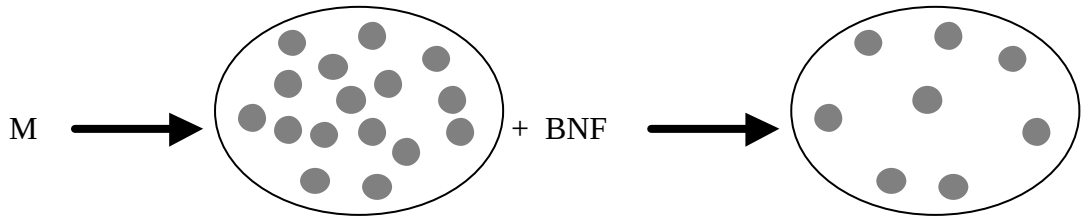
Un programme PR est une implémentation d'une solution pour résoudre un problème P sur un modèle donné de M.

PR(P) = < MD, S(P, M) >

où :

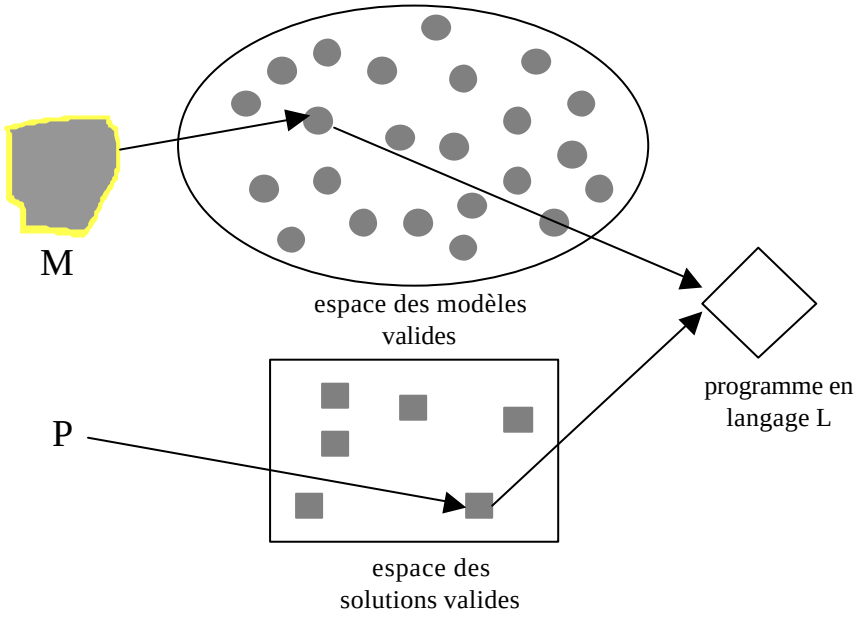
MD est un modèle de M (le monde du problème P)
S (P, M) est une implémentation d'une solution de P sur M

Il faut noter que pour un même monde, il y a plusieurs modèles possibles, chacun avec des caractéristiques non-fonctionnelles différentes. Ces CNF sont surtout liées à des aspects tels que extensibilité, couplage et cohésion [YC79]. Les BNF imposent aussi des conditions sur les CNF du modèle, et restreindrent ainsi les modèles valides.



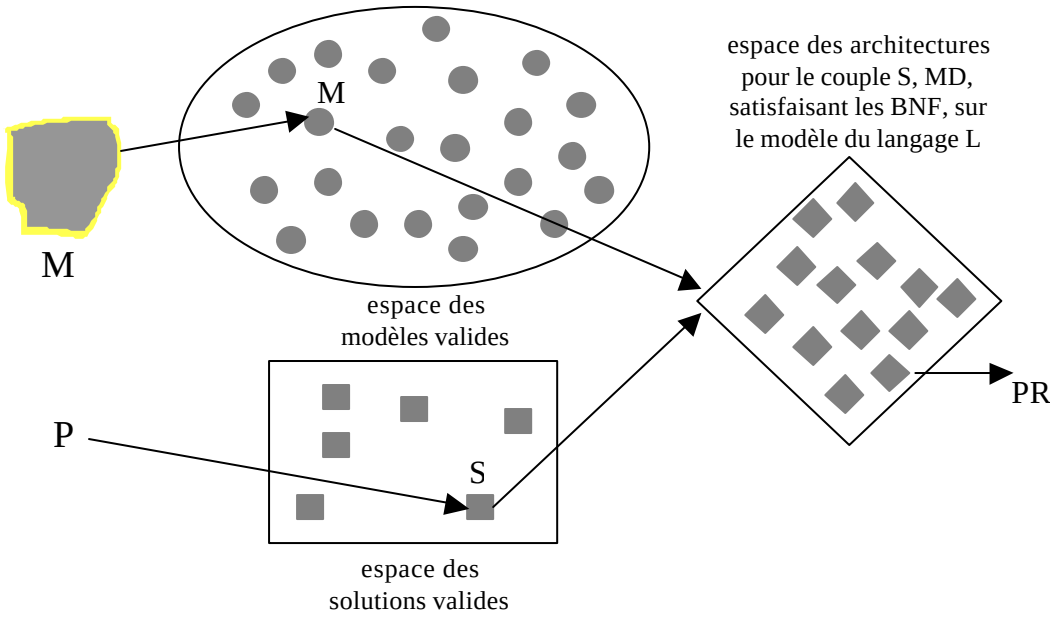
Il est important de remarquer que les caractéristiques non-fonctionnelles d'un programme sont liées autant à la modélisation qu'à la solution, et que la plupart des fois il y a un fort compromis entre les deux. Il y a toujours, par exemple, un compromis de conception entre espace de mémoire occupé et performance. S'il y a une contrainte de performance, il peut falloir changer de modèle pour pouvoir la satisfaire. En particulier, il est possible que pour un ensemble donné de BNF, il n'existe pas de programme pour résoudre un problème, même s'il y a des modèles et des solutions imaginables.

En gros, on peut dire que la tâche d'un programmeur consiste à choisir une solution de l'espace des solutions valides du problème et un modèle de l'espace de modélisations valides du monde. Ensuite, le programmeur doit traduire les deux dans un langage de programmation L, pour les exprimer ensemble, en termes des éléments et des structures disponibles de L (structures de données et procédures, si L est un langage impératif).



D'un point de vue structurel, le programme résultat est simplement un ensemble d'éléments du modèle du langage, connectés d'une certaine manière entre eux (qui appelle qui, qui a un paramètre d'un type de donnée, quelle donnée a une référence vers quelle donnée, etc.). Le graphe (ou l'ensemble de graphes) modélisant cette topologie, est connu sous le nom d'architecture du programme.

Avec cette vision structurelle, il est possible de dire que pour tout couple modèle - solution, il existe un espace d'architectures imaginables (chacune avec ses propres CNF), chacune représentant un programme. De cette façon, le processus de traduction vers un langage de programmation peut être vu comme la sélection d'une architecture dans le sous-espace où les programmes correspondants satisfont les BNF.



4. Le modèle d'objets

Dans cette partie nous présentons le modèle d'objets, par rapport aux concepts définis dans les sections précédentes, et nous montrons les principales différences avec les modèles classiques et structurés.

De façon informelle, on peut dire que le modèle d'objets fait référence à deux choses : d'une part, il définit les éléments du modèle du langage (i.e. les éléments de l'architecture du programme), et d'autre part, il guide la manière de choisir une architecture dans l'espace des architectures possibles. Nous commençons par la deuxième partie, en comparant l'architecture obtenue avec les modèles classiques, structurés et d'objets.

Ce que nous appelons l'approche classique correspond à la vision de développement d'un logiciel antérieure aux méthodologies structurées [RS77, YC79]. Cette approche peut être résumée dans les termes suivants :

- on peut choisir n'importe quel modèle du monde,
- on peut choisir n'importe quelle architecture, à condition que le programme résultant satisfasse les BNF

Pour des programmes de petite taille et peu évolutifs, cette approche peut donner de bons résultats, mais son utilisation a pour conséquence que l'on ne peut rien dire à propos de la localisation des éléments du modèle du monde dans l'architecture, ni comment la solution du problème et le comportement du monde ont été mélangés dans l'ensemble des procédures constituant le programme. En résumé, on a perdu toute trace sémantique dans l'architecture, et il faut se contenter des aspects purement syntaxiques de la topologie du programme.

Dans l'approche structurée, née dans les années 70 comme une réponse à "la crise du logiciel", on s'appuie sur une méthodologie de construction de programmes [YC79] inspirée par le principe de "diviser et conquérir". L'architecture est donc guidée par la décomposition du problème en sous-problèmes, ce qui produit un programme constitué d'une hiérarchie de modules. Ceci a plusieurs conséquences :

Avantages :
- l'espace des architectures possibles est réduit,
- les architectures possibles ont des caractéristiques communes, ce qui facilite la création d'outils et la définition de formalismes et de critères de qualité,
- il y a une relation claire entre la solution et l'architecture, car l'architecture correspond à la décomposition de la solution en sous-solutions.

Inconvénients :
- on a perdu toute trace du modèle du monde, son comportement est quelque part dans l'architecture (typiquement éparpillé dans la solution) et il est impossible de généraliser la façon de le localiser,
- les modèles structurés supportent très mal les évolutions, en particulier, les évolutions du monde du problème,

- il est impossible de proposer une méthodologie générale de maintenance d'un programme ou de guider son évolution, et on doit se contenter de quelques conseils et recettes génériques.

Il est certain que c'est un avancement par rapport à ce qu'on avait avant, mais il est aussi clair qu'on ne peut pas toujours remonter le niveau d'abstraction d'un programme et réfléchir au niveau du monde du problème : la double traduction (monde ? modèle ? architecture) n'est pas réversible.

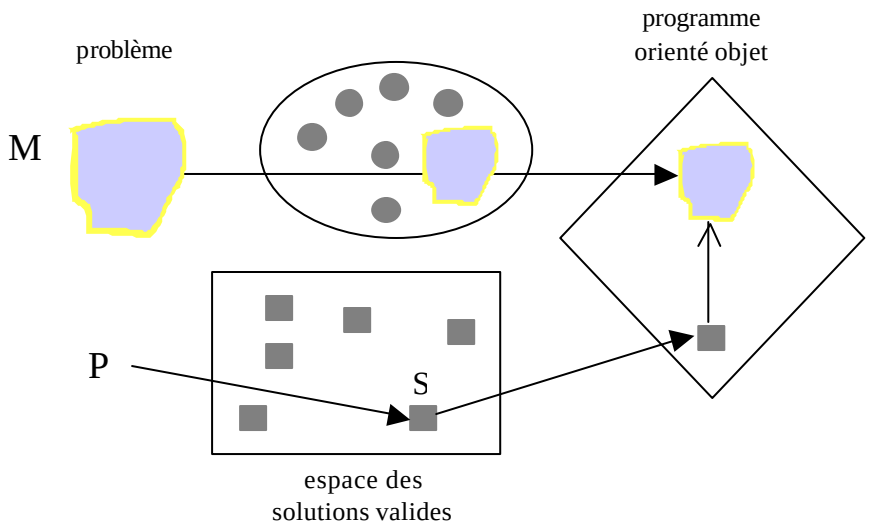
L'idée du modèle d'objets est d'une simplicité étonnante, et c'est peut être pour cela qu'elle est parfois mal interprétée ou banalisée. Tout est basé sur l'idée des modèles de simulation, et elle indique comment choisir le modèle du monde, et comment sélectionner l'architecture de l'espace des architectures possibles.

Dans l'approche objet, le modèle qui doit être sélectionné, parmi tous les modèles possibles, correspond au modèle isomorphe au monde du problème. Cela veut dire, que c'est un modèle où on trouve une correspondance un à un entre les éléments du modèle et les éléments du monde participant au problème.

D'autre part, il faut choisir une architecture de l'espace d'architectures possibles dans le sous-espace défini par les contraintes suivantes :

- le modèle et la solution sont séparés,
- il y a une traduction directe entre le modèle du monde et les concepts du modèle d'objets, ce qui veut dire que le programme comporte un modèle opérationnel du monde, isomorphe au monde et capable de simuler son fonctionnement,
- la solution n'inclut que l'ensemble des stimulus dont on parlait dans les sections initiales du chapitre, traduits en termes d'interactions avec le modèle opérationnel du point précédent.

Graphiquement, il est possible de résumer ces idées avec la figure suivante :



De tous les avantages de cette approche, le plus important c'est qu'il y a un lien sémantique et syntaxique entre le problème et le programme, parce qu'on a évité les pertes d'information dans la double traduction du processus de développement (modélisation et traduction dans un langage) et tout mélange entre le modèle et la solution. Ceci a trois conséquences directes :

- il est possible proposer des méthodes pour gérer l'évolution d'un programme, ce qui permet l'introduction du cycle de vie en spirale lié à la programmation orientée objet,
- la taille d'une modification dans un programme est proportionnelle à la taille de la modification du problème (une propriété très importante du point de vue calcul d'impact),
- il est envisageable de remonter de niveau d'abstraction et de réfléchir au niveau du monde du problème, étant donné qu'on a beaucoup plus que le simple graphe de connexions des éléments du programme.

Il est à noter que, dans le modèle d'objets, il n'y a pas de structure définie pour exprimer la solution du problème (on la met toujours dans le `main` du programme). Si la solution est complexe, sa structure pourrait ne pas être triviale, et on ne saurait pas comment la structurer pour faciliter son évolution. On verra vers la fin du chapitre, qu'une partie de cette responsabilité sera assignée à ce qu'on va appeler le contrôle de l'application.

Nous définissons un programme orienté objet, comme un programme avec une architecture d'objets, dont le modèle opérationnel du monde est isomorphe au monde du problème.

Le reste des propriétés, des avantages et des inconvénients de chaque approche ont déjà été répertoriés par plusieurs travaux [Mey97, Boo94], et c'est hors de la portée de cette thèse.

En ce qui concerne les éléments du modèle d'objets, il comporte cinq concepts : objet, classe, attribut, méthode et message [Mey97], et un mécanisme, appelé l'héritage, pour simplifier l'expression d'une architecture, très apprécié dans les années 80 pour sa capacité de factoriser et réutiliser la définition d'un concept, mais de plus en plus critiqué à cause des problèmes de maintenance et d'évolution que ceci engendre [GHJ95].

Finalement, nous introduisons le terme "invariant" d'une classe comme l'ensemble des contraintes et des relations entre les attributs d'une classe pour représenter (modéliser) un élément du monde du problème. Les invariants correspondent aux règles de cohérence et de validité du monde du problème.

5. Cycle de vie et objets

Bien que dans la section 4 nous ayons présenté les idées générales du modèle d'objets, nous avons évité de donner une définition du concept d'objet, car il y a un facteur que nous n'avons pas considéré et que nous introduisons dans cette partie : le cycle de vie.

Le cycle de vie de développement d'un programme orienté objet se base sur l'idée du raffinement des mêmes concepts, en évitant un cycle avec des étapes très marquées avec des concepts et formalismes différents. Ceci implique que nous allons retrouver les mêmes concepts tout au long du cycle de vie mais à différents niveaux d'abstraction. Le terme objet, par exemple, va représenter des choses distinctes à chaque étape.

Bien que l'idée de baser le processus de développement sur le raffinement du même ensemble de concepts ait énormément d'avantages, il est vrai aussi que cela complique les définitions et les comparaisons entre les concepts de modèles différents, comme c'est le cas avec le concept de composant, abordé dans le chapitre 3 de ce document.

Nous montrerons dans la suite, de manière très schématique, les étapes du cycle de vie de développement, et, pour chacune des étapes, les éléments, les définitions et les concepts les plus importants. Cette vision de cycle de vie correspond à une approche tout à fait différente de celle utilisée dans la section précédente, puisque nous nous intéressons maintenant à la problématique de la construction d'une architecture d'objets. La différence est que l'espace des architectures possibles existe dans la théorie, mais pas dans la pratique, et que, d'un point de vue méthodologique, il faut définir la manière d'arriver à une architecture par construction à partir du problème, et non seulement d'en choisir une.

5.1. L'analyse objet

Objectif : identifier le problème (P), modéliser le monde du problème (M) et définir les BNF.

Dans cette étape, un objet est une abstraction d'un élément de M et il y a une relation un à un entre les objets du modèle et les éléments de M.

5.2. La conception objet

Objectif : raffiner le modèle du monde (MD) et écrire une solution S de P pour qu'elle puisse être implémentée. L'ensemble doit satisfaire les BNF.

Cette étape est l'étape la plus compliquée de tout le cycle de vie, car il n'est pas clair (du point de vue méthodologique) ce que veut dire raffiner le modèle du monde et comment mesurer si le résultat satisfait les BNF. Le but, en tout cas, est de maintenir le modèle du monde aussi proche que possible du modèle obtenu dans l'analyse (dit le modèle idéal), mais en satisfaisant les BNF. Plus on s'éloigne du modèle idéal, plus difficile va être l'évolution. Dans l'optique de l'espace des architectures possibles, et si on pouvait parler d'une continuité de l'espace, on dirait qu'il faut chercher une architecture voisine de l'architecture idéale, mais avec les propriétés non fonctionnelles imposées par les BNF.

A la fin de cette étape, et d'un point de vue purement structurel, un objet est un ensemble d'attributs et de méthodes. En général, il correspond simplement à une concrétisation d'un objet (abstrait) de l'étape précédente. Au niveau du modèle, on peut dire que la sortie de la phase de conception est presque une réification du modèle de l'analyse.

Il est important de noter, qu'on ne peut plus garantir une stricte relation un à un entre les objets et les éléments de M. Soit, il y a des objets qui viennent d'apparaître (en réponse à un BNF), soit, il y a des objets qui ont disparu pour la même raison (e.g. pour des raisons de performance ou de complexité). Nous nommons les objets qui naissent dans l'étape de conception les faux objets du modèle, et les objets qui disparaissent dans cette même étape, les objets morts.

Un objet peut disparaître pour des raisons d'espace de mémoire ou de performance. Par exemple, dans le cas d'un éditeur de texte, bien qu'on puisse considérer chaque lettre du texte comme un objet, du point de vue implémentation ceci n'est pas réaliste, et il faut adapter le modèle avec une représentation différente pour les lignes du texte.

On peut trouver des exemples de faux objets dans l'application de la plupart des patrons de conception. Le fait d'ajouter une façade [GHJ95] ou un médiateur [GHJ95] dans un modèle, pour diminuer le couplage et simplifier sa communication interne, implique nécessairement l'apparition de faux objets. De même pour les techniques de diminution de temps de réponse, où l'on rajoute au modèle des objets pour stocker des informations pré-calculées ou redondantes.

Après l'architecture du modèle du monde, le problème le plus délicat est l'assignation de responsabilités aux objets. La capacité d'évolution du modèle est profondément liée à la manière dont les méthodes de chaque classe ont été conçues. Larman [Lar01] propose une liste de patrons d'assignation de responsabilités (GRASP) pour guider ce processus. La tendance à ajouter une méthode `getXX` et une méthode `setXX` pour chaque attribut `XX` de la classe, est l'un des pires problèmes au moment de l'évolution.

Dans l'exemple des deux mobiles, si dans la classe `Mobile` on définit, par exemple, une méthode `setPosition`, on introduit deux problèmes : d'une part, on permet aux clients de la classe de contrôler une caractéristique qui n'est pas indépendante, et qui ne peut pas être contrôlée directement de l'extérieur. D'autre part, le fait de laisser une partie du comportement du monde dehors de la classe responsable, va rendre difficile son évolution.

5.3. L'implémentation objet

Objectif : traduire le modèle et la solution dans un langage de programmation. Si c'est un langage objet, en général cette traduction est un processus mécanique direct.

Pour des langages comme C++ [Str97], qui ne sont pas des langages d'objet purs, il faut souvent remplacer les objets par des pointeurs vers les objets, gérer à la main le cycle de vie des objets (destructeurs), etc.

5.4. L'exécution objet

Objectif : exécuter un programme objet.

Dans cette étape il y a plusieurs situations possibles, qui dépendent des propriétés du langage choisi.

D'abord nous avons le cas dans lequel l'objet n'existe plus pendant l'exécution, puisqu'il a été traduit par un ensemble d'artefacts qui simulent son comportement. C'est le cas de tous les langages objets "non purs" (comme C++, par exemple). Ces langages traduisent les objets en termes des éléments d'un autre modèle (e.g. le modèle impératif ou fonctionnel). Typiquement, un objet C++ est traduit comme un pointeur à une structure de données du langage C.

Dans le second cas, l'objet existe pendant l'exécution, mais il a perdu toutes les méta-informations concernant son modèle. Ceci veut dire qu'il peut s'exécuter, mais qu'il ne peut pas agir en réfléchissant sur sa propre structure.

Dans le troisième cas, on trouve les modèles d'exécution avec introspection (comme Java [GJS00], par exemple), qui permettent à un objet de récupérer à l'exécution les méta-informations qui le concernent (sous forme d'objets), et d'agir en conséquence. Les objets qui représentent la méta-information sont des objets qui apparaissent uniquement à l'exécution (c'est le cas des classes `Class` ou `Method` de java). Nous allons appeler ces objets les méta-objets, pour les distinguer des objets du programme.

Nous avons aussi les modèles d'exécution réflexifs (comme Smalltalk [GR85], par exemple), qui permettent à un objet de récupérer et de modifier à l'exécution ses méta-informations. Dans ce type de modèles, on peut déclarer une nouvelle méthode pendant l'exécution, créer une nouvelle classe, etc. Ce type de modèle est basé aussi sur des méta-objets.

Nous définissons un objet comme un élément appartenant au modèle du monde du problème, que nous retrouvons à chaque étape du cycle de vie de développement, y compris pendant la phase d'exécution.

Il est important de souligner que d'après notre définition de programme orienté objet, seulement un sous-ensemble de tous les programmes que nous pouvons écrire dans un langage objet, sont considérés orientés objet, car l'important c'est le fait d'être supporté par un modèle d'objets, et pas seulement le langage de programmation utilisé pour écrire le programme. De la même façon, nous pouvons affirmer que tous les groupements de données et de procédures ne forment pas un objet.

Dans la littérature on retrouve souvent le terme objet métier (*business object*) [Per00] pour distinguer un objet du monde du problème des autres interprétations possibles du terme, et le terme "*business model*" pour faire référence au modèle du monde du problème.

Avant de terminer cette section, et pour profiter du contexte défini, nous voulons introduire le concept d'objet du modèle COM [Box98, COM], puisque c'est un type d'objet qui n'existe que pendant l'exécution. En simplifiant un peu, on peut dire qu'un objet COM est une structure de données avec un ensemble de routines associées (ce qui, vu de loin, ressemble à la structure d'un objet), qui a été compilé d'une façon particulière pour générer, pendant l'exécution, un tableau de pointeurs à routines qui satisfait les normes exigées par le protocole COM. Ceci permet d'utiliser les procédures d'un autre objet, même s'il est écrit dans un langage différent. Rien n'empêche qu'un objet COM corresponde à un objet d'un modèle, mais ce n'est pas indispensable.

6. Applications et les mondes de la solution

Dans les sections précédentes, nous avons montré un programme orienté objet d'une manière très simpliste, pour faciliter la présentation des divers concepts, mais il est clair qu'il manque toute la partie concernant l'utilisateur et, en général, tous les aspects qui font partie des applications interactives.

6.1. Applications interactives simples

Normalement, une application interactive n'est pas faite pour résoudre un problème, mais pour permettre à une personne d'interagir, d'échanger des informations, et de résoudre plusieurs problèmes partageant le même monde. Dans ce cas, ce type d'application peut être considéré comme un cadre de travail pour un client, qui, de façon interactive, utilise l'application pour appliquer les solutions aux différents problèmes qu'elle peut résoudre.

Une application interactive A est une généralisation du concept de programme, capable de résoudre un ensemble de problèmes sous le pilotage d'un usager.

$A (P_1, \ldots, P_n) = < MD, \{ S(P_i, MD) \}, C, V >$

où :

$P_1,\ldots, P_n$ est un ensemble de problèmes qui partagent le même monde M
MD est un modèle de M
$S(P_i, MD)$ est une implémentation d'une solution de P_i sur MD
C : est le contrôle de l'application (expliqué dans la suite)
V : est la visualisation de l'application (expliqué dans la suite)

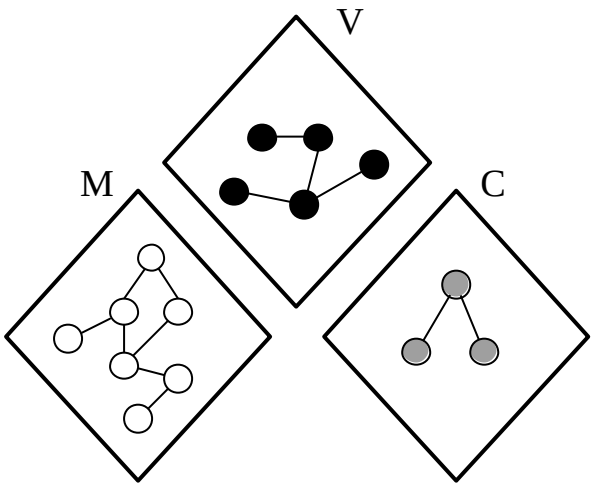
Le contrôle C est chargé de gérer le dialogue avec l'utilisateur, pour saisir les entrées et définir l'ordre d'application des solutions. Il garantit que le modèle du monde satisfait la précondition du problème pour que la solution correspondante s'exécute. Il y a plusieurs formalismes possibles pour modéliser et exprimer le contrôle d'une application [Vil86].

La visualisation V montre à l'usager le résultat des solutions des problèmes et l'état du modèle.

6.2. Une vision multi-monde d'une application

Nous avons introduit le concept d'application dans le chapitre des objets afin de montrer comment, en plus du monde du problème, il y a toujours plusieurs mondes concernés dans un logiciel. Ces mondes doivent vivre ensemble et se synchroniser, malgré le fait que chacun d'eux comporte un ensemble disjoint de concepts et de comportements.

Si on veut utiliser une approche objet pour écrire une application interactive, telle qu'elle a été définie dans la section au-dessus, on se rend compte qu'au lieu d'un seul monde il y en a trois. Dans le monde de la visualisation, par exemple, on retrouve des éléments comme les fenêtres, les boutons, les menus, les lignes, etc., avec leurs propriétés et leur comportement, complètement indépendant du monde d'un problème spécifique. La même situation est vraie pour le monde du contrôle de dialogue, où on retrouve des éléments comme les états de dialogue, les options, les transitions, etc., liés à une mécanique de fonctionnement.



La question à laquelle il faut pouvoir répondre est comment faire inter-opérer ces trois mondes qui sont à la fois disjoints (ils ne partagent pas d'éléments) et sémantiquement dépendants (il y a une seule exécution commune de l'application où ils doivent se synchroniser) ?

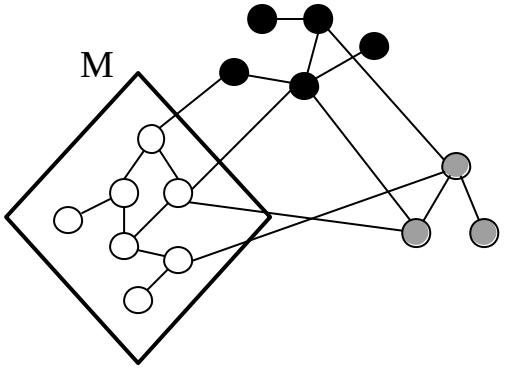
Est-ce qu'il y a un mécanisme de composition de mondes dans le modèle d'objets ?

La réponse à la question est non. Simplement, dans la nature même du modèle d'objets il n'existe pas le concept de plusieurs mondes simultanés, ni une façon de les composer. Que faire alors ? Nous sommes obligés dans ce cas de "remonter" de niveau et d'aller chercher des modèles de coordination de mondes, et ceci fait partie du domaine de l'architecture logicielle plus que du domaine des objets.

6.3. Une solution simple au problème des mondes M-V-C

La solution la plus simple au problème, et la plus fréquemment utilisée, consiste à mélanger les modèles des trois mondes. Dans ce cas, on crée des dépendances très profondes entre les modèles (les objets d'un modèle ont des références vers les objets des autres modèles, et ils les utilisent pour faire des appels sur leurs méthodes) et on perd ainsi les propriétés individuelles de chaque modèle d'objets, en particulier l'extensibilité.

Typiquement, on ajoute dans l'un des modèles les objets des autres mondes, et on passe à ses méthodes la responsabilité de gérer la cohérence entre les objets originaux du modèle et les objets connectés, polluant de cette façon le modèle de base.



Cette solution est pratique pour des petits projets, mais inintéressante à tout point de vue.

6.4. L'architecture MVC

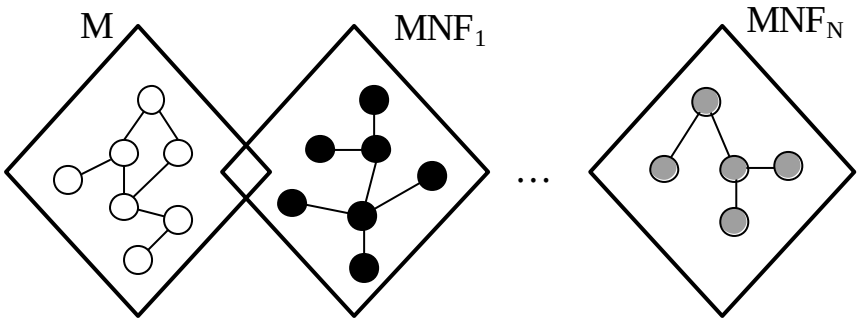
Avec les premières implémentations du langage Smalltalk, il y a plus de 20 ans, on trouve une solution de niveau architecture pour implémenter les applications graphiques interactives avec trois modèles d'objets indépendants, sans perdre ses propriétés individuelles d'évolution.

L'architecture modèle-vue-contrôle [BMR96] utilisée actuellement dans la plupart des modèles graphiques (par exemple, dans le *framework* swing de java [SWING]), propose une connexion faible entre les mondes, en se basant sur le pattern observer [GHJ95] et un système d'événements, permettant la synchronisation entre les objets des différents mondes concernés. Cette architecture est une bonne solution au problème, mais elle n'est pas généralisable pour tous les mondes de la solution, tel qu'on va le montrer plus loin.

L'idée de base de l'architecture MVC consiste à permettre aux objets d'un monde d'être observés. Cela veut dire que ces objets savent qu'il y a des observateurs (peut-être dans un monde différent) qui sont intéressés par son état, et qu'il doit leur notifier chaque modification de l'état de l'objet. De cette manière, les mondes peuvent se synchroniser, sans générer des dépendances directes. L'architecture MVC définit un protocole de synchronisation entre le monde du problème, le contrôle de l'application et sa visualisation, en utilisant un système de souscriptions (se déclarer comme observateur de quelqu'un) et de notifications (avertir un observateur d'un changement d'état).

6.5. Le cas général de plusieurs mondes

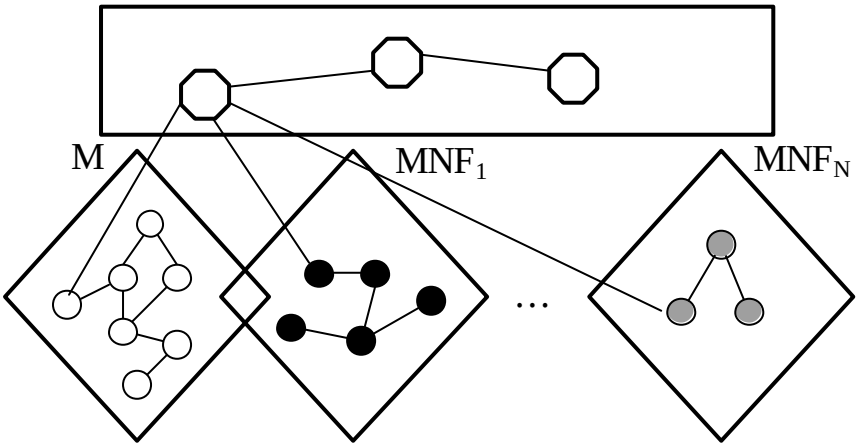
Dans le cas général, une application a un monde du problème et N mondes de la solution (les mondes non-fonctionnels), chacun avec un modèle propre.



Comme exemples typiques de mondes non-fonctionnels, nous avons la persistance, la distribution, la sécurité, etc., en plus des mondes de la visualisation et du contrôle. Dans la plupart des cas, ce type de mondes correspond à des mondes connus, où l'on peut déjà trouver des modèles et des implémentations, sous forme de descriptions fonctionnelles (JDBC [WFC99, REE00], JAAS [GON99], JNDI [LS00]), de *frameworks* (swing [WC00], RMI [GRO01, PM01]), d'architectures (CORBA [SIE96, RM97]), de connecteurs et d'outils, etc.

Les BF caractérisent les solutions du monde du problème, et les BNF définissent les contraintes sur chaque monde non fonctionnel et sur les relations entre les mondes.

Actuellement, la question de fond est de savoir s'il y a une façon générale de coordonner le monde du problème et les mondes non fonctionnels, en conservant leurs propriétés individuelles d'évolution. Si la réponse est affirmative, il faudrait définir une sorte de monde pour la synchronisation, au-dessus des autres mondes, avec un ensemble d'artefacts et de mécanismes pour synchroniser et pour faire coopérer les objets des différents mondes.



Ceci nous permettrait de modéliser une application de la façon suivante :

A (P_1, …, P_n) = < MD, { S(P_i, MD) }, { MNF }, CD >

où :

P_1,…, P_n est un ensemble de problèmes qui partagent le même monde M
MD est un modèle de M
S(P_i, MD) est une implémentation d'une solution de P_i sur MD
MNF est un monde non fonctionnel
CD est un modèle de coordination entre M et les mondes non fonctionnels

Dans les chapitres suivants, nous analysons les différentes approches proposées pour résoudre les diverses parties du problème, et nous définissons plus précisément les objectifs directs de notre travail et nos propositions.

Dans le reste du chapitre, nous faisons le point sur les limitations du modèle à objets, comme une manière de définir une base de discussion et de comparaison entre les différentes approches.

7. Limitations du modèle d'objets

Dans cette partie nous indiquerons les trois limitations que nous considérons les plus importantes du modèle à objets, et qui n'ont pas une solution évidente dans le cadre du modèle lui-même. Ces limitations vont motiver l'introduction des concepts et des mécanismes présentés dans les chapitres qui suivent.

7.1. La représentation de plusieurs mondes simultanés

Le modèle à objets a été proposé comme une solution au problème de l'évolution des logiciels, à une époque (années 70) où la problématique était différente de la problématique actuelle. A cette époque, les BNF étaient très peu importants par rapport aux BF, et proposer de guider l'architecture d'une application par la structure du monde du problème était raisonnable et suffisant. L'utilisation d'une architecture comme MVC permettait de réaliser, de manière élégante, une application graphique interactive ; ce qui représentait les BNF les plus sophistiqués des années 70 et 80.

La difficulté est que la problématique a beaucoup changé au cours des vingt dernières années. On s'est rendu compte que le modèle d'objets ne peut pas être étendu pour représenter plusieurs mondes à la fois, et qu'il n'est pas possible de généraliser l'architecture MVC pour considérer tout type de monde non-fonctionnel.

La solution n'est pas de proposer des extensions au modèle, parce que cela provoquerait rapidement des conflits avec les idées de base des objets. La solution est donc de passer par un changement de niveau d'abstraction, et de chercher des réponses dans le domaine des architectures logicielles, en utilisant éventuellement le modèle d'objets comme l'élément de base. Si on n'a pas encore de propositions concrètes, c'est parce qu'après un démarrage très prometteur du thème des architectures logicielles, on se retrouve actuellement face à un manque décevant de résultats utilisables.

7.2. Les problèmes de la réutilisation

La réutilisation est l'un des chevaux de bataille de la programmation orientée objet [Mey97, Mey99c]. Bien que cette propriété soit une réalité à certains niveaux (les objets implémentant les structures de données, les objets des modèles graphiques, les *frameworks* de communications, etc.), on se rend compte que ce n'est pas le cas général, et qu'on n'arrive pas à des choses réutilisables par hasard [Mey96, Fin98].

D'abord, on a les problèmes méthodologiques et d'organisation, qui font de la réutilisation un objectif difficile à atteindre dans la pratique.

Mais le problème le plus grave, que depuis quelques années on commence à constater, est que la réutilisation ne se réduit pas à un bon usage des mécanismes tel que l'héritage ou la généricité. Le problème est beaucoup plus profond. Même dans le cas d'une application qui n'évolue pas du point de vue fonctionnel (i.e. le monde du problème et le problème sont constants), on trouve une évolution non-fonctionnelle permanente que le modèle d'objets n'est pas préparé à supporter. Ces évolutions correspondent, dans la plupart des cas, à des changements de la technologie : une évolution du système d'exploitation, un changement de langage, une nouvelle version du compilateur ou d'un protocole, etc. Ce type particulier d'évolution est connu sous le nom d'adaptation. La raison de cette difficulté est que le modèle d'objets est un modèle abstrait, qui ne considère pas les aspects concrets de son implémentation, et qui a laissé les outils et les langages de programmation définir leurs propres règles de jeu : un objet ne peut se communiquer avec un autre objet que s'il est exprimé dans le même langage, la même version, et s'exécute sur la même machine dans le même processus. Toute autre situation engendre des problèmes technologiques qu'on ne sait pas comment exprimer et isoler dans un modèle d'objets pour

faciliter leur évolution (e.g. *threads*, *sockets*, processus, sérialisation, synchronisation, etc.).

En résumé, on peut dire que les propriétés d'évolution du modèle d'objets n'existent que dans un monde abstrait, et qu'au moment de concrétiser les concepts sur une technologie précise, il est impossible de garantir une propriété quelconque dans l'application résultante.

7.3. L'assemblage comme mécanisme de programmation

Dans de nombreuses publications des années 80, on pouvait lire qu'avec les objets on allait changer profondément la façon de développer un logiciel, puisqu'il suffirait d'acheter les objets nécessaires et de les assembler pour obtenir l'application. Mais le modèle d'objets n'a pas de mécanismes d'assemblage à part l'héritage et la composition, tous les deux agissant lors de la compilation, et ils sont clairement insuffisants si on veut séparer la programmation et l'assemblage pour en faire deux étapes différentes du développement d'une application.

D'abord, il faudrait pouvoir spécifier un objet de manière indépendante de son implémentation. C'est pour cette raison que le concept d'interface a été introduit dans le modèle d'objets dans les années 90. Ensuite, il faudrait disposer de langages et de mécanismes de haut niveau, permettant de décrire l'application en termes de la spécification des objets participants et de ses connections. Le modèle d'objets n'a pas ce type de facilités.

8. Conclusions et résumé

Les principales conclusions qu'on peut tirer de ce chapitre sont les suivantes :

- Le modèle d'objets ne correspond pas à la vision simpliste et banale, présentée souvent dans la littérature, qui le réduit à une liste de propriétés syntaxiques sans une architecture ou un esprit derrière.
- Le modèle d'objets a des propriétés théoriques très importantes, qu'on ne trouve pas dans les autres modèles, en particulier en ce qui concerne l'évolution.
- Le modèle d'objets a des contraintes pour (1) représenter et gérer multiples mondes simultanés, (2) représenter et gérer les caractéristiques non-fonctionnelles d'une application sans les mélanger avec les objets, ce qui rend difficile son adaptation et (3) utiliser l'assemblage comme un vrai mécanisme de programmation.

Dans ce chapitre nous avons introduit plusieurs définitions, que nous résumons dans la suite, pour faciliter la lecture du reste du document :

- Monde : Un monde M est un système fermé, composé d'un ensemble d'éléments, un ensemble de relations entre ces éléments et un ensemble de règles de fonctionnement (dit son comportement) qui lui permettent d'évoluer dans le temps.

- Objet : Un objet O est un élément appartenant au modèle du monde du problème, que nous retrouvons à chaque étape du cycle de vie de développement, y compris pendant la phase d'exécution.

- Besoin fonctionnel (BF) : Un BF est une fonction qui définit un problème P dans un monde.

- Solution : Une solution S d'un problème P est un algorithme qui fait passer le monde du problème d'une situation initiale de P à une situation finale de P. Un problème peut avoir zéro ou plusieurs solutions.

- Caractéristique non-fonctionnelle CNF : Toute caractéristique permettant de différencier deux solutions d'un même problème P est appelée une CNF. Actuellement, on retrouve aussi dans la littérature le terme caractéristique extra-fonctionnelle pour faire référence au même concept. Bien qu'on n'ait pas fait une classification des CNF, il est clair qu'il y en a plusieurs types différents et qu'il devrait être possible de trouver une caractérisation de chaque groupe.

- Besoin non-fonctionnel (BNF) : Un BNF définit un ensemble de caractéristiques non-fonctionnelles exigées à une application.

Chapitre III

Composants

1. Introduction

Ce chapitre étudie les solutions basées sur le concept de composant concernant les limitations des objets. Les objectifs sont de proposer une caractérisation de haut niveau des modèles de composants, de montrer toutes les possibilités disponibles dans l'espace conceptuel défini par cette caractérisation, de classer les modèles commerciaux actuels par rapport à cet espace, et de montrer les limites de l'approche.

Nous proposons d'abord une classification des composants en trois groupes : d'une part, (1) les composants dont l'objectif est de faciliter l'évolution d'un monde, et qui peuvent être considérés comme une extension du modèle d'objets. D'autre part, (2) les composants du monde de la coordination, introduit dans le chapitre précédent, dont le but est d'exprimer les relations entre les divers mondes d'une application, et de leur permettre de se synchroniser sans compromettre leur indépendance. Et finalement, (3) les composants produits de l'évolution du concept de module de la programmation structurée, et qui représentent une capacité de calcul, exprimée en termes d'un ensemble de services.

Nous utiliserons le terme "composant de composition", pour les composants du premier groupe (présentés dans la section 2), le terme "composant de coordination" pour les composants du deuxième groupe (présentés dans la section 3), et le terme "composant fonctionnel" pour ceux du troisième groupe (présentés dans la section 4). Bien que chaque groupe soit destiné à résoudre une problématique différente, et que chacun compte sur un ensemble de concepts et de mécanismes distincts pour répondre à ses besoins, ils sont presque toujours présentés dans la littérature comme une seule et même chose. Il faut remarquer aussi, que quelques modèles commerciaux de composants supportent les trois types de composants à la fois.

Dans le chapitre 4, nous présenterons et classerons les approches "non-composants" au problème d'évolution du modèle d'objets et au problème des mondes multiples dans les applications, et nous essaierons de caractériser leurs capacités et leurs limitations intrinsèques.

Il nous semble important d'aborder le thème des composants de cette façon, car, actuellement, le rôle des composants dans l'architecture d'une application n'est pas clair, et ceci rend difficile la comparaison avec d'autres solutions possibles, et même entre deux modèles de composants différents. Nous pensons qu'il y a une tendance actuellement à rajouter aux modèles de composants tout ce qui peut sembler utile, en mélangeant les aspects technologiques, méthodologiques et conceptuels. Il ne faut pas oublier que les composants surgissent du milieu industriel comme une réponse pragmatique au manque de

résultats utilisables des milieux académiques aux problèmes d'inclusion et évolution des BNF dans des applications non triviales. Cela a comme conséquence l'absence d'un cadre conceptuel commun, et une forte liaison avec des produits et des technologies spécifiques.

2. Composants comme éléments d'assemblage

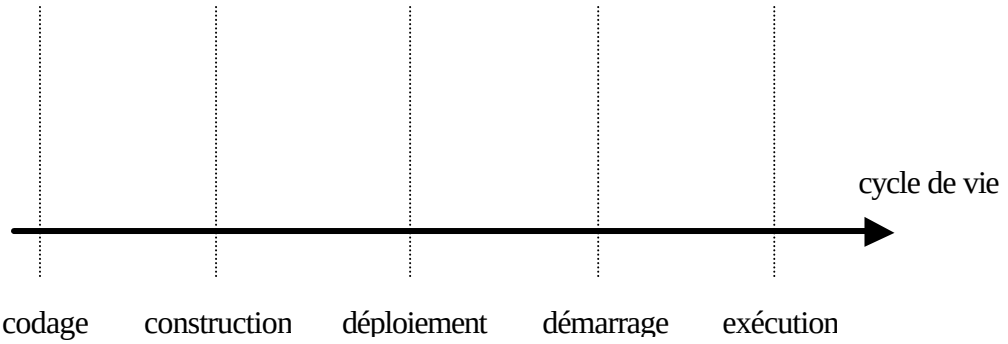
Le premier groupe de composants, dont on parlait dans l'introduction, sont les composants de composition, utilisés comme moyen pour résoudre les problèmes d'évolution des BNF et les limitations dans la réutilisation des objets. Par simplicité, dans cette section, le terme composant sera utilisé uniquement pour les composants de composition.

Ces composants et les mécanismes associés peuvent être considérés comme une extension du modèle d'objets, car son seul but est de permettre une meilleure expression du même modèle, de telle façon qu'il soit possible d'adapter et de faire évoluer l'application plus aisément. Ces composants s'utilisent autant dans le monde du problème, que dans les mondes non-fonctionnels, mais pour un seul monde à la fois. Dans les mondes non-fonctionnels, les composants sont surtout utilisés pour découpler la structure du monde d'une technologie spécifique.

2.1. La solution proposée

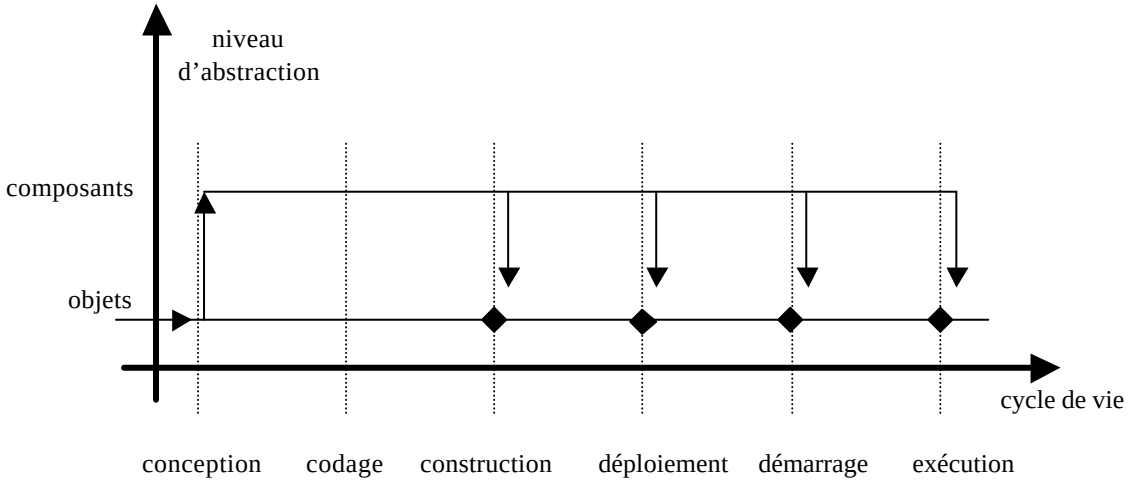
Les problèmes d'évolution des objets sont fortement liés au moment de prendre une décision dans le cycle de vie de développement d'une application. Plus on retarde le choix d'une caractéristique, plus il sera facile de la changer. Avec le modèle d'objets toutes les décisions de niveau architecture sont prises avant ou durant l'étape du codage, parce que les seuls mécanismes de composition disponibles sont la composition simple d'objets et l'héritage, toutes les deux exprimées dans le code du programme.

On distingue cinq moments principaux où on peut fixer les décisions après la phase de conception : dans le code (une classe a un attribut d'un type donné), au moment de la construction de l'application (quelle version d'une classe utiliser), au moment du déploiement (fixer les paramètres d'installation ou décider du gestionnaire de base de données à utiliser), au moment de démarrer l'application (les paramètres d'exécution) ou lors de l'exécution (chercher dans un système de localisation dynamique d'objets).

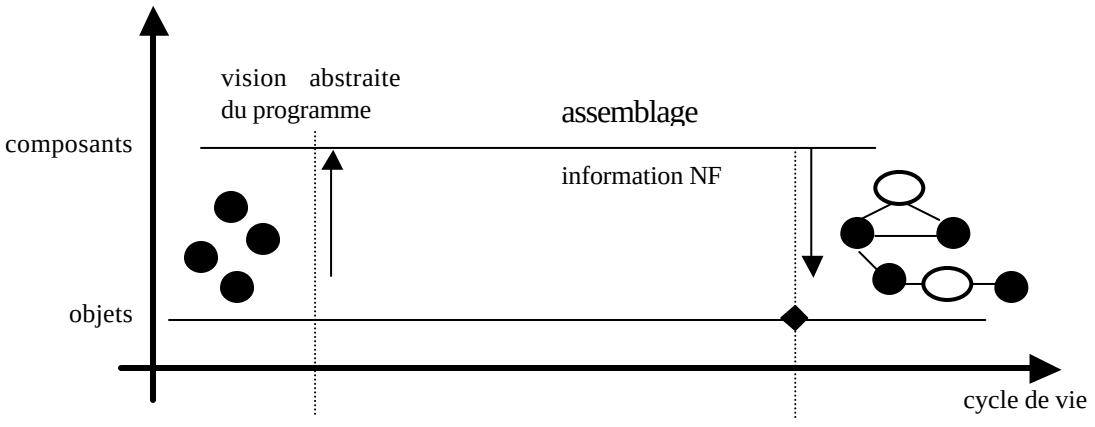


Un modèle de composants (de composition) est un ensemble de concepts et de mécanismes qui permettent de reporter quelques décisions architecturales du modèle d'objets aux étapes du cycle de vie postérieures au codage, introduisant ainsi des facilités additionnelles

d'adaptation. Pour ce faire, le modèle de composants fournit une façon d'abstraire le modèle d'objets, produit par l'étape de conception, en termes d'éléments de plus haut niveau d'abstraction : les composants. Avec ces éléments, et en se basant sur un ensemble de mécanismes de traduction, il est possible d'exprimer les composants en termes d'objets, au long des 4 étapes du cycle de vie postérieures au codage, et avec une architecture prenant en compte les BNF du problème.



Ceci permet de séparer la construction d'un programme en deux étapes différentes : le codage, où l'on bâtit les éléments de base du programme (i.e. les objets), et l'assemblage, où, en prenant en compte les BNF du problème, on définit une architecture adéquate. Le programme ainsi obtenu comporte les éléments de base construits et, possiblement, un ensemble de faux objets représentant quelques-uns des mécanismes du modèle de composants. Cette approche évite le lien direct entre l'architecture du modèle d'objets à la fin de l'étape de conception et l'architecture du modèle d'objets pendant l'exécution, ce qui rendait si difficile l'évolution causée par les changements des BNF du problème.



Il est important de remarquer que l'une des conséquences de cette approche est qu'avant de "redescendre" au niveau des objets, le processus de traduction peut interagir avec des modèles et des informations non fonctionnelles additionnelles élaborés lors des étapes du cycle de vie postérieures au codage. Ceci veut dire, par exemple, que le modèle exact d'objets qui va s'exécuter peut prendre en compte des informations du modèle de déploiement (sur quel type de système d'exploitation va s'exécuter le programme ou avec quelle version).

Dans la figure ci-dessus, un ovale dans le modèle d'objets de l'exécution pourrait représenter, par exemple, un mécanisme d'adaptation dynamique lui permettant d'aller chercher sur le réseau les éléments nécessaires pour répondre à un changement d'une caractéristique non-fonctionnelle.

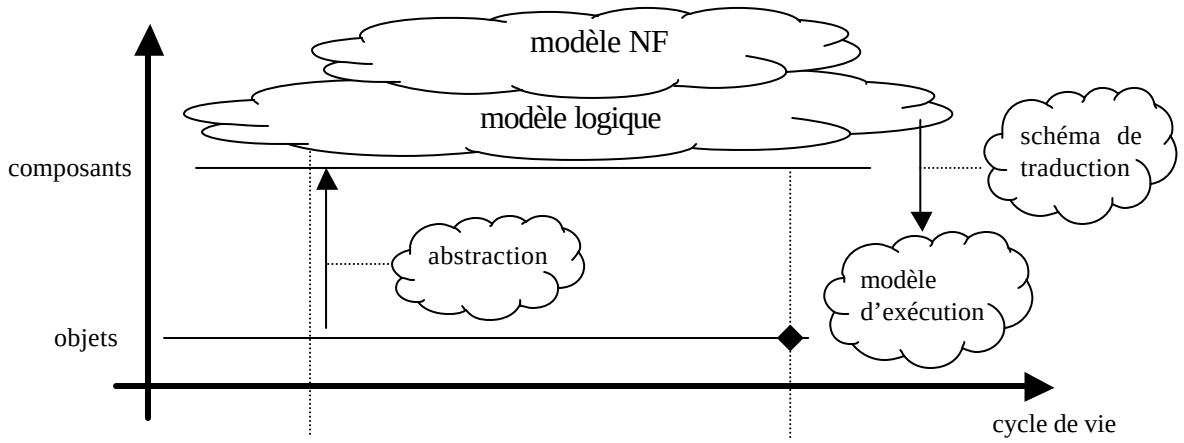
Il est clair que le modèle de composants vise surtout à résoudre un cas particulier du problème d'évolution concernant les changements des BNF, qui est appelé le problème d'adaptation. Si c'est le monde du problème ou le problème lui-même qui changent, la capacité de bien supporter cette évolution dépend du fait que la modification implique un changement entre les frontières des composants ou du fait que cette modification est incluse complètement dans un seul composant. Dans le premier cas, il est difficile de prédire les conséquences, car tout dépend du processus d'abstraction objet–composant, qui n'a pas été étudié jusqu'à maintenant. Dans le deuxième cas, où l'évolution est déjà isolée, le modèle supporte très bien le changement.

Cette vision à deux niveaux d'abstraction nous amène à définir un modèle de composants MC comme un triplet :

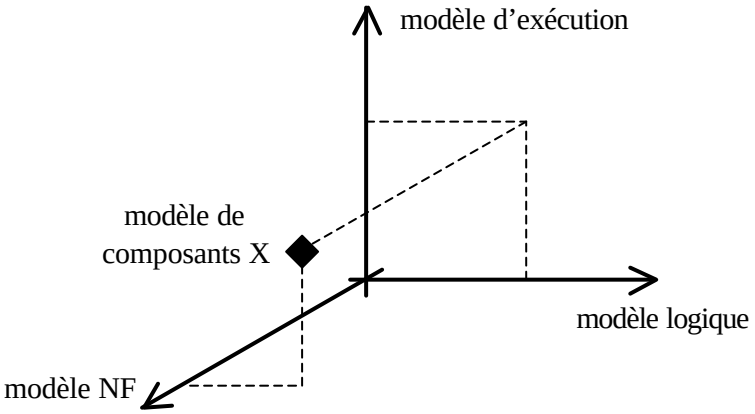
MC = < ML, M-NF, ME >

où :

- ML est le modèle logique d'un composant (i.e. comment décrire un composant et ses relations)
- M-NF est le modèle non-fonctionnel (i.e. quelles caractéristiques NF sont associées aux éléments du modèle logique)
- ME est le modèle d'exécution, qui définit l'architecture du modèle d'objets au moment de l'exécution. Le schéma de traduction d'un modèle de composants est capable de fournir un modèle d'exécution à partir du modèle logique et du modèle NF



Ceci veut dire qu'au moment de faire la conception d'un modèle de composants, il faut choisir un point dans un espace à trois dimensions, définies par les trois modèles qui constituent un modèle de composants. De même que pour comparer deux modèles de composants, il faut considérer ces trois dimensions.



Du point de vue de l'utilisation d'un modèle de composants, il faut avoir aussi :

- une façon d'abstraire le modèle d'objets et de remonter au niveau des composants
- un ensemble d'outils chargés de supporter le modèle, appelé le *framework* du modèle

Dans la suite, nous étudierons les trois dimensions d'un modèle de composants, et traiterons superficiellement le thème de son *framework*. Le seul sujet que nous n'abordons pas est celui du processus d'abstraction objet – composants, qui est, en ce moment, un problème ouvert, appartenant au domaine de la méthodologie, et qui est hors de la portée de ce travail.

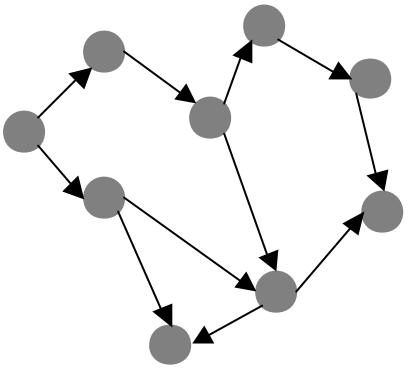
2.2. Le modèle logique d'un modèle de composants

D'un point de vue purement structurel, le modèle d'objets est un graphe où les objets sont les nœuds et les relations entre les objets sont les arcs. Toutes les caractéristiques de ce graphe sont définies au moment du codage, et toute évolution implique la modification des sources et la recompilation de l'application, ce qui parfois peut être un vrai problème.

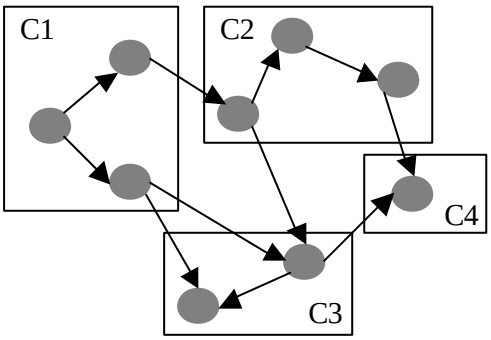
Un composant (de composition) est un artefact qui permet de grouper et d'isoler un sous-graphe du modèle d'objets, en définissant de façon explicite ses responsabilités et ses besoins par rapport au reste de l'application, ce qui lui permet d'évoluer de manière indépendante.

L'objectif d'un modèle de composants est de modéliser le monde du modèle d'objets, en termes d'éléments de plus haut niveau d'abstraction, sans le déformer et sans perdre ses propriétés d'évolution.

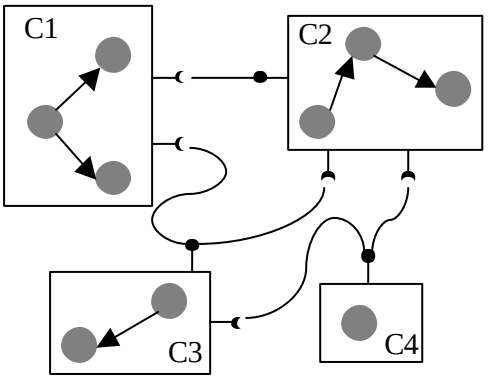
Les figures suivantes donnent une vision intuitive des idées élaborées ci-dessous, à l'aide de formalismes qu'on expliquera dans la suite :



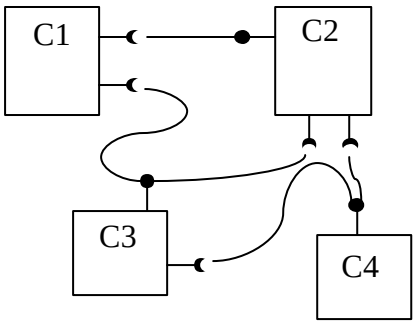
Graphe du modèle d'objets



Partition du graphe en sous-graphes



Définition des frontières



Vision boîte noire des sous-graphes

Le modèle logique d'un modèle de composants est défini par le triplet :

ML = < VE, VC, VI >

où:

- VE est la vision externe du composant : c'est-à-dire, la définition de sa frontière en termes de ses responsabilités fonctionnelles et de ses besoins. Pour ce faire, les modèles actuels se basent sur le concept de plusieurs types de ports, chacun exprimant un type particulier de relation avec le reste du modèle.
- VC est la vision de connexion de composants : une manière de montrer une relation entre les ports de deux composants. Du point de vue relation avec le modèle d'objets, les connexions montrent les relations qui existent entre les sous-graphes d'objets.
- VI est la vision interne du composant. Cette vision comporte trois parties : (1) les éléments internes du composant (avec sa structure) qui lui permettent de satisfaire ses contrats, (2) l'assignation des responsabilités externes aux éléments internes, et (3) l'assignation des ressources fonctionnelles externes aux éléments internes.

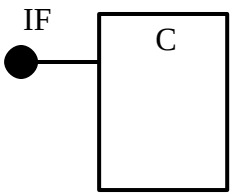
Dans les sections suivantes nous étudierons chaque dimension du modèle logique, et introduisons les éléments présents dans les modèles logiques des modèles de composants actuels.

2.2.1. Vision externe d'un composant

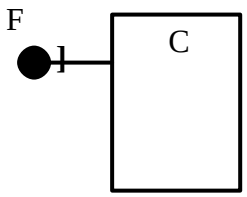
La façon d'isoler un composant du reste du modèle consiste à définir sa frontière par un contrat, décrivant les responsabilités et les besoins du composant. Ceci correspond à une vision boîte noire d'un sous-graphe d'objets.

Pour décrire cette frontière, le modèle à composants introduit plusieurs concepts :

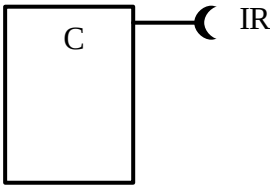
- Interface : comporte un ensemble de signatures de méthodes, groupées sous un type. C'est un concept déjà introduit dans plusieurs langages d'objets (i.e. java [GJS00]), mais inexistant dans le modèle original d'objets.
- Interface fournie IF : est un port qui représente un ensemble de services fournis par le composant, caractérisés par une interface. Dans certains modèles, une interface fournie a aussi un ensemble d'attributs associés.



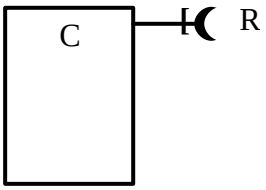
- Facette F: est un port qui représente un ensemble de services fournis par le composant, caractérisés par une interface, et un nom associé. La principale différence avec une interface fournie est que le composant peut fournir plusieurs fois la même interface sous des noms différents, ce qui est parfois important (deux fois la même fonctionnalité, chacune associée à une partie différente du problème).



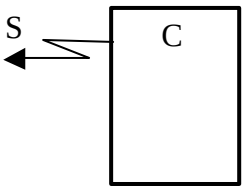
- Interface requise IR : est un port qui représente un ensemble de services dont le composant a besoin pour son fonctionnement, caractérisés par une interface.



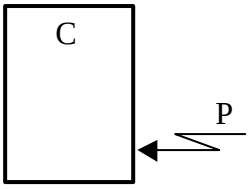
- Réceptacle R: est un port qui représente un ensemble de services dont le composant a besoin pour son fonctionnement, caractérisés par une interface et un nom associé. C'est le concept dual du concept de facette.



- Evènement : est un élément utilisé pour informer de façon asynchrone que "quelque chose vient de se passer". La plupart des langages d'objets utilisent les événements comme moyen de synchronisation, mais seulement c# [Arc01] l'introduit comme un élément primitif. Un événement est caractérisé par un type.
- Source S : est un port qui représente un événement qui va être émis par le composant. C'est une façon de déclarer que le composant est observable.



- Puits P : représente un événement requis par le composant, et émis par un autre composant comme moyen de coordination.



Les ports de type interface requise et réceptacle ont une cardinalité associée, qui définit le nombre de connexions que le port peut accepter simultanément. De même pour les sources.

La vision externe des composants de tous les modèles actuels se base sur les concepts présentés. Un langage servant à décrire cette vision externe d'un composant s'appelle un langage de description de composants.

Dans le cas général, on peut dire que la vision externe d'un composant est un couple :

VE = < R, B >

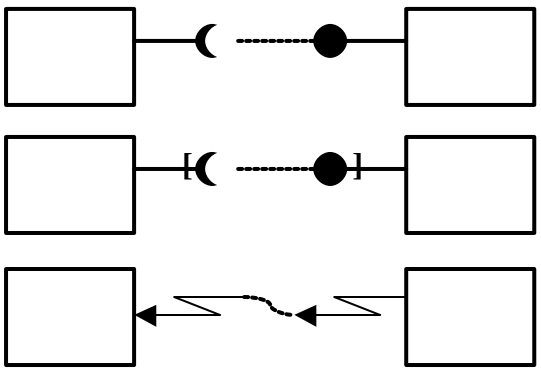
où :

R est un ensemble de ports de type interface fournie, facette ou source, représentant ses responsabilités fonctionnelles,
B est un ensemble de ports de type interface requise, réceptacle ou puits, représentant ses besoins fonctionnels.

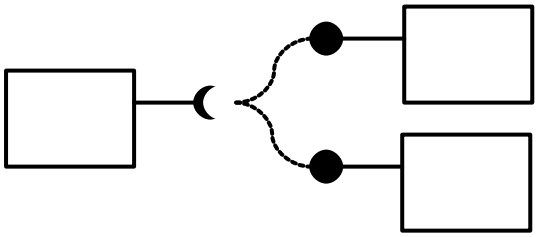
2.2.2. Vision de connexion des composants

Avec les frontières entre composants clairement identifiées, il suffit maintenant de connecter les composants entre eux, en suivant l'architecture du modèle d'objets. La connexion se fait alors entre couples de ports, et en suivant quelques règles :

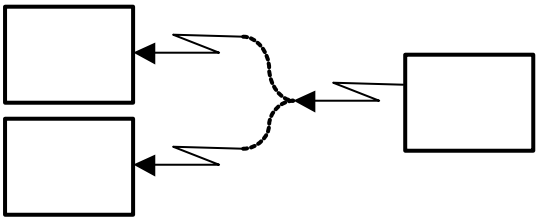
- une interface fournie peut se connecter à une interface requise ou à un réceptacle
- une facette peut se connecter à un réceptacle ou à une interface requise
- une source peut se connecter à un puits
- un port P1 (interface fournie, facette ou source) est compatible avec un port P2 (interface requise, réceptacle ou puits) si (1) P1 et P2 sont caractérisés par la même interface ou si (2) l'interface caractérisant P2 est un sous-type de l'interface qui caractérise P1.



En plus, les connexions doivent respecter la cardinalité des ports de type interface requise et réceptacle.



La connexion entre une source et un puits peut être multiple, dans le sens où une source peut se connecter à plusieurs puits à la fois.



Les connexions entre composants représentent un couplage beaucoup plus faible que celui entre objets, puisqu'une connexion entre composants ne fait que déclarer une dépendance fonctionnelle et non une dépendance vers une classe particulière.

Plus formellement, on peut dire que la vision de connexion d'un modèle de composants est définie par le couple :

VC = < C, Cx >

où :

C est un groupe de composants (pas nécessairement un ensemble),
Cx est l'ensemble de connexions entre les ports des composants du groupe C.

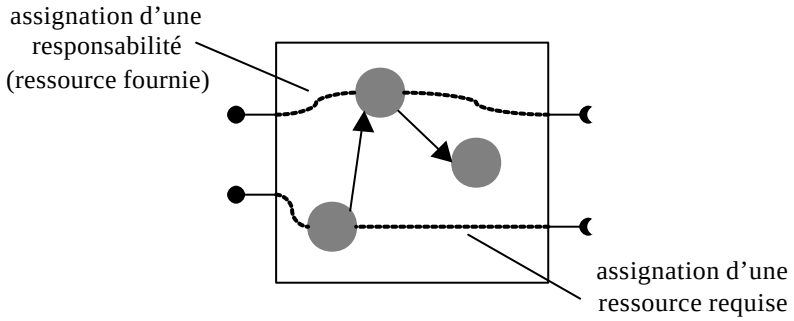
Un langage servant à définir les connexions entre composants se nomme un "langage d'assemblage de composants".

Plusieurs autres mécanismes de composition et de description sont envisageables, comme par exemple l'héritage de ports ou de composants, mais cela ne représente qu'une façon plus simple d'exprimer le même modèle, avec peut-être quelques facilités additionnelles d'évolution et de réutilisation. Dans ce document, nous n'approfondissons pas ce domaine.

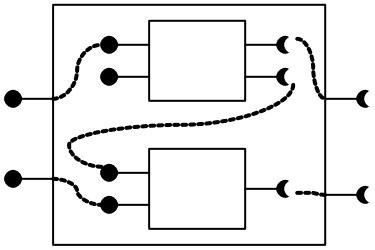
2.2.3. Vision interne d'un composant

La structure interne exprime le graphe d'éléments contenus dans un composant et sa relation avec les ressources fournies (ports de type interface fournie, facette ou source) et les ressources fonctionnelles dont le composant a besoin (ports de type interface requise, réceptacle ou puits). On peut considérer cette vision comme la vision physique du composant, bien que ce ne soit pas nécessairement le cas. Nous avons trois situations possibles :

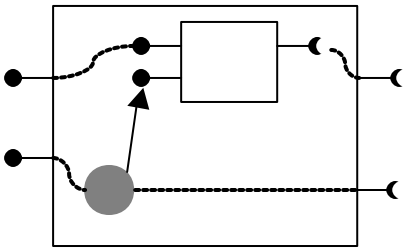
Cas 1: les éléments du composant sont un sous-graphe d'objets (représenté par son graphe de classes). Il faut donc définir la manière de construire physiquement un tel sous-graphe, et de déléguer les responsabilités du composant aux objets créés. Il ne faut pas oublier que, à ce niveau, un lien entre deux objets représente un couplage fort entre les classes de ces objets.



Cas 2 : les éléments implémentant le composant sont des instances de composant (représentées par un graphe de composants). Il suffit alors d'utiliser un langage d'assemblage pour le décrire, étendu avec une manière de connecter les ports des instances de composant internes avec les ports du composant qu'on est en train de décrire.



Cas 3: les éléments implémentant le composant sont un mélange d'objets et d'instances de composant. Comme les objets et les instances de composant sont des éléments de deux niveaux différents d'abstraction, il y a des contraintes pour les connecter directement : un objet peut éventuellement faire référence à une instance de composant, mais l'inverse n'est pas possible.



Nous considérons comme hors du modèle le cas dans lequel un objet (ou une instance de composant) est partagé par plusieurs composants, même si cela est faisable avec la plupart des modèles actuels.

La vision interne d'un composant peut être modélisée à travers un couple :

VI = < G, CE >

où :

G est un graphe de classes et de composants, (représentant la structure d'un graphe d'objets et d'instances de composants), appartenant à l'un des trois cas présentés ci-dessus.
CE est un ensemble de connexions entre les ports du composant, et les éléments du graphe G

Un langage servant à définir le graphe d'éléments et sa relation avec le contrat du composant s'appelle un langage de construction de composants.

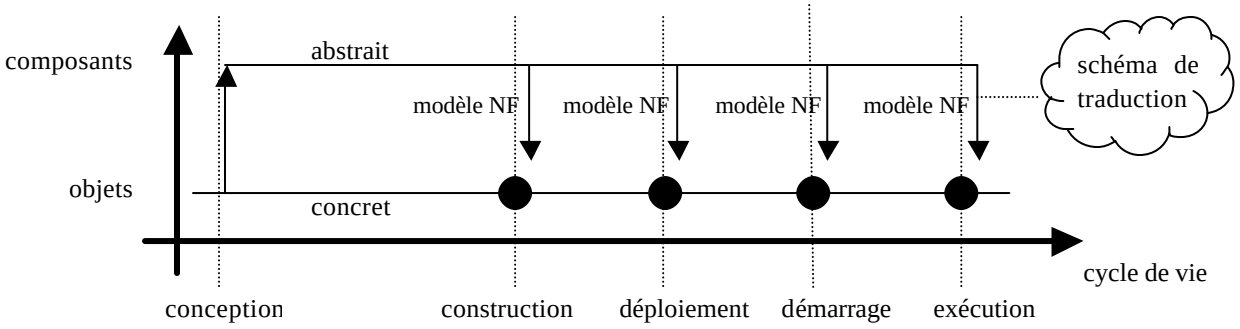
Dans la plupart des modèles de composants actuels, la description interne des composants n'est pas prise en compte, mais remplacée par le concept de fabrique. La fabrique d'un composant est alors un élément du modèle d'exécution (décrite dans le modèle NF), chargée de construire des instances de œ composant dans la mémoire. Avec ce concept, on peut substituer la description interne d'un composant (i.e. une approche déclarative), par un bout de code capable de créer les instances pendant l'exécution (i.e. une approche opérationnelle).

2.3. Le modèle non-fonctionnel

Il est certainement difficile de définir avec précision ce qu'est un modèle non-fonctionnel, quand la seule information dont on dispose est qu'il s'agit d'un aspect de l'application externe à sa simple fonctionnalité abstraite. Si l'on considère que la partie fonctionnelle d'une application explique le « quoi », mais pas le « comment », on peut dire que toute caractéristique concrète d'une application fait partie de son modèle non-fonctionnel. Le langage de programmation, le gestionnaire de bases de données choisi, et même le modèle d'objets qui implémente l'application sont des exemples des éléments appartenant au modèle non-fonctionnel.

Heureusement, dans cette section nous sommes concernés par un problème beaucoup plus simple, et pas par toute cette problématique, impossible à aborder sans une caractérisation et une classification plus fine.

L'objectif du modèle NF d'un modèle de composants de composition est de fournir au schéma de traduction et au modèle d'exécution l'information nécessaire pour que le modèle logique puisse être exécuté. Autrement dit, le modèle NF permet de concrétiser le modèle logique pour le rendre exécutable.



Il faut remarquer que comme le modèle de composants de composition ne considère qu'un seul monde, les aspects non-fonctionnels comme la persistance ou la distribution ne sont pas considérés ici, mais traités dans le contexte des composants de coordination.

Du point de vue contenu, le modèle NF comporte une définition concrète de toutes les caractéristiques que le modèle de composants permet de repousser aux étapes postérieures à la conception. Pour cette raison, sa structure dépend de plusieurs modèles, comme par exemple le modèle de déploiement et le modèle d'exécution.

Pour illustrer l'idée, imaginons quatre modèles de composants MC1, MC2, MC3 et MC4, avec le même modèle logique : un composant qui fournit une simple interface. La seule décision qu'on prend après l'étape de conception est la classe concrète qui va implémenter le composant. Dans le tableau suivant, on montre le type d'information qui comporte le modèle NF de chaque modèle de composants de l'exemple :

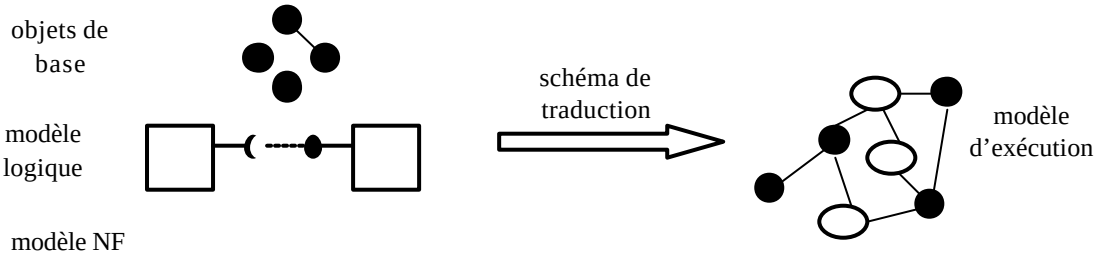
	fonctionnement	modèle NF
MC1	La classe concrète est choisie de manière statique	• Pour chaque composant, le modèle NF a le nom de la classe qui l'implémente
MC2	La classe concrète est choisie en fonction du système d'exploitation et de sa version	• Pour chaque composant, le modèle NF a le nom de la classe qui l'implémente dans chaque système d'exploitation et chaque version • Le système d'exploitation actuel
MC3	La classe concrète est choisie en exécution et elle peut changer selon un niveau de sécurité donné	• La description de la façon de se connecter à un serveur, utilisé comme une fabrique générique (par exemple, une adresse IP, un port et un protocole)
MC4	La classe concrète est choisie de manière statique, mais la fonctionnalité du composant n'est pas toujours disponible	• Pour chaque composant, le modèle NF a le nom de la classe qui l'implémente • Pour chaque composant, le modèle NF a les heures du jour où il est disponible (par exemple)

De façon générale, on peut dire que le modèle NF permet d'associer, à chaque élément ou connexion du modèle logique, des informations qui seront interprétées par les modèles des étapes postérieures à la conception. Il est impossible d'aller plus loin dans la caractérisation du modèle NF, car il y a un nombre infini de modèles imaginables et de combinaisons possibles.

Un langage servant à définir les caractéristiques d'un modèle NF s'appelle un langage non-fonctionnel. Il est possible de considérer le descripteur de déploiement de certains produits commerciaux (e.g. EJB) comme la description du modèle NF d'une application exprimée dans le langage non-fonctionnel de son modèle de composants.

2.4. Le modèle d'exécution et le schéma de traduction

Le modèle d'exécution définit l'architecture du programme au cours de l'étape d'exécution, en termes des objets de base et d'un ensemble de faux objets (connecteurs, adaptateurs, mandataires, etc.). Cette architecture est créée en suivant les informations du modèle logique et en prenant en compte les caractéristiques non-fonctionnelles définies dans le modèle NF.



Une bonne partie des propriétés et des capacités d'un modèle de composants est liée à son modèle d'exécution et à son modèle NF, plutôt qu'à son modèle logique.

L'objectif de cette section est de montrer l'espace des possibilités concernant le modèle d'exécution, et les conséquences d'un choix particulier. Ce qu'il faut remarquer aussi, c'est que pour le même modèle de composants il y a plusieurs *frameworks* possibles, chacun avec des langages, des formalismes et des outils différents. Bien qu'un *framework* soit simplement une implémentation pour supporter un modèle de composants, il est vrai aussi qu'il définit quelques-unes des propriétés non-fonctionnelles de l'application. Par exemple, ce n'est pas la même chose (du point de vue évolution) si on peut définir les connections entre composants avec un outil graphique ou avec un langage textuel de haut niveau, que de laisser câblé en dur dans le code des objets les appels vers une API qui permet de définir ces mêmes connexions. Dans la section suivante, nous étudierons de façon très superficielle le problème du *framework*, car il n'est pas facile de caractériser à ce niveau l'espace des possibilités.

Dans la suite nous présentons le modèle d'exécution comme un modèle à plusieurs dimensions : un modèle de base, qui définit les caractéristiques structurelles, et plusieurs axes de décisions avec les principaux aspects de l'exécution.

2.4.1. Le modèle de base

D'abord, on peut dire que le modèle d'exécution peut se baser sur deux approches distinctes :

- La première possibilité est la traduction complète vers un modèle différent : dans ce cas, les outils du *framework* traduisent tous les éléments du modèle logique vers un ensemble d'éléments d'un autre modèle, pour utiliser le *framework* de ce modèle pendant l'exécution. Lors de cette traduction on introduit et on connecte les faux objets correspondant au modèle NF.

  Un exemple typique de cette approche est l'utilisation du modèle d'objets et du *framework* de java (sa JVM et son compilateur) comme cibles de la traduction. Dans ce

cas, tous les éléments du modèle logique et toutes les propriétés du modèle NF sont traduits en termes d'objets, compilés et ensuite exécutés sur la machine virtuelle de java.

Bien qu'il soit possible de laisser des informations concernant le modèle logique pour les retrouver pendant l'exécution (introspection du modèle logique), la principale conséquence de cette approche est que toute évolution implique une nouvelle phase de traduction, ce qui la rend plutôt statique.

- La deuxième possibilité est l'implémentation d'une machine virtuelle du modèle de composants (CVM : *Component Virtual Machine*), qui est un interpréteur du modèle logique et du modèle NF. Cette machine virtuelle correspond typiquement à une API. Cependant, on pourrait développer un langage intermédiaire (une espèce de *bytecode*) et l'interpréter.

  Pour permettre l'implémentation de quelques caractéristiques NF, la CVM peut aussi émettre des événements informant de ses changements d'état.

  Pour obtenir le modèle d'exécution, il suffit de faire la traduction entre les langages d'expression du modèle logique et le langage d'expression du modèle NF vers l'API (ou le langage intermédiaire) de la CVM.

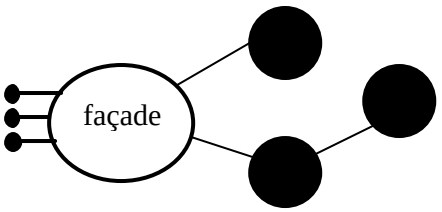
  Avec cette approche, il est possible de doter le modèle d'exécution de la capacité de réflexion, facilitant les BNF d'adaptation dynamique.

Pour résumer, il est possible d'affirmer que les deux approches on la même capacité d'expression, et excepté pour l'adaptation dynamique, on peut dire qu'elles supportent les mêmes modèles logique et NF avec le même niveau de difficulté.

2.4.2. Représentation d'un composant

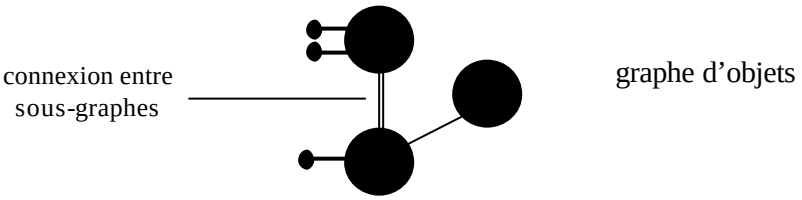
La représentation dans le modèle d'exécution d'une instance de composant peut se faire normalement de deux manières :

- Une instance de composant est représentée par un objet, qui assume toutes les responsabilités du composant. Ceci peut impliquer, dans certains modèles, l'introduction d'un faux objet (patron façade[GHJ95]) qui est capable de gérer la structure interne de l'instance pour donner l'idée d'une seule unité logique.



- La deuxième possibilité est de laisser les objets (implémentant une instance de composant) connectés d'une certaine façon entre eux, et avec un protocole prédéfini qui leur permet de fournir les divers services à travers chacun de ses éléments internes. Dans la figure on utilise une ligne simple pour montrer les relations de composition

entre les objets d'un sous-graphe, et une ligne double pour montrer les connexions entre les divers sous-graphes du composant.



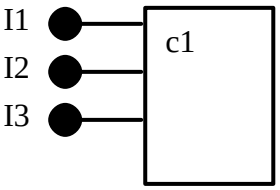
Dans ce cas, chaque racine d'un sous-graphe doit être capable de répondre à la question de qui implémente les autres interfaces du composant, et de retourner cet objet. Cette façon de faire travailler ensemble un groupe d'objets est connue sous le nom de protocole *QueryInterface* (QI).

Le principal problème de cette dernière solution est la pollution du code des objets de base avec l'implémentation du protocole QI, et le fait que la réutilisation de ces objets pour les assembler sous un autre modèle de composants ne soit pas évidente.

2.4.3. Navigation entre interfaces fournies (ou facettes)

Si, dans le modèle logique, un composant fournit plusieurs interfaces, il y a toujours un problème pour traduire la navigation d'une interface vers une autre. D'un point de vue conceptuel, cette navigation n'est qu'un casting entre les types des interfaces fournies, mais dans la pratique ceci peut être un peu plus compliqué.

```
I2 i2 = (I2) c1   // façade
I2 i2 = (I2) i1   // graphe d'objets
```



En interne, par contre, il y a plusieurs cas possibles, résumés dans les tableaux suivants :

- Cas 1 - modèle de base par traduction :

façade		graphe d'objets	
interface fournie	facette	interface fournie	facette
casting direct	ajout par génération d'une interface de service de la façade, avec la responsabilité de retourner une facette étant donné son nom	chaque objet du graphe implémente *QueryInterface* (QI), qui retourne l'objet chargé de la fonctionnalité demandée	chaque objet du graphe implémente une variante de QI, nommée *QueryFacette* (QF), qui retourne l'objet chargé d'implémenter une facette
(I2) c1	(I2) c1.getFacette("N2")	(I2) i1.QI(I2)	(I2) i1.QF("N2")

- Cas 2 - modèle de base avec CVM :

façade		graphe d'objets	
interface fournie	facette	interface fournie	facette
casting direct	utilisation des services de la CVM pour rechercher une facette d'un composant, étant donné son nom	▪ il est possible d'utiliser le protocole QI ▪ on peut centraliser la responsabilité de la navigation entre interfaces dans la CVM	▪ il est possible d'utiliser le protocole QF ▪ on peut centraliser la responsabilité de la navigation entre facettes dans la CVM
(I2) c1	(I2) CVM.QF(c1, "N2")	(I2) i1.QI(I2) (I2)CVM.QI(i1, I2)	(I2) i1.QF("N2") (I2) CVM.QF(i1, "N2")

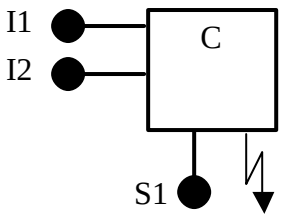
2.4.4. Création d'une instance de composant

Pour la création des instances de composant, il y a deux approches possibles : la première option est de centraliser cette responsabilité dans le modèle d'exécution. Il y a une espèce de fabrique générique, capable de créer n'importe quelle instance de composant à partir de sa description interne. La seconde possibilité est de laisser cette responsabilité à chaque composant, qui déclare dans le modèle NF sa fabrique, et qui sera utilisée pour créer de nouvelles instances. Dans ce dernier cas, la fabrique peut être simplement un bout de code qui appelle les constructeurs des classes du graphe d'objets.

2.4.5. Opérations de service du composant

Pour la gestion de certaines caractéristiques non-fonctionnelles (introduites plus tard dans les composants de coordination), le composant est souvent obligé d'implémenter un ensemble de procédures, dites les opérations de service du composant. Ces opérations sont caractérisées par des interfaces, et sont utilisées aussi bien dans le modèle de base avec CVM que dans le modèle de base par traduction.

Parfois, le composant doit aussi émettre des événements pour que le modèle d'exécution soit capable de réagir à certaines situations pendant l'exécution du composant. La situation générale peut se visualiser dans la figure suivante :



2.5. Le *framework* d'un modèle de composants

Le *framework* d'un modèle de composants est l'ensemble des outils, des langages, des formalismes et des bibliothèques de classes qui permettent de construire et d'exécuter une application définie en termes d'un modèle de composants.

Dans le cas général, on peut dire qu'il comporte :

- un formalisme pour décrire la frontière d'un composant
- un formalisme pour décrire l'assemblage des composants
- un formalisme pour décrire la structure interne d'un composant
- un formalisme pour définir le modèle NF
- un traducteur / compilateur pour chacun des formalismes
- une implémentation de la machine virtuelle du modèle (optionnelle)

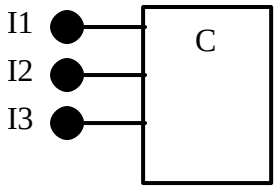
2.6. Classification des modèles actuels

Dans cette section, nous présentons les modèles de composants (de composition), les plus importants à niveau commercial, dans le cadre théorique que nous venons d'introduire. Nous avons deux objectifs : d'une part, situer chaque modèle dans l'espace de possibilités montré, en faisant abstraction de tous les aspects technologiques, et des terminologies particulières, pour pouvoir les comparer. Et d'autre part, si nous arrivons à exprimer tous les aspects importants de chaque modèle, nous validons le cadre conceptuel proposé dans ce document.

Nous voulons faire une présentation très simplifiée et schématique des principaux aspects des modèles, ce qui parfois peut nous amener à ignorer des caractéristiques considérées comme importantes dans les produits commerciaux respectifs.

Pour faciliter la lecture, nous utilisons le même format (un tableau), avec l'ensemble d'items constituant les dimensions du modèle de composants, même si parfois il faut adapter le contenu à chaque situation particulière.

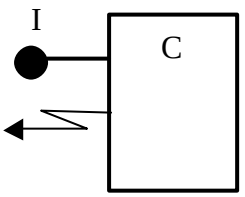
2.6.1. Le modèle de composants de java

description	Le cas le plus simple de composant est le composant introduit dans le langage java. Tous les langages objet comme java avec le concept d'interface et avec les mécanismes de liaison dynamique et de création d'instances par le nom de sa classe, implémentent le même modèle.
modèle logique (vision externe)	
modèle logique (connexions)	Il n'y a pas de modèle de connexion externe au programme. L'assemblage se fait à travers la composition au niveau objet (un

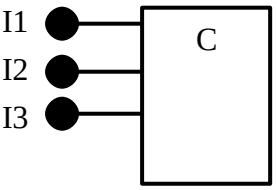
	objet a un attribut du type d'une des interfaces).
modèle logique (vision interne)	Inexistante : utilise le schéma de fabrique.
modèle NF	Définition de la fabrique de chaque composant (le nom de la classe concrète implémentant les interfaces. Ce nom peut être défini même durant la phase d'exécution de l'application)
modèle d'exécution (modèle de base)	Aucune traduction, car le langage supporte déjà les concepts (interface) et les mécanismes nécessaires (liaison dynamique et création d'instances par le nom de sa classe). Le modèle utilise l'introspection de java mais il n'est pas réflexif.
modèle d'exécution (représentation)	L'instance de composant est un objet implémentant les interfaces définies dans le modèle logique. Si l'objet a besoin de se connecter à d'autres composants, la fabrique est responsable de l'appel aux fabriques correspondantes.
modèle d'exécution (navigation)	Mécanisme de casting de java.
modèle d'exécution (création d'instances)	Chaque composant est responsable.
modèle d'exécution (opérations service)	Aucune.
framework	Le même *framework* du langage (i.e. la machine virtuelle de java (JVM) et son compilateur).
autres informations	Ce type de composant correspond conceptuellement à l'application de deux patrons de programmation (le pont et la fabrique [GHJ95]). C'est pour cela, que quelques auteurs [Mey99] affirment que les composants sont juste un patron, qui reflète les bonnes normes de conception d'objets (rendre obligatoire ce qui était seulement un conseil de qualité de conception). Comme il n'y a pas réellement un mécanisme de composition différent de celui des objets, le seul avantage est qu'on évite le fort couplage entre deux classes concrètes en permettant une évolution indépendante, et que le choix de la classe à utiliser peut être fait avec des informations additionnelles concernant le contexte des étapes ultérieures au codage.

2.6.2. Le modèle des JavaBeans

description	Les *javaBeans* [BEAN97, Eng97] sont un modèle de composants, introduit par Sun Microsystems en 1996, pour faciliter la création d'interfaces graphiques à partir d'un mécanisme de composition générique. Le modèle se réduit à un ensemble de conventions de programmation et à un ensemble de normes lexicales, permettant par introspection de retrouver l'interface et les événements générés par

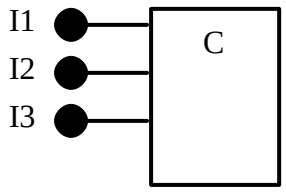
	le composant. Optionnellement, cette information peut être déclarée de manière explicite dans une classe additionnelle (*BeanInfo*).
modèle logique (vision externe)	 L'interface correspond à un ensemble de méthodes d'accès aux attributs (méthodes `get` et `set`). Les attributs ainsi gérés s'appellent les propriétés du *javaBean*
modèle logique (connexions)	Les *javaBeans* ont été définis pour être assemblés par des outils externes. Ceci veut dire que le modèle fournit les mécanismes de base, mais que la composition est une responsabilité du *framework*.
modèle logique (vision interne)	Vision interne inexistante.
modèle NF	La même classe qui décrit le composant (par introspection), définit la manière de créer ces instances (elle a un constructeur par défaut obligatoire). Le nom de cette classe est défini pendant l'exécution des outils de composition.
modèle d'exécution (modèle de base)	Aucune traduction, car le langage java supporte déjà tous les mécanismes de base. Il y a de l'introspection du modèle logique (en utilisant l'introspection de java), mais le modèle d'exécution n'est pas de réflexif.
modèle d'exécution (représentation)	L'instance du composant est un objet de la classe qui décrit le composant
modèle d'exécution (navigation)	Non applicable.
modèle d'exécution (création d'instances)	Chaque composant est responsable.
modèle d'exécution (opérations service)	Aucune.
framework	▪ Le *framework* de java. ▪ La classe java.beans.Beans, qui fournit quelques services de base, comme la création d'instances. ▪ Des outils de composition de *javaBeans* (e.g. NetBeans [NetBeans], BeanBox [BeanBox]). ▪ Une extension de la spécification originale [BEAN98] rajoute le concept de contexte (*BeanContext*), ce qui permet la gestion de services partagés.
autres informations	Bien que le modèle n'inclue pas un langage de description externe des composants, il est possible de le rajouter pour permettre la définition de langages d'assemblage, comme cela a été proposé dans Beanome [CFD02], où l'on peut décrire l'interface d'une application graphique en utilisant des langages textuels de haut niveau.

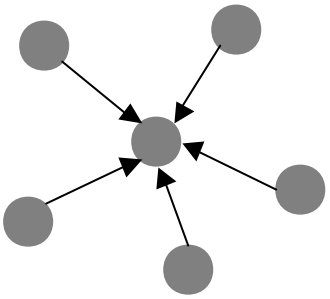
2.6.3. Le modèle COM

description	COM (Component Object Model) [COM, Box98] est un modèle de composants introduit par Microsoft en 1995. COM consiste notamment en un modèle d'exécution (présenté en §II.5.4), mais qui, du point de vue logique, comporte tous les éléments d'un modèle de composants. Il a été conçu pour le système d'exploitation Windows, et même si l'on peut écrire un composant COM en n'importe quel langage, il est très lié au langage C++.
modèle logique (vision externe)	 Il n'y a pas de langage de description externe du composant. Les interfaces fournies se calculent par rapport au graphe d'objets retourné par la fabrique du composant, ce qui implique que l'ensemble d'interfaces n'est nécessairement ni prédéfini ni constant. Ceci ouvre la possibilité de gérer dynamiquement, non seulement l'évolution structurelle d'un monde, mais aussi l'évolution fonctionnelle.
modèle logique (connexions)	Il n'y a pas de modèle de connexion externe au programme. L'assemblage se fait à travers la composition au niveau objet (un objet a un attribut du type d'une des interfaces).
modèle logique (vision interne)	Inexistante : utilise le schéma de fabrique.
modèle NF	Définition de la fabrique. La fabrique est capable de créer le graphe d'objets et de retourner l'objet du composant chargé d'implémenter une interface particulière.
modèle d'exécution (modèle de base)	Aucune traduction, car les langages supportés doivent permettre la gestion de pointeurs à routines, qui est la seule contrainte imposée par le modèle. Il n'y a ni introspection ni réflexion du modèle.
modèle d'exécution (représentation)	Un graphe d'objets avec les responsabilités partagées.
modèle d'exécution (navigation)	Protocole *QueryInterface*.
modèle d'exécution (création d'instances)	Chaque composant est responsable.
modèle d'exécution (opérations service)	Aucune.
framework	▪ Le système d'exploitation Windows. ▪ Les compilateurs des divers langages doivent générer de façon particulière le code exécutable. ▪ Il y a une interface (*IUnknown*) dont toutes les interfaces doivent hériter, et qui, en plus de la méthode *QueryInterface*, inclut des

	méthodes de gestion du cycle de vie des objets du graphe (*AddRef*, *Release*).
autres informations	COM propose deux mécanismes additionnels de composition : l'inclusion (*containment*) et l'agrégation, mais qui correspondent simplement à l'application du patron délégation [GHJ95] et à la restructuration par programmation du graphe d'objets.

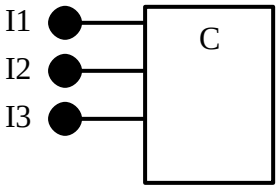
2.6.4. Les modèles OM et OMI

description	OM (Object Modeler) [EFSO2, San02] est un modèle de composants introduit par Dassault Systèmes, comme une réponse aux problèmes d'adaptation des grosses applications écrites en C++. OMI [VE00] est un modèle développé dans le laboratoire LSR, avec le même modèle logique de l'OM, mais avec un modèle d'exécution basé sur une CVM. L'objectif de l'OMI était d'étudier des variantes du modèle OM pour réduire la complexité de quelques-uns de ces aspects.
modèle logique (vision externe)	La vision externe d'un composant OM partage la plupart des caractéristiques du modèle logique de COM, sauf que dans l'OM on trouve deux mécanismes additionnels : l'adhésion conditionnelle d'interfaces et l'héritage de composants. Avec l'adhésion conditionnelle, un composant implémente une interface sous certaines conditions du contexte d'exécution.  Avec l'héritage, un composant est défini comme une extension d'un autre composant. Ce mécanisme d'héritage est très lié à l'héritage de la structure interne des composants, et comporte plusieurs problèmes conceptuels. Le modèle propose un langage de description externe des composants.
modèle logique (connexions)	Il n'y a pas de modèle de connexion externe au programme. L'assemblage se fait à travers la composition au niveau objet (un objet a un attribut du type d'une des interfaces).
modèle logique (vision interne)	A la différence de COM, le modèle OM introduit une vision interne des composants. Pour lui, un composant a un graphe d'objets constitué par un objet central (appelé la base) et un ensemble d'objets périphériques (appelés les extensions) connectés à la base, avec les règles suivantes : ▪ une base ne connaît pas ses extensions, ▪ une extension fait référence à sa base, mais pas aux autres extensions

	 L'héritage des composants est le résultat de l'héritage de ses objets centraux, avec tout ce que cela implique. l'OM introduit un langage de description interne des graphes d'objets, ce qui va permettre la centralisation du processus de création d'instances. Chaque composant a un nom (le nom de la classe de la base), ce qui permet de faire le lien entre la description interne et la description externe.
modèle NF	Le modèle OM propose le mécanisme de délégation d'interfaces, avec lequel il est possible de reporter à la phase d'exécution la décision de qui implémente une interface. Dans la description externe on déclare cette interface comme fournie (et déléguée), et dans la description interne, le graphe d'objets reste "incomplet". Le modèle fournit les mécanismes pour le compléter pendant l'exécution.
modèle d'exécution (modèle de base)	OM : il n'y a pas de processus de traduction, car les mécanismes ont été rajoutés au langage C++ en utilisant des macros et des bibliothèques de classes. OMI : il y a une CVM réflexive chargée de maintenir pendant l'exécution une description du modèle logique et de la structure interne de chaque instance construite.
modèle d'exécution (représentation)	Un graphe d'objets avec les responsabilités partagées
modèle d'exécution (navigation)	OM : chaque objet (bases et extensions) du graphe interne du composant implémente le protocole *QueryInterface*. OMI : le protocole *QueryInterface* est centralisé dans la CVM, ce qui simplifie l'écriture des bases et des extensions d'un composant
modèle d'exécution (création d'instances)	La création d'instances est centralisée, et se fait en utilisant la description interne de chaque composant. Pour créer une instance, il suffit de fournir le nom du composant (le nom de sa base)
modèle d'exécution (opérations service)	Aucune opération de service
framework	OM : il y a une bibliothèque de classes, un ensemble de macros, un langage de description externe et interne, des outils de visualisation du modèle (OMVT - *Object Modeler Visualization Tool* [San02]), des outils de validation de la structure, etc.

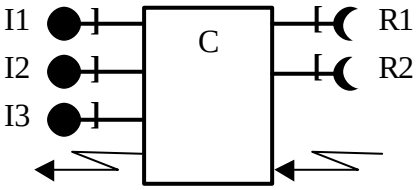
	OMI : il y a, en plus, la CVM.
autres informations	Nous avons défini, formalisé et implémenté le modèle OMI en 2000 [VE00] avec des résultats plus qu'encourageants pour des langages comme java, mais difficilement généralisables dans des langages comme C++.

2.6.5. Le modèle OSGi

description	OSGi (*Open Service Gateway Initiative*) [OSGi] est un modèle de composants introduit en 1999, visant, notamment, les problèmes liés à l'exécution dans des dispositifs avec des contraintes de mémoire et avec des BNF d'adaptation dynamique. Le modèle a été proposé par un groupe de 80 sociétés concernées par cette problématique. Le modèle est destiné à fonctionner avec java, donc il se base fortement sur le *framework* de ce langage. Avant de commencer, il est important de noter que dans OSGI un composant -tel qu'il a été présenté dans ce document- est appelé un service, et que le terme composant est utilisé en OSGI pour faire référence à un groupe de services liés par de caractéristiques physiques (ils partagent le même fichier de déploiement).
modèle logique (vision externe)	 OSGi a le même modèle logique que le modèle de composants de java. Les différences se trouvent dans le modèle d'exécution. Même si dans un *bundle* (voir modèle d'exécution) il est possible de déclarer explicitement les besoins fonctionnels des composants, cette information sert seulement de documentation.
modèle logique (connexions)	Il n'y a pas de modèle de connexion externe au programme, car, d'après la philosophie même du modèle, toute la composition entre les instances de composants doit se faire de manière dynamique, en fonction des ressources disponibles à chaque instant de l'exécution. La responsabilité de l'assemblage est partagée entre l'activateur d'un *bundle*, le moteur du modèle (appelé le *framework* en OSGi), qui permet de retrouver les instances nécessaires, et le composant lui-même.
modèle logique (vision interne)	Inexistante
modèle NF	Les aspects non-fonctionnels sont liés à la gestion des *bundles*. Même s'il n'y a pas de déclaration externe des caractéristiques non-fonctionnelles des composants (à part sa classe), le modèle d'exécution fournit tous les éléments nécessaires pour permettre l'adaptation dynamique. Cette gestion se fait au niveau de la programmation de chaque composant et non pas au niveau déclaratif.

modèle d'exécution (modèle de base)	étant donné que les composants (et ses instances) peuvent être ajoutés et enlevés du modèle à tout instant, le modèle d'exécution comporte un ensemble de mécanismes et éléments pour supporter cette fonctionnalité : ▪ *bundle* : unité physique de déploiement, composée d'un activateur et d'un ensemble de composants. Au moment du démarrage d'un bundle, son activateur est appelé pour déclarer les composants contenus. A chaque composant, il peut lui associer un ensemble d'attributs, et un sous-ensemble des interfaces implémentées par le composant. ▪ un composant (appelé service en OSGi) est défini à travers la classe qui implémente le composant. ▪ au moment de chercher dans le modèle une interface fournie par un composant, le modèle d'exécution permet l'utilisation d'un langage de requête (langage basé sur LDAP [WD00]) pour exprimer, en plus du type, les attributs exigés. En résumé, l'exécution se base sur une CVM, avec une API pour la gestion de *bundles* et un ensemble d'événements émis. Ces événements correspondent à l'arrivée de nouveaux composants, le départ d'autres, etc.
modèle d'exécution (représentation)	L'instance de composant est un objet implémentant les interfaces définies dans le modèle logique. Si l'objet a besoin de se connecter avec d'autres composants, la CVM fournit leur références.
modèle d'exécution (navigation)	Mécanisme de casting de java
modèle d'exécution (création d'instances)	Au moment de déclarer un composant, le moteur crée une instance et la met à disposition de tous les autres. Ceci veut dire que le moteur va retourner cette instance à chaque fois qu'un composant cherche cette fonctionnalité. Pour changer ce fonctionnement par défaut, un composant doit implémenter l'interface *ServiceFactory*, qui est appelée par le moteur.
modèle d'exécution (opérations service)	Aucune
framework	La plupart des services fournis par le *framework* sont liés à la gestion du cycle de vie des bundles. Il doit, par exemple, générer des événements qui peuvent être écoutés par les composants (e.g. un *bundle* qui vient de disparaître), de telle façon que chacun puisse s'adapter toujours aux nouvelles conditions de l'exécution.
autres informations	Le travail de Cervantes et al. [CER02] vise à proposer des extensions du modèle d'exécution et du modèle NF d'OSGi, et traduire ces extensions en termes des mécanismes déjà disponibles sur le *framework* du modèle de base.

2.6.6. Le modèle CCM

description	CCM (*CORBA Component Model*) [CCM, MM01] est, à plusieurs points de vue, le modèle de composants le plus complet du moment. Il a été proposé par l'OMG (Object Management Group) en 1999, et il inclut, à la fois, des mécanismes pour la composition de composants et pour la coordination entre mondes. Il est à noter que, en ce moment, il n'y a pas d'implémentation complète du modèle, et que la spécification est toujours en cours de raffinement. CCM est composé de 5 modèles. Dans cette section nous ne présentons que les aspects, les plus importants, du créneau de composition du modèle, car résumer plus de 1000 pages de spécification en moins de deux pages, implique beaucoup de simplifications.
modèle logique (vision externe)	 CCM a un langage de description externe de composants (OMG IDL3), qui correspond à une extension du langage IDL (*Interface Definition Language*) de CORBA 2.0. CCM a un mécanisme d'héritage simple entre composants, ce qui permet la réutilisation du modèle logique.
modèle logique (connexions)	Il n'y a pas de langage déclaratif d'assemblage en CCM. Les connexions entre composants se font dans le code à travers d'un ensemble de méthodes rajoutées au composant (par génération d'interfaces et par héritage), qui permettent de gérer les ports comme des attributs "publics". Pour les facettes: • Par héritage, tout composant implémente un ensemble de méthodes génériques de navigation (e.g. `provide_facet`, `provide_all_facets`), qui permettent l'introspection du composant • Par génération, dans le composant il y a une méthode `provide_`<nom-facette> pour chaque facette déclarée Pour les réceptacles: • Par héritage, tout composant implémente un ensemble de méthodes génériques de connexion : `connect` et `disconnect`. • Par génération, dans le composant il y a une paire de méthodes `connect_`<nom-receptacle> et `disconnect_`<nom-receptacle> pour chaque réceptacle défini, qui permettent de gérer la connexion entre un réceptacle et une facette. • Par génération, dans le composant il y a une méthode `get_connection_`<nom-receptacle>, qui permet de récupérer la facette connectée à un réceptacle donné. • Les réceptacles peuvent être simples ou multiples. Dans le cas de

	réceptacles multiples, le modèle introduit le concept de *cookie* pour les identifier lors de l'exécution. Avec les méthodes décrites ci-dessus, le programmeur est responsable de gérer les connexions entre ses instances de composants. Pour la gestion des ports asynchrones (sources et puits) on a un ensemble de méthodes équivalent à celui défini pour les facettes et les réceptacles.
modèle logique (vision interne)	CCM a un langage de description interne (CIDL: *Component Implementation Description Language*). Ce langage permet de définir l'ensemble d'objets CORBA qui vont implémenter les différentes facettes. Un composant est structuré comme un ensemble d'exécuteurs (*executors*), et un exécuteur comme un ensemble de segments. Chaque segment implémente au moins une facette. Il n'y a pas d'opérateurs de composition de segments.
modèle NF	La seule caractéristique NF définie est la fabrique du composant (tout le reste des caractéristiques, comme la persistance, appartient au modèle de coordination). La fabrique est définie comme un exécuteur particulier exprimé en CIDL.
modèle d'exécution (modèle de base)	CCM peut être considéré comme étant simplement une couche logique additionnelle au-dessus de CORBA. Le modèle d'exécution de CCM est alors une simple traduction, des éléments logiques introduits, en termes des éléments du modèle d'exécution de CORBA. Etant donné que cette traduction se fait par génération de code, la structure logique est statique, ce qui empêche toute adaptation dynamique structurelle [MM1].
modèle d'exécution (représentation)	Un composant de CCM est représenté avec une façade. Cette façade est un objet CORBA (la référence de base) qui implémente les méthodes génériques d'introspection et de connexion du composant. Chaque facette, de son côté, est aussi implémentée par un objet CORBA. Cet objet implémente une méthode qui lui permet de récupérer la façade, ce qui complète les facilités de navigation du modèle.
modèle d'exécution (navigation)	La navigation se fait avec les méthodes décrites dans les sections précédentes.
modèle d'exécution (création d'instances)	Chaque composant est responsable de déclarer sa fabrique.
modèle d'exécution (opérations service)	Aucune opération de service.
framework	L'implémentation la plus connue de CCM est openCCM [MM01], qui est capable de générer du code pour plusieurs des *frameworks* disponibles dans le marché (e.g. ORBacus[ORBacus]).
autres informations	CCM est la réponse à l'énorme complexité et difficulté pour utiliser l'ensemble de services fournis par CORBA. Il y a une telle quantité de détails et de problèmes technologiques que la tâche de programmer et maintenir une application est devenue un obstacle

	difficile à surmonter.

2.7. Conclusions et discussion

Dans cette première partie du chapitre, nous avons étudié les composants de composition. L'objectif de ce type de composant est de proposer une solution aux limitations des objets concernant l'évolution.

Les principaux aspects que nous avons montrés sont :

- La stratégie des composants de changer de niveau d'abstraction pour permettre un vrai processus d'assemblage (composition de boîtes noires) comme mécanisme de programmation
- La façon d'isoler de l'application une partie de ses caractéristiques NF, et de pouvoir les exprimer après la phase de conception
- Un cadre conceptuel, à trois dimensions, pour montrer la structure et le contenu d'un modèle de composants (de composition)
- L'étude des principaux modèles commerciaux de composants, en se basant sur le cadre conceptuel du point antérieur

Cette présentation nous permet de tirer les conclusions suivantes :

- Il est clair qu'avec les composants (de composition) il y a un énorme potentiel, et une grande quantité de possibilités qui n'ont pas été exploités, ni même étudiées jusqu'à maintenant. Autant au niveau de la définition des modèles d'exécution et non-fonctionnel, que dans la construction des *frameworks* de support (e.g. définition de langages NF). Par exemple, la construction d'une plate-forme pour le développement de composants auto-adaptables (s'adaptant par propre initiative à chaque situation précise, et se restructurant de manière intelligente), dynamiques (capables d'aller chercher dans des bibliothèques les éléments qui lui manquent), hétérogènes (chaque partie écrite dans un langage ou technologie différente), et avec un mécanisme d'évolution fonctionnel (une évolution du protocole *QueryInterface*) est parfaitement envisageable.

- Tous les aspects méthodologiques sont des points ouverts, où l'on trouve très peu de recherche. Le choix du modèle de composants à utiliser est une décision de conception, où il faut considérer que quelques modèles ont été construits pour un type particulier de problème et qu'ils sont difficilement adaptables à d'autres contextes.

- Il faut écarter l'idée de trouver le meta-modèle des modèles de composants, étant donné que les modèles d'exécution et le modèle NF sont des modèles ouverts à tout type de structure et comportement. Cependant, on a la possibilité, avec cette vision uniforme des modèles de composants, de construire un *framework* générique pour supporter la création de nouveaux modèles.

- Les comparaisons générales et parfois simplistes qui abondent dans la littérature entre composants et objets n'ont aucun sens

3. Composants comme éléments de coordination

A la fin du chapitre 2 on a montré qu'une des limitations des objets était la difficulté de gérer plusieurs mondes à la fois, sans compromettre ses propriétés d'évolution. On a proposé alors l'introduction d'un monde, au-dessus de tous les mondes d'une application, dont l'objectif est de les coordonner. Les éléments de ce monde sont les composants de coordination, et ils sont le thème de cette section.

Nous abordons ce sujet en trois étapes : d'abord nous montrons un cadre général avec les idées de base, puis, nous étudions le cas particulier du monde de la distribution (RMI [RMI], CORBA [Sie96]), et, finalement, nous étudions le modèle des EJB (*Enterprise Java Beans* [Rom99]). Étant donné que le modèle CCM [Rom99] s'inspire du modèle des EJB, nous ne traitons que le premier.

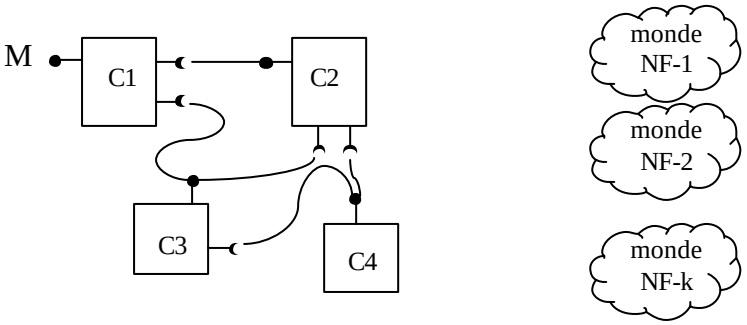
L'objectif de cette section, plus que de présenter un produit spécifique ou les solutions existantes, est d'essayer de comprendre le problème général et de montrer les composants de coordination actuels comme des solutions très partielles du problème.

3.1. Du problème global au modèle de conteneur

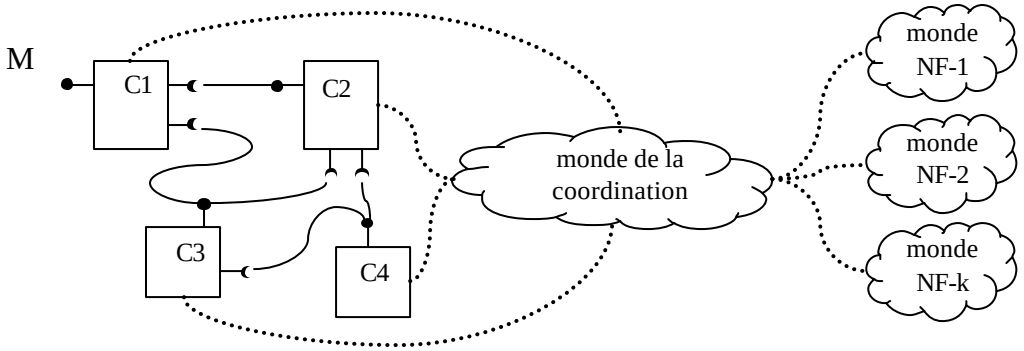
3.1.1. Définitions et espace de possibilités

Le problème à résoudre peut être énoncé comme suit :

- Il y a un modèle du monde du problème M, représenté par un modèle de composants de composition
- Il y a un ensemble de mondes non-fonctionnels MNF-i avec lesquels M doit se coordonner.



Le but du monde de la coordination est de permettre au monde M et aux mondes NF de travailler ensemble :

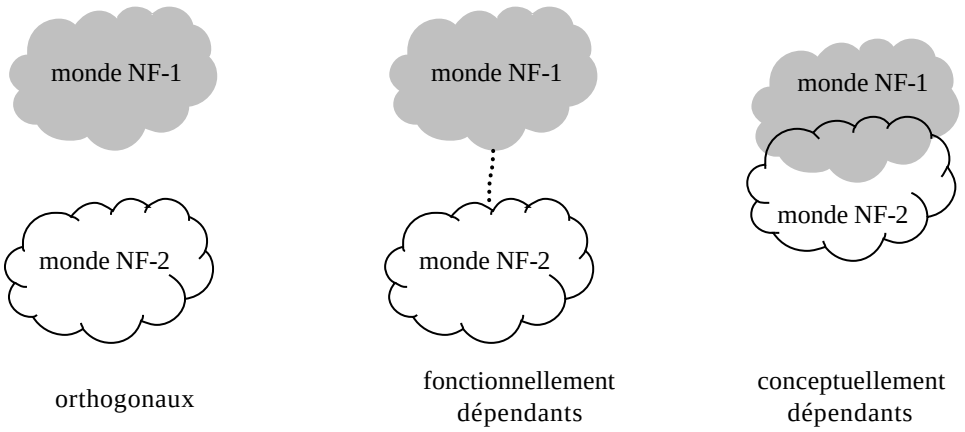


Comment est ce monde ? Et quel type d'éléments on peut trouver à l'intérieur ? La réponse à ces questions dépend en grande mesure des caractéristiques des mondes NF et de la stratégie de décomposition qu'on utilise.

Selon les caractéristiques que l'on impose aux mondes NF, la structure du monde de la coordination peut être généralisée pour gérer l'intégration d'applications (tel qu'on montre dans le chapitre 5), ou n'être utilisable que pour un sous-ensemble prédéfini de mondes NF.

D'abord, un monde non-fonctionnel MNF_i peut être actif ou passif. Il est actif s'il peut changer d'état sans l'intervention du monde du problème. Dans ce cas, le monde de la coordination est responsable de réagir et de rendre cohérent M et tous les autres mondes NF avec les changements de MNF_i. Si MNF_i est passif, il n'a pas d'initiative, et son évolution dépend toujours de l'évolution de M. Dans la suite de ce chapitre, nous allons supposer que les mondes NF sont passifs. Le cas des mondes actifs, plus près de la problématique de l'intégration d'applications, sera abordé dans le chapitre de fédérations.

Le deuxième point important est de définir s'il y a des relations (conceptuelles ou fonctionnelles) entre les divers mondes NF.



Le cas le plus simple est de supposer que les mondes sont orthogonaux. Ceci veut dire qu'ils ne sont pas connectés directement et que les relations de coordination avec le monde M peuvent être composées par simple union, car elles n'interfèrent pas. C'est le cas des mondes qui ne partagent pas de concepts, et qui peuvent travailler en même temps avec M sans problème.

Une autre possibilité sont les mondes fonctionnellement dépendants, qui possèdent des connexions directes entre eux, leur permettant de travailler ensemble, sans utiliser le monde M comme moyen de coordination (ils partagent un fichier, par exemple). C'est le cas des mondes NF, disjoints ou pas du point de vue conceptuel, mais liés dans leur fonctionnement par le partage de données ou de contrôle. Ces mondes n'ont pas besoin de M pour travailler ensemble, et la relation de M avec l'un de ces mondes, peut faire comporter l'autre monde NF comme un monde actif.

La troisième possibilité sont les mondes NF fonctionnellement indépendants et conceptuellement dépendants, où il n'y a pas une connexion directe (les mondes ne se connaissent pas), mais qui partagent des concepts qui doivent être gérés de façon uniforme.

Dans ce cas, la gestion de la cohérence des concepts partagés entre les deux mondes NF doit se faire à travers M.

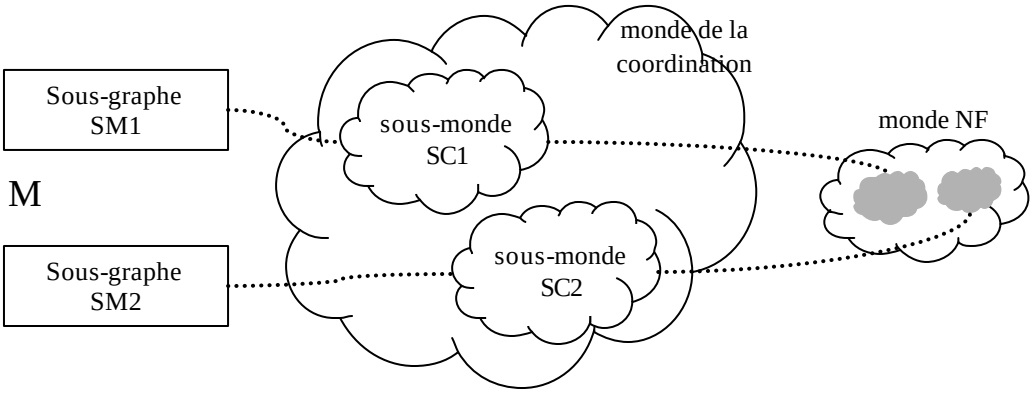
Dans la suite du chapitre nous allons supposer que les mondes sont orthogonaux. Les cas des mondes fonctionnellement dépendants ou conceptuellement dépendants seront traités plus tard dans ce document.

Il est aussi important de savoir, au moment de définir le monde de la coordination, s'il doit gérer un nombre ouvert et indéfini de mondes NF, ou s'il n'est concerné que par un ensemble donné de mondes NF, dont les caractéristiques et les relations sont connues à l'avance. Cette décision nous amène à gérer soit un modèle ouvert et général de coordination, soit un modèle fermé prédéfini.

Dans le modèle des EJB, tel qu'on le montrera plus tard dans ce chapitre, il y a trois mondes NF prédéfinis (persistance, sécurité et distribution), qui sont passifs et orthogonaux. Son modèle de coordination est alors fermé. Dans la suite de cette section, nous allons travailler sur l'idée d'un ensemble ouvert de mondes NF.

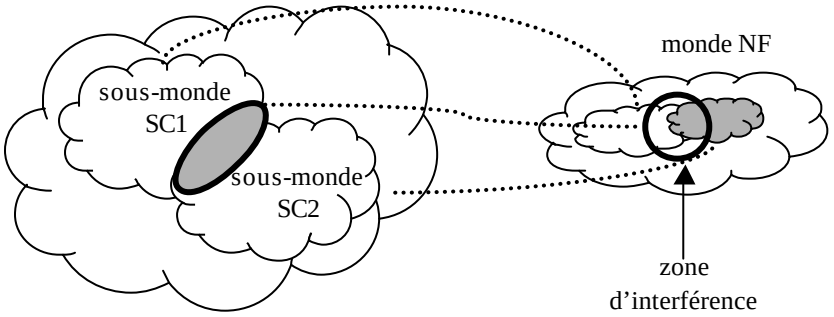
Avant de continuer avec les hypothèses de travail des composants de coordination, nous voulons introduire le concept d'isolement non-fonctionnel. D'abord, nous définissons un sous-monde du monde de la coordination, comme une structure capable de coordonner le comportement d'un sous-graphe de M avec les mondes NF. Nous allons utiliser la notation MC(SM) pour dire que le sous-monde de la coordination MC est chargé de coordonner le sous-graphe SM de M, avec les mondes NF.

Nous disons, alors, que deux sous-mondes disjoints SC1(SM1) et SC2(SM2) du monde de la coordination sont isolables dans un monde NF (dit NF-isolables), s'il n'y a aucune relation, ni interférence possible, dans la coordination de SM1 et SM2 avec le monde NF. Ceci équivaut à dire que ce monde NF peut être traité comme deux mondes NF orthogonaux du point de vue de M.

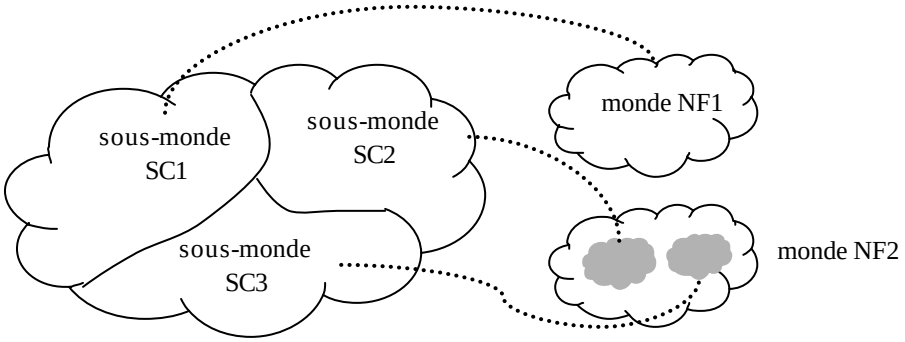


Un exemple d'isolement NF se trouve fréquemment dans la relation avec le monde de la persistance. Si on trouve deux sous-graphes de M dont la persistance puisse se gérer de manière indépendante (le stockage en disque dur d'un des sous-graphes n'interfère pas le stockage de l'autre, parce qu'ils sont sur des tableaux ou de bases de données différents, par exemple), on dit alors que les deux sous-mondes correspondants du monde de la coordination sont isolables dans le monde de la persistance.

Dans la même direction, si deux sous-mondes de M ne sont pas NF-isolables, il faut prendre en compte la gestion de l'espace d'intersection, dans la coordination de chacun de ces sous-mondes avec le monde NF, tel qu'on l'illustre dans la figure suivante :



Compte tenu du fait que le problème qui nous concerne actuellement est de trouver une structure adéquate pour le monde de la coordination, nous pouvons dire que la solution, la plus naturelle (et surtout la plus simple), serait alors de trouver une partition de M, afin de traiter la coordination de chaque sous-monde comme un problème indépendant, soit parce que chacun des sous-mondes est concerné par un monde NF indépendant, soit parce que les sous-mondes sont NF-isolables à l'intérieur des mondes NF communs.



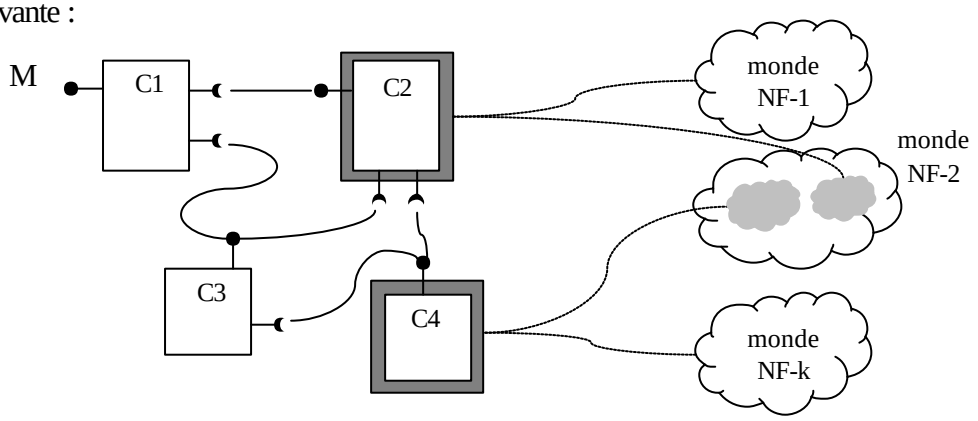
Cette solution a deux gros problèmes : d'abord, elle n'est pas généralisable, car rien ne nous laisse supposer que cette partition existe dans tous les cas. Et d'une autre part, même si cette partition existait, le fait de guider l'architecture du monde de la coordination par cette partition mettrait en danger l'évolution de l'application, puisque l'introduction d'un nouvel élément pourrait tout changer.

Une solution pour éviter ces problèmes est de partir des prémisses suivantes:

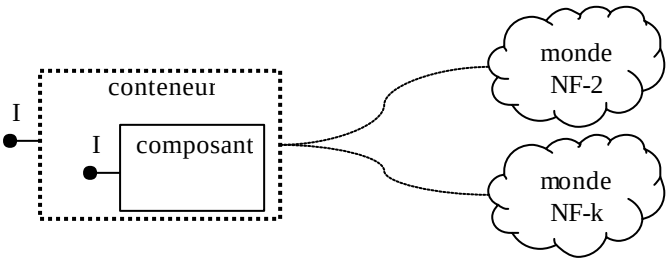
- le monde de la coordination est composé d'un ensemble d'éléments indépendants, chacun d'entre eux chargé de gérer la coordination d'un seul composant de M (i.e. sous-graphes de M de cardinalité 1). Ceci veut dire, qu'un composant de coordination "contient" un composant de composition.
- tous les composants de coordination sont NF-isolables dans tous les mondes NF

Des très fortes restrictions, certes, mais qui permettent d'aborder le problème avec une complexité raisonnable, et d'expliquer les modèles de composants de coordination existants. D'autres solutions, moins restrictives, seront étudiées tout au long des chapitres 5 et 6 de ce document.

Pour résumer, on peut dire que l'idée de fond de la solution proposée est d'introduire des éléments du monde de la coordination, pour encapsuler individuellement les composants, et gérer ainsi la synchronisation avec les mondes non-fonctionnels, tel que le suggère la figure suivante :



Chacun de ces éléments (en gris dans la figure), que nous appelons composant de coordination, assume la communication et le protocole de synchronisation avec les mondes NF concernés, sans modifier les contrats fonctionnels du composant (de composition). Dans la suite, et pour simplifier un peu la présentation, nous allons utiliser le terme "conteneur" pour faire référence à un composant de coordination, et "composant", pour les composants de composition.



L'avantage de cette vision est que, étant données les hypothèses de mondes orthogonaux et de NF-isolement dans tous les mondes, chaque conteneur est complètement indépendant, et n'a qu'à se soucier de son composant, et de gérer ses relations avec ses mondes NF. Le reste du monde de la coordination n'existe pas pour lui. Il est à noter que cet avantage est aussi son principal inconvénient, au moment de vouloir généraliser la solution.

La solution, telle qu'elle a été présentée, peut être implémentée de diverses façons, en impliquant un grand nombre de décisions sur plusieurs axes, qui seront abordés ci-dessous. Dans la plupart des cas, on montrera que si l'on veut arriver à une solution utilisable au niveau commercial, il faut aussi imposer des contraintes aux composants.

3.1.2. Implémentation du conteneur

Il y a différentes façons d'implémenter le conteneur. Nous commençons par cet aspect, car il est l'un des plus simples, et nous voulons éliminer de la discussion qui suit tout ce qui concerne une implémentation concrète.

Pour accomplir sa mission, le conteneur doit pouvoir, entre autres :

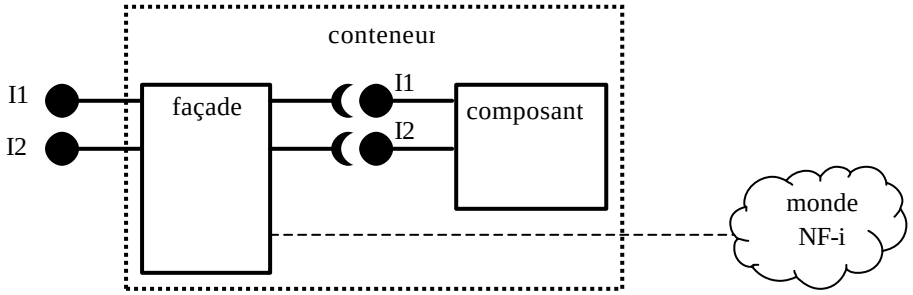
- Gérer le cycle de vie de son composant
- Intercepter tous les appels vers les méthodes du composant

- Interagir avec le modèle NF du composant (voir §2.3)

La première condition implique que le conteneur doit contrôler les processus de création et destruction d'instances de son composant pour les rendre compatibles avec les mondes NF associés. Pour faire cela, il doit pouvoir accéder au modèle NF de son composant pour obtenir les informations concernant la création d'instances, et il doit prendre le contrôle de l'exécution à chaque demande de création d'une instance du composant.

Pour intercepter les méthodes vers le composant, il y a au moins trois façons de faire : le rajout d'un mandataire (patron *proxy*), la modification du code du composant (au niveau source ou binaire), et la modification de l'interpréteur du langage. Ce sujet sera traité dans le chapitre suivant.

Pour l'instant, nous introduisons une syntaxe graphique pour montrer la structure logique d'un conteneur, en faisant abstraction des implémentations possibles.

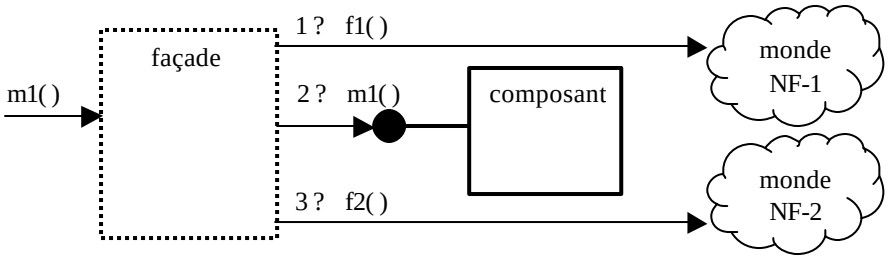


Le concept de façade ne correspond pas nécessairement à un mandataire, mais à une implémentation quelconque pour l'interception des appels des méthodes.

3.1.3. Niveaux de composition

Il y a deux niveaux de composition possibles : la composition boîte noire et la composition boîte grise.

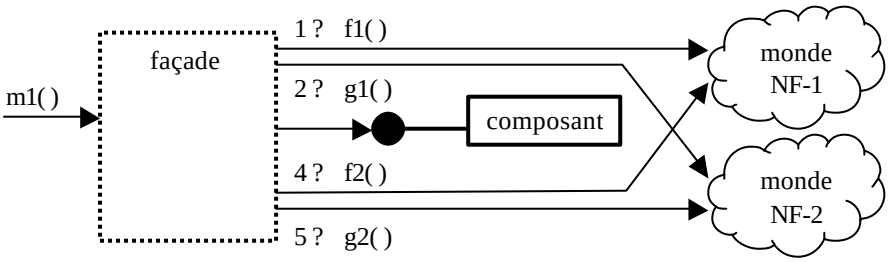
Dans la composition boîte noire, le composant (et en général le monde M) ignore complètement l'existence des mondes NF. La seule composition possible se fait en rajoutant des appels aux mondes NF avant et / ou après la méthode fonctionnelle.



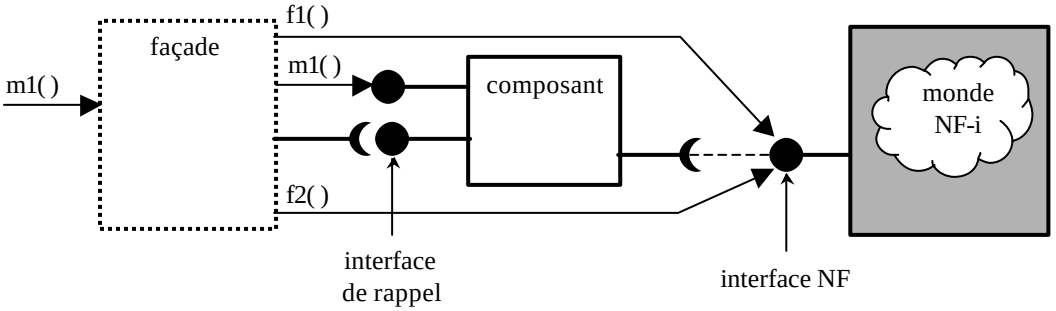
Cette approche a l'avantage de faciliter la réutilisation du composant dans plusieurs contextes non-fonctionnels possibles (il est possible de supprimer ou d'ajouter un monde NF à l'insu du composant). L'inconvénient est que ce niveau de composition n'est pas suffisant pour certains mondes, soit pour des problèmes de performance, soit parce que le

monde NF et le monde M ont des relations plus profondes que celles qu'on peut exprimer avec une boîte noire. On reparlera de ce type de relations à la fin du chapitre 4, au moment de comparer les compositions boîte noire et grise des composants, avec la composition boîte blanche de la programmation par aspects.

Étant donnée l'hypothèse que les mondes sont orthogonaux, la composition de la coordination se fait par simple union des appels de synchronisation avant et après la méthode fonctionnelle.



Dans la composition de niveau boîte grise, le composant est conscient de la présence des mondes NF et pour cette raison (1) il implémente pour eux un ensemble d'interfaces (appelées interfaces de rappel (*call-back*) et / ou (2) il utilise en interne les interfaces fournies par les mondes NF. Il faut noter que les interfaces de rappel sont normalement utilisées de manière indirecte par les mondes NF, à travers la façade du conteneur.



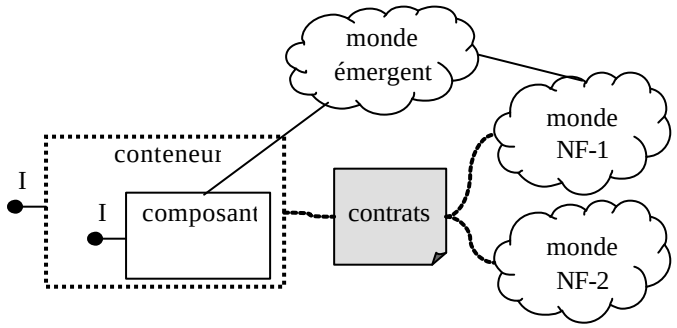
Avec ce niveau de composition on peut exprimer des relations de synchronisation un peu plus «profondes » entre M et les mondes NF, mais aux dépens de la pollution du code du composant et de ses capacités d'évolution et d'adaptation. Les EJB utilisent une composition de niveau boîte grise avec des interfaces de *call-back*, mais sans des appels directs du composant vers les interfaces des mondes NF (dans les *persistence-container-managed* EJB), ce qui semble une bonne solution de compromis.

3.1.4. Types de composition

Le thème de la composition sera traité en détail dans le chapitre 5 de ce document. Pour l'instant on peut dire qu'il y a deux types de composition : la composition syntaxique et la composition sémantique.

Dans la première, la composition se base seulement sur la correspondance des signatures des méthodes, et la composition se réduit à appeler les méthodes avec des valeurs du type adéquat pour chaque paramètre.

Dans la composition sémantique, deux éléments additionnels peuvent surgir : (1) les contrats (protocole et état) et (2) un ensemble de caractéristiques qui n'appartiennent ni au monde du problème ni au monde non-fonctionnel, mais à la relation entre les deux : les mondes émergents.



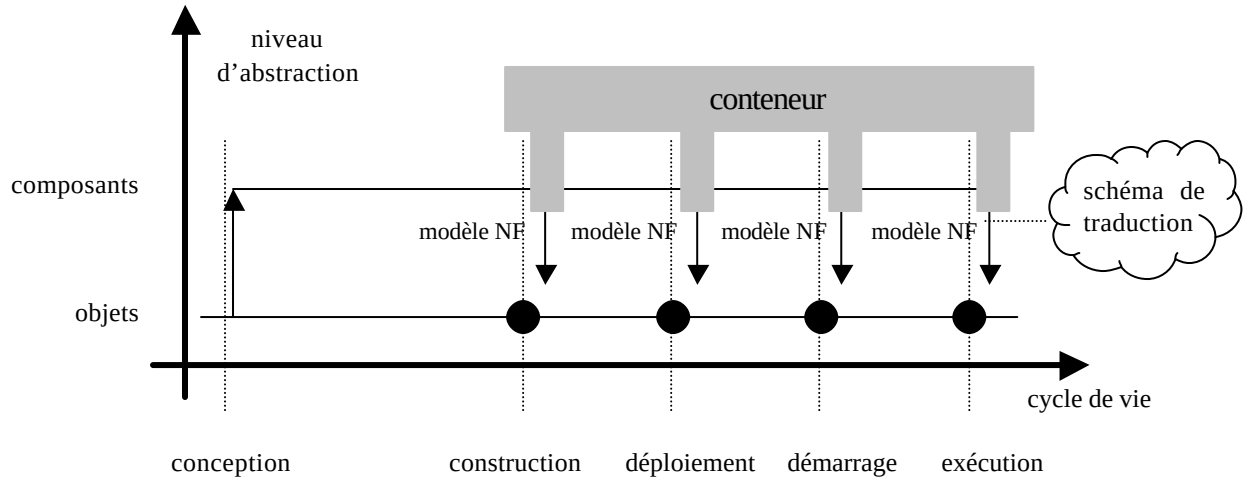
Un contrat correspond au concept de contrat d'interaction présenté en [BBB00], qui explique la mécanique de l'interaction entre tous les participants, les règles de jeu et les compromis de chacun envers les autres. Un type particulier de contrat sont les transactions. Les fédérations de composants vont utiliser les contrats comme l'un des mécanismes de coordination. Dans le langage de programmation NF des conteneurs EJB, par exemple, il est possible de définir la façon par laquelle le conteneur peut participer dans les contrats transactionnels avec le monde de la persistance.

Les "mondes émergents" ne seront pas présentés dans ce chapitre.

3.1.5. Composition non-fonctionnelle

Tel qu'on l'a présenté dans la section 2.3 de ce chapitre, le modèle NF d'un composant exprime toutes les caractéristiques permettant de concrétiser le modèle logique du composant. Dans le cas le plus simple, la seule interaction entre le conteneur et le modèle NF est celle qui permet au conteneur de récupérer les informations concernant la création d'instances du composant.

Si le modèle NF du composant est plus complexe, il est parfois indispensable que le conteneur participe à toutes les étapes de concrétisation. Ceci veut dire, que le conteneur doit pouvoir s'intégrer au niveau non-fonctionnel dans plusieurs étapes du cycle de vie du composant, et non seulement dans la création de ses instances.



Bien que cette idée d'un conteneur existant durant tout le cycle de vie du composant (et non seulement lors de son exécution) soit intéressante, nous ne l'avons pas exploitée, car cela impliquerait de considérer des domaines jusqu'à maintenant ignorés, comme par exemple les modèles de déploiement.

3.1.6. Programmation non-fonctionnelle

La définition de toutes les caractéristiques et du comportement attendu du conteneur font partie de ce qu'on va appeler sa programmation non-fonctionnelle.

Cette programmation peut se faire à plusieurs niveaux:
- Au niveau programmation, cas dans lequel il faut implémenter physiquement le conteneur comme une classe et coder tout son comportement
- Au niveau déclaratif, si dans le *framework* de support du modèle de coordination il y a des langages de spécification NF et des interpréteurs de ces langages.
- Un mélange des deux niveaux précédents.

3.1.7. Caractérisation d'un conteneur

Avec toutes les contraintes énoncées dans les sections précédentes, on peut caractériser un conteneur CT comme le tuple :

CT = < C, { MNF-i}, MC, CT-NF >

où :

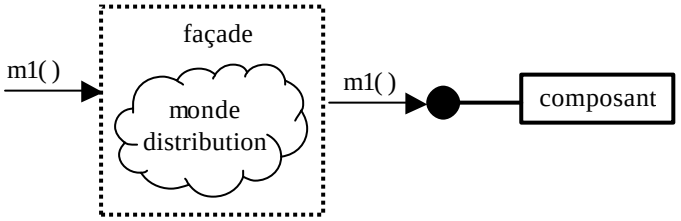
- C est un composant
- MNF-i est un monde non-fonctionnel
- MC est un modèle de coordination. Dans le cas d'une composition boîte noire, le modèle de coordination est un ensemble de triplés < A1, M, A2 >, où M est une méthode du composant, A1 est une action de coordination à exécuter avant l'appel de la méthode M, et A2 est une action de coordination à exécuter après l'appel de M.
- CT-NF est le modèle non-fonctionnel du conteneur. C'est-à-dire, l'ensemble des informations nécessaires pour sa concrétisation et exécution.

Si les mondes NF sont prédéfinis et ses relations sont préétablies, le modèle de coordination est figé et inclus dans l'implémentation du conteneur, comme c'est le cas des EJB. Sinon, l'implémentation du conteneur doit être capable de consulter son modèle de coordination pour ajouter les actions avant et après les appels des méthodes dirigés vers le composant. Dans le chapitre prochain, nous montrerons un travail qui utilise le modèle de tissage de la programmation par aspects pour exprimer le modèle de coordination d'un conteneur [Duc02].

3.2. Le monde de la distribution

Avant de présenter le modèle de composants des EJB, il faut traiter à part le monde de la distribution, car ce monde n'a pas les mêmes caractéristiques que les mondes NF sur lesquelles sont basés les modèles de coordination. En particulier, parce qu'au lieu d'une composition avec une fonctionnalité fournie par un autre monde, dans la distribution la

principale responsabilité de la façade du conteneur est de rendre transparente la situation physique du composant au client.

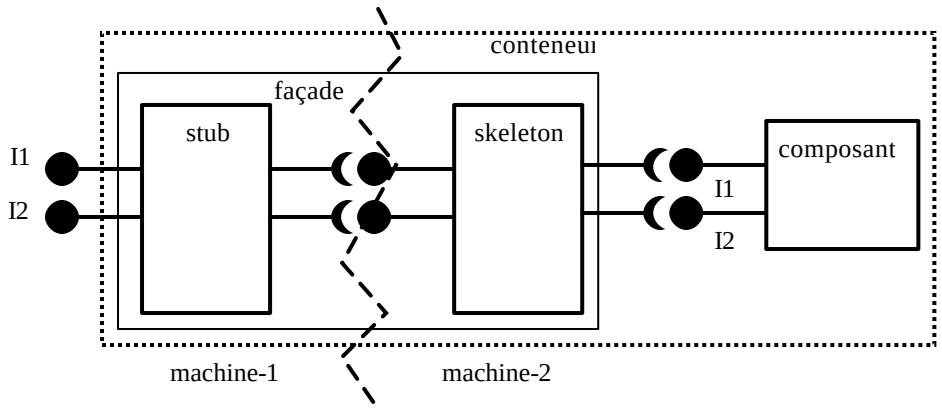


Le problème central du monde de la distribution est qu'il ne suffit pas de faire un appel à une interface d'un monde NF pour définir la localisation physique d'un composant, mais qu'il faut résoudre tous les problèmes technologiques liés au fait qu'un composant se trouve sur une machine différente de celle qui l'utilise. Il y a, au moins, quatre problèmes additionnels : (1) faire référence à un composant sur une autre machine, (2) localiser un composant sur le réseau pendant l'exécution, (3) faire passer les appels et les paramètres du client au composant et (4) faire passer la réponse du composant au client.

Dans cette section nous étudions le monde de la distribution, et montrons les modèles CORBA et RMI comme des modèles de composants dédiés à la gestion d'un seul monde non-fonctionnel : la distribution. Chaque modèle utilise une technologie différente pour résoudre les quatre points mentionnés dans le paragraphe précédent. On peut trouver de bonnes comparaisons entre ces deux modèles en [OH98].

CORBA et RMI sont reconnus comme les premiers modèles de composants, possiblement parce que c'était le premier cas dans lequel il était obligatoire de découpler l'interface de l'implémentation. Mais, il est clair que dans son intention, la composition (ni la coordination) n'étaient pas son principal souci, puisque dans les deux cas, le code de l'application termine fortement pollué avec tous les aspects de la distribution, ce qui rend difficile l'adaptation et l'évolution.

En général, les conteneurs de gestion du monde de la distribution ont la structure suivante :



La façade implémente le patron broker [BMR96], ce qui permet de rendre transparent au client la position physique du composant. Du point de vue technologique, CORBA et RMI utilisent des techniques très différentes, mais du point de vue fonctionnalité (découplage interface - implémentation et distribution "transparente"), ils sont équivalents.

3.2.1. Le modèle de composants de CORBA

Les principales caractéristiques du modèle de composants CORBA 2.0 (*Common Object Request Broker Architecture*) [RM97, Sie96] sont les suivantes :

- Le modèle a été proposé par l'OMG (Object Management Group) au début des années 90.
- Les sources du conteneur (*stub* et *skeleton*) sont générés par le *framework* à partir d'une description des interfaces exportées, faite avec un langage neutre (CORBA-IDL [OH98]).
- Le programmeur (par héritage ou composition) assemble à la main son composant avec le conteneur.
- Il n'y a pas de modèle NF déclaratif : la solution des problèmes liés à la distribution (localisation, publication, etc.) se fait dans la programmation, lors de la phase de codage. Par conséquent, le code du composant est entremêlé avec le code pour résoudre quelques-uns des problèmes du monde de la distribution. Dans la syntaxe des appels aux méthodes du composant, la distribution est transparente.
- Le composant peut être écrit dans des langages de programmation différents, si le *framework* du modèle les supporte.
- Le modèle d'exécution de CORBA propose un service de localisation d'objets distants, étant donné un nom symbolique.
- CORBA spécifie un ensemble de services additionnels (COS Services [OMG95]) utilisables par les composants, permettant de gérer plusieurs types de problèmes (transactions, déclaration d'interfaces, etc.). Toute la gestion de la cohérence entre ces divers mondes est la responsabilité du programmeur.
- Le protocole de communication est IIOP (Internet InterORB Protocol) [OH98].

3.2.2. Le modèle de composants RMI

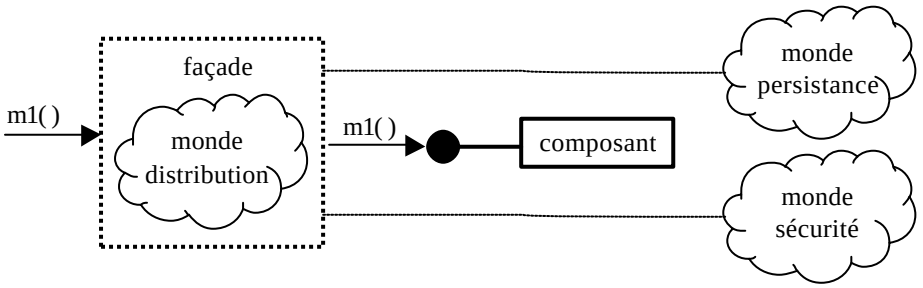
Les principales caractéristiques du modèle de composants RMI [Gro01, PM01] sont les suivantes :

- Le modèle a été introduit par Sun Microsystems en 1997, avec jdk 1.1., et il utilise comme protocole JRMP (Java Remote Method Protocol).
- Dans la définition des interfaces, il faut déclarer explicitement qu'elles vont être implémentées de façon distante (interface `Remote`).
- Le code du composant est écrit par le programmeur par héritage d'une classe du *framework* du modèle.
- Le *framework* génère le conteneur à partir du composant (compilateur rmic), sous forme de fichiers exécutables par la JVM.
- Il n'y a pas de modèle NF déclaratif : la solution des problèmes liés à la distribution (localisation, création, etc.) se fait au niveau de la programmation. On retrouve les mêmes problèmes de pollution de code mentionnés dans le modèle CORBA.
- RMI est fortement basé sur la capacité de Java de sérialiser les objets et de les envoyer de cette façon par un *socket*.

Avec les mêmes idées de base, mais en changeant de protocole de communication entre le *stub* et le *skeleton* par celui de CORBA (i.e. IIOP), on trouve le modèle RMI-IIOP [Sun-IIOP].

3.3. Le modèle de composants des EJB

Le modèle de composants des EJB (*Enterprise Java Beans*) [Rom99, Mon01] a été introduit par Sun Microsystems, vers la fin des années 90, comme réponse aux problèmes de développement d'applications industrielles. La version actuelle est la 2.0. Les EJB sont destinés à la construction d'applications commerciales typiques, où la problématique concerne, notamment, la persistance des donnés et la distribution. En gros, on peut résumer son modèle avec la figure suivante :



où :

- les aspects du monde de la distribution sont traités de manière uniforme, décrits au niveau déclaratif, hors du code (dans le descripteur de déploiement)
- les mondes NF de la persistance et de la sécurité sont gérés aussi au niveau déclaratif (la gestion de la sécurité n'est que partiellement spécifiée dans le modèle)

Le modèle se base aussi sur les prémisses suivantes :

- Les mondes NF sont prédéfinis, orthogonaux et passifs
- Le conteneur est implémenté avec un mandataire
- La composition est syntaxique, de niveau boîte grise
- Il y a une relation 1-1 entre conteneurs et composants
- Les conteneurs sont isolés du point de vue non-fonctionnel

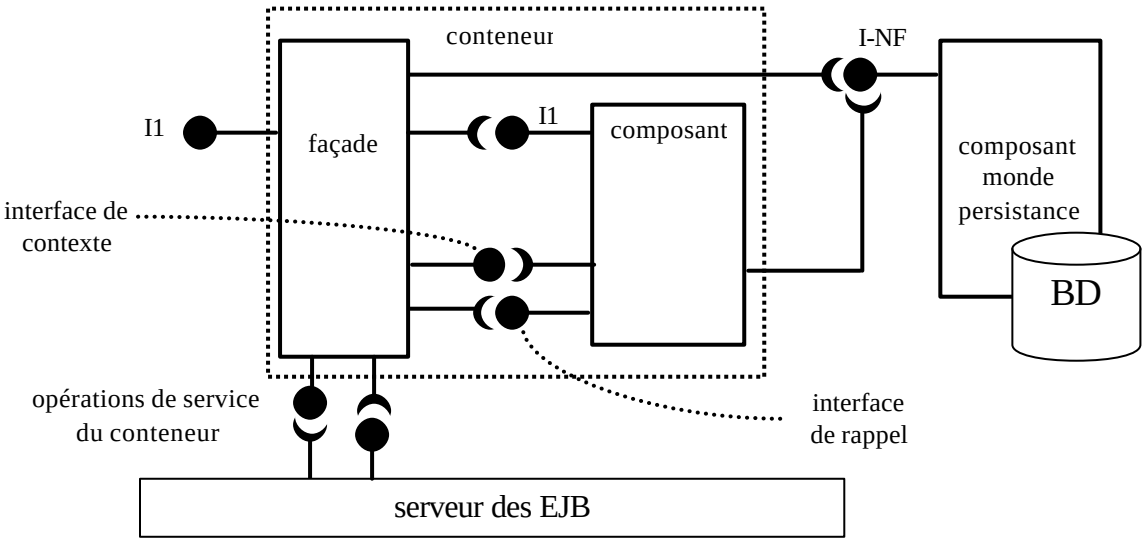
3.3.1. Le modèle de distribution

Le composant peut s'exécuter de façon distribuée. Les caractéristiques de la distribution sont décrites dans le modèle NF du conteneur. Les classes de tous les paramètres de toutes les méthodes de toutes les interfaces doivent être sérialisables (ils vont être passés comme argument entre le client et le composant en utilisant le protocole RMI-IIOP).

3.3.2. Le modèle de persistance

Dans cette section, nous nous concentrons sur le monde de la persistance et sa relation avec le conteneur, peut-être l'aspect le plus innovant du modèle. Nous nous limitons à la présentation des *entity beans*, où nous pouvons trouver tous les éléments du modèle de coordination.

Graphiquement, on peut visualiser l'architecture interne (les relations et l'assignation de responsabilités) avec la figure suivante, qui sera expliquée par la suite :



- Le conteneur "exporte" la même interface que le composant (I1), laquelle est implémentée par délégation sur le composant

- Le composant implémente une interface de rappel (call-back interface) dont la fonctionnalité est définie par l'interface `EntityBean`. Dans cette interface il y a deux responsabilités : d'une part gérer la connexion entre le composant et sa façade (`setEntityContext`, `unsetEntityContext`), et, d'autre part, fournir les opérations de base pour la coordination avec le monde de la persistance (`ejbActivate`, `ejbLoad`, `ejbPassivate`, `ejbRemove`, `ejbStore`). Le modèle de coordination du conteneur est exprimé en termes de ces méthodes. Pour la persistance sur des bases de données relationnelles, le conteneur dispose d'une implémentation par défaut de ces opérations.

- Le conteneur implémente des services qui peuvent être utilisés par le composant (interface de contexte), dont la fonctionnalité est définie par l'interface `EJBContext`. Avec cette interface le composant a accès à des informations du contexte d'exécution (transactions, fabrique, etc.). Il faut noter que le composant est responsable de maintenir son lien avec sa façade.

- Le monde de la persistance implémente une interface (I-NF) qui peut être utilisée par le conteneur (dans les *container-managed beans*) ou par le composant (dans les *beans-managed beans*). Dans ce dernier cas, les opérations de l'interface de rappel sont implémentées par le composant en termes des opérations de l'interface I-NF.

- La façade et le serveur des EJB sont connectés par interfaces cachées au composant.

Dans le descripteur de déploiement (le langage de programmation NF des EJB) le programmeur déclare si la persistance est gérée par la façade ou par le composant, et définit les paramètres du monde de la persistance.

Etant donné que la fabrique du composant (le *home*) est aussi concernée par la persistance, elle doit aussi interagir avec le monde de la persistance pour récupérer les instances de composants déjà créés et stockés quelque part. Pour faire cela, la fabrique passe par les composants, en leur demandant de gérer le concept de clé d'accès, et de fournir des opérations de création et de récupération (e.g. `ejbCreate`, `ejbFindByPrimaryKey`, etc.)

3.3.3. La gestion des transactions

Les transactions sont un cas particulier de contrat de coordination (traités dans le chapitre 5), où chaque participant a la garantie d'obtenir les propriétés ACID [Rom99] (*Atomicity*, *Consistency*, *Isolation*, *Durability*) dans sa relation avec le monde de la persistance. La structure du contrat est définie par le modèle transactionnel utilisé (plat, imbriqué, en deux phases, etc.)

Les EJB utilisent un modèle transactionnel plat. Dans le descripteur de déploiement, le programmeur doit définir la participation du composant dans des transactions. Dans les EJB il y a 6 types possibles de participation (*`Required`*, *`RequiresNew`*, *`Mandatory`*, *`NotSupported`*, *`Supports`*, *`Never`*). Le conteneur décide à partir de cette information et de l'information contextuelle, s'il doit démarrer une nouvelle transaction, participer à une transaction déjà en exécution, etc.

Le composant a toujours accès à la transaction actuelle en utilisant l'interface de contexte de la façade (`EJBContext`). Il y a deux possibilités pour qu'une transaction échoue : soit il y a une exception du système (`EJBException`) pendant l'exécution de la méthode, soit la méthode interagit directement avec la transaction en lui demandant d'échouer.

3.4. Conclusions et discussion

Les principaux points qu'on peut conclure sur les composants de coordination sont:

- On n'est qu'au début des possibilités ouvertes par l'inclusion du monde de la coordination dans les applications. Avec les conteneurs, et l'idée de coordonner plusieurs mondes NF pour les faire travailler ensemble, on s'approche de la problématique de l'intégration d'applications [Pol01, LSH03] et des modèles de composition et coordination [PA98], où il y a aussi un énorme espace de possibilités à explorer dans les prochains ans. Notre travail des fédérations de composants se dirige vers ces deux directions.

- Les EJB sont une implémentation d'un modèle très simple de composants de coordination, avec un ensemble de mondes NF prédéfinis, et sans mécanisme d'extension à d'autres mondes. Les EJB ont le mérite de montrer un vrai langage NF, utilisé dans le descripteur de déploiement, et d'introduire une façon concrète de séparer le monde du problème des caractéristiques NF de son implémentation.

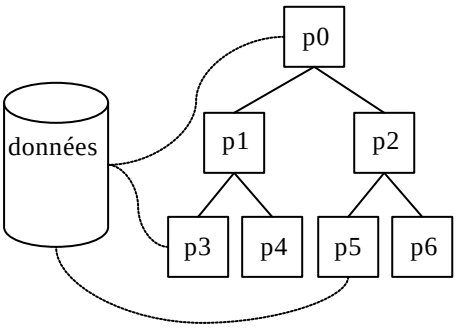
- Les contrats de composition et de coordination et les mondes émergents, dont on a parlé dans le chapitre, méritent une spéciale attention, puisque la nécessité de passer d'une composition purement syntaxique à une composition sémantique est de plus en plus claire.

- Actuellement, la technologie liée aux composants est lourde, coûteuse et complexe, et elle n'est destinée qu'à des problèmes d'une certaine envergure, dans un domaine restreint (les applications commerciales classiques).

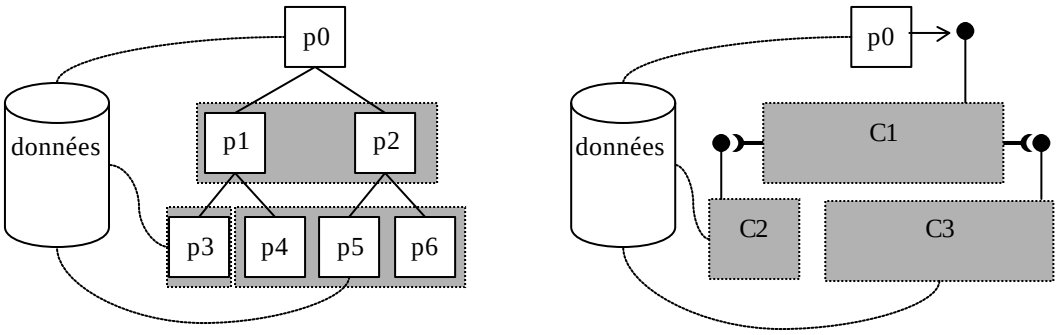
4. Composants comme des fournisseurs de services

Le troisième groupe de composants correspond à ce qu'on a appelé dans l'introduction du chapitre les composants fonctionnels. Ce type de composant est une évolution du concept de module de la programmation structurée, où l'on a découplé la description du module de son implémentation.

Typiquement, dans l'approche structurée, une application a une architecture où les données et les procédures sont séparées. Les procédures, pour leur part, sont structurées en une hiérarchie, qui représente la décomposition du problème en sous-problèmes (voir chapitre 2).

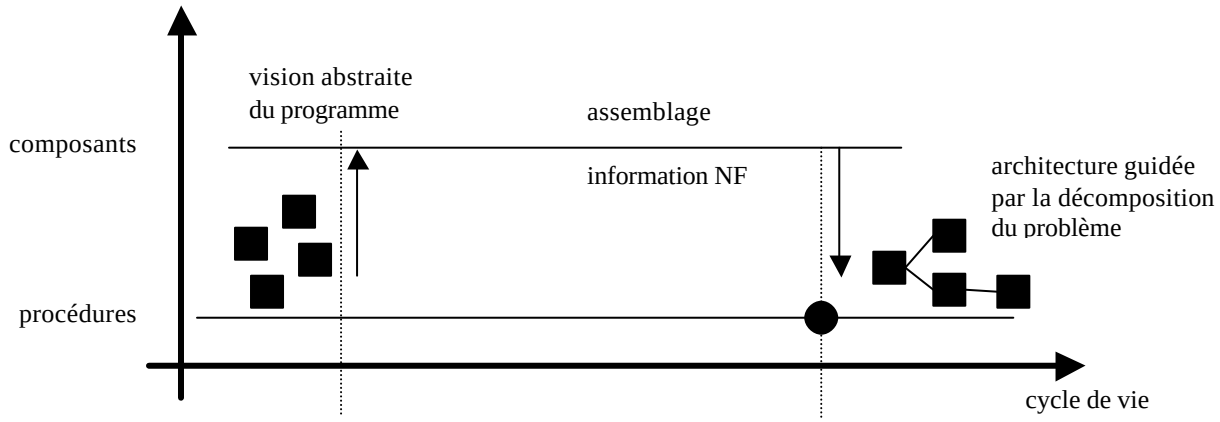


De la même façon qu'avec les objets (dans le cas des composants de composition), ici on essaie de remonter le niveau d'abstraction d'un graphe de procédures, pour exprimer l'application en termes des éléments plus abstraits : les composants fonctionnels.



Avec cette remontée de niveau d'abstraction, on peut grouper et encapsuler les procédures en unités abstraites de service. On évite ainsi les liens directs dans le code entre les différentes procédures, ce qui va faciliter l'évolution de l'application grâce à des mécanismes d'assemblage, utilisées dans les étapes du cycle de vie de l'application postérieures au codage.

On obtient des avantages similaires à ceux obtenus dans le cas des objets avec les composants de composition (i.e. décisions retardées dans le cycle de vie du logiciel), mais avec une architecture et un ensemble d'éléments de base très différents : "le même gâteau, mais pour une autre fête".



Même si du point de vue purement syntaxique les trois types de composants ont la même structure (notamment, des interfaces représentant des implémentations), il est très important de remarquer que le fait de représenter des entités différentes implique que chaque type de composant va avoir un ensemble distinct de propriétés, de mécanismes de composition, de méthodologies et d'outils.

Des exemples des composants fonctionnels sont les WEB-services [Sta02], et, en général, tout ce qui concerne la programmation orientée services [BC01]. Cependant, il doit être clair qu'il est possible d'implémenter ce type de composant sur le *framework* de n'importe quel modèle de composants de composition (e.g. CORBA, Java, OSGi, etc.), surtout sur le *framework* d'un modèle comme OSGi qui a été conçu très orienté vers les composants fonctionnels.

Typiquement, les composants fonctionnels ont les propriétés suivantes :

- Une instance d'un composant peut être partagée par plusieurs clients. Il est même normal d'avoir une seule instance de chaque composant. A la différence des composants de composition, dont les instances sont utilisées pour représenter une partie du monde, les instances des composants fonctionnels ne représentent qu'une partie de la solution du problème.

- Ils sont facilement remplaçables en exécution, car l'état de l'application n'est pas représenté dans le composant. Si quelqu'un d'autre fournit le même service, le client peut changer sa connexion pendant l'exécution (si les deux instances de composant utilisent la même source de données, bien entendu).

- Plus qu'une mécanique de création (e.g. une fabrique), le modèle de composants fournit une mécanique de localisation (e.g. un système de registre).

Un bon exemple de mécanisme de composition des composants fonctionnels est BPM4WS (*Business Process Management for Web Services*) [WC02], qui utilise un *workflow* pour composer, à haut niveau, un groupe de services.

Beaucoup des discussions qu'on trouve dans la littérature surgissent du fait que, à cause des similitudes syntaxiques, les caractéristiques des trois types de composants sont entremêlées.

5. Conclusions

La définition de composant et le cadre conceptuel présentés dans ce chapitre ne contredisent pas ce qu'on trouve dans la littérature [BBB00, CHJ02, DT03, DW98, LV01, Mey99a, Szp02, SHL01, HC01, Bro96, BW98], mais, par contre, ils permettent d'expliquer et donner un fondement à plusieurs des caractéristiques qui sont accordées aux composants sans explication apparente.

Chapitre IV

Les approches non-composants

1. Introduction

Ce chapitre présente un groupe de solutions alternatives au problème de la construction d'applications par composition. Nous montrerons un troisième type de composition, appelée l'intégration, dans lequel la composition se fait directement sur la structure du programme qui implémente le modèle du monde, en non sur une abstraction, comme c'est le cas pour les composants. La programmation orientée aspects [KLM97, EAK0l, LLM99] est un exemple d'un modèle de composition par intégration.

L'objectif du chapitre est surtout de comparer l'approche des composants (composition par connexion et composition par coordination) avec les approches de composition par intégration. Nous tenterons de faire abstraction de tous les détails concrets des modèles, et d'établir ainsi leur réelle capacité pour résoudre les problèmes et, surtout, d'établir leurs limitations.

Nous utiliserons le terme "composition de niveau boîte noire" pour toutes les approches qui remontent de niveau d'abstraction avant de composer, et le terme "composition de niveau boîte blanche" pour les approches qui composent directement sur les structures des langages de programmation.

Nous commençons le chapitre avec une présentation du problème. Ensuite nous montrerons en termes abstraits les idées de base de chaque niveau de composition, pour pouvoir faire une comparaison sur les concepts et non sur la mécanique sous-jacente.

Une section du chapitre sera dédiée à la présentation du modèle de conteneurs flexibles, un modèle de composants de coordination qui permet l'extension de son modèle d'exécution avec un mécanisme d'intégration basé sur l'idée des aspects.

2. Composition boîte noire et composition boîte blanche

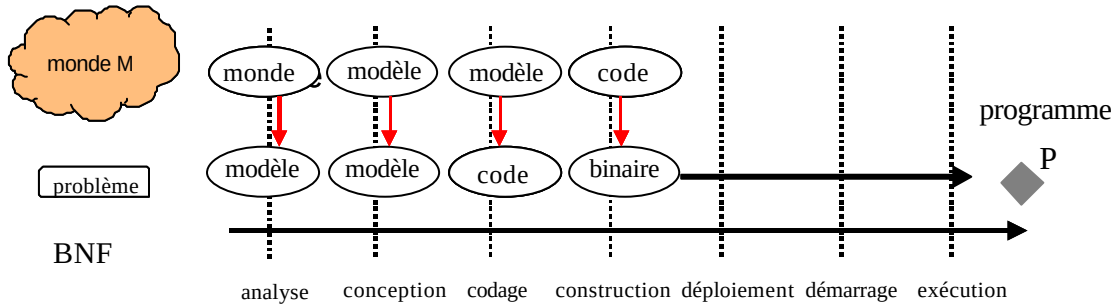
Pour faciliter la comparaison des approches, nous allons adapter un peu la terminologie, tout en essayant de garder l'esprit des concepts.

2.1. Définition du problème

Si nous considérons que le modèle d'objets représente une bonne solution aux problèmes d'évolution des logiciels (dans cette partie de la présentation nous allons supposer qu'il y a un seul monde concerné), il est possible d'affirmer que le seul problème ouvert est la gestion (inclusion et maintenance) des caractéristiques non fonctionnelles dans les programmes (au niveau de conception et d'implémentation).

Les méthodologies objets prennent en compte un modèle du monde du problème et une solution à ce problème, et définissent l'architecture du programme correspondant. Le seul ennuie est que les besoins non-fonctionnels ont été traduits en terme de code et intégrés en de nombreux endroits du programme.

Graphiquement, on pourrait imaginer le processus de développement d'un logiciel de la manière suivante :



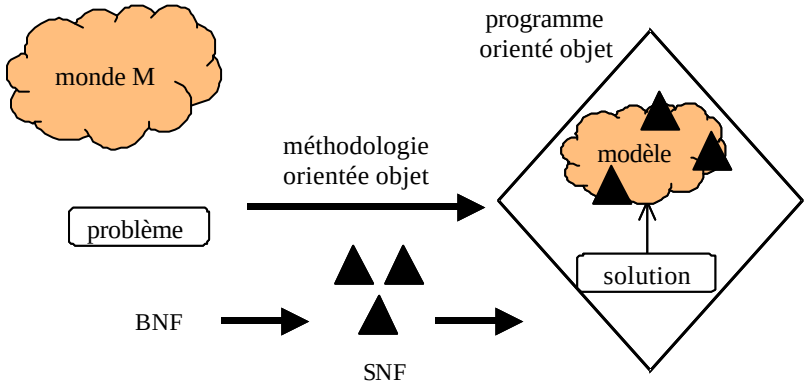
En entrée, on a un monde du problème, un problème et un ensemble de besoins non-fonctionnels (BNF) à satisfaire. Dans la phase d'analyse, on crée le modèle du monde. Dans la phase de conception, on raffine le modèle pour le rendre "exécutable". Dans l'étape de codage, on traduit le modèle en termes d'éléments du langage de programmation. Dans l'étape de construction, on traduit les éléments du langage en instructions interprétables par une machine. Dans la phase de déploiement, le code du programme est "installé" sur une machine particulière. Dans l'étape de démarrage, le programme reçoit un ensemble de paramètres (dans la ligne de commandes, dans des variables d'environnement, ou dans des fichiers de configuration) et démarre le code du programme. Et, finalement, dans l'étape d'exécution, le code du programme interagit avec son environnement (le client, le réseau, etc.) et exécute les instructions.

Le principal avantage de la programmation orientée objet est sa capacité de pouvoir garder une trace des éléments initiaux (i.e. les éléments du monde du problème, et le problème lui-même) tout au long du cycle de vie du programme. Cette propriété de traçabilité est la base de l'évolution des programmes orientés objet, et de son cycle de vie en spiral.

Le problème, tel qu'on l'a présenté à la fin du chapitre deux, est la gestion des BNF. Dans quelle étape introduit-t-on les éléments pour implémenter les aspects non-fonctionnels de l'application ? Où sont-ils dans le programme ? Y a-t-il une trace entre les BNF et un ensemble d'éléments du programme pour faciliter son évolution ? Ces questions n'ont pas de réponse claire, car les méthodologies ne disent pas comment incorporer les BNF dans l'architecture du programme et, en général, on n'a d'autre choix que d'ajouter le code pour gérer les aspects non-fonctionnel dans les méthodes des classes du modèle du monde, ce qui compromet toutes les propriétés d'évolution et d'adaptation. Par exemple, dans une classe du modèle du monde d'une application repartie avec une base de données de support, on va trouver des méthodes qui incluent des instructions pour la localisation des objets distants, pour la gestion des transactions, ou même des requêtes SQL. Dans ce cas, tout changement de technologie (i.e. passer de CORBA a RMI) peu impliquer la réécriture

d'un nombre imprévisible de méthodes et de classes : les aspects non-fonctionnels ont été introduits de manière "arbitraire" dans le modèle du monde.

En utilisant la même syntaxe graphique du chapitre 2, on pourrait montrer cette idée de pollution avec la figure suivante :



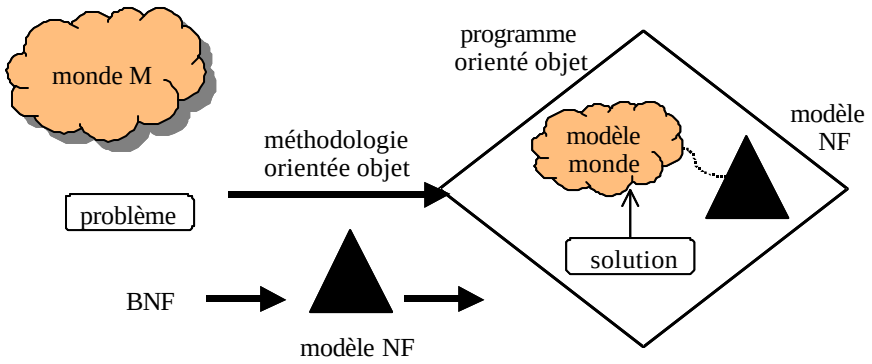
On peut voir que les besoins non-fonctionnels ont été traduits en structures du langage de programmation (la solution non-fonctionnelle - SNF) et, mélangées avec le modèle du monde pour synchroniser son exécution. Du point de vue structurel, la SNF est un ensemble de classes, et un ensemble d'instructions qui seront "tissés" avec les instructions des méthodes des classes du modèle du monde, pour obtenir ainsi un programme qui est capable de résoudre le problème, en satisfaisant à la fois un ensemble donné de BNF.

La différence de fond entre les composants (composition de niveau boîte noire) et les aspects (composition boîte blanche) est la façon de trouver une solution au problème exposé ci-dessus.

2.2. La solution idéale

Le problème peut se résumer par le fait que les aspects NF ne sont pas traités comme des éléments de premier ordre dans les méthodologies et que, par conséquent, il n'y a pas une trace entre les BNF et les éléments du programme qui l'implémentent. Les méthodologies ne disent pas comment manipuler les BNF à chaque étape du cycle de vie de développement, ni comment construire un modèle pour les représenter.

Si on applique les mêmes idées des objets à un monde non-fonctionnel imaginaire, on pourrait arriver à une architecture comme la suivante :



Dans cette architecture, chacun des éléments initiaux du problème (monde, BF et BNF) est considéré comme un élément de premier ordre, et il est représenté et implémenté par un modèle indépendant. De cette façon chaque élément initial du problème (monde, BF, BNF) pourrait être maintenu individuellement, et les problèmes d'évolution et d'adaptation seraient résolus ou, au moins, simplifiés, grâce au lien direct entre les éléments du problème, et les éléments du programme pour le résoudre.

Pourquoi cette solution n'est-elle pas utilisée ?

Il y a plusieurs réponses a cette question :

- D'abord, parce qu'on a mis dans les BNF un mélange de contraintes de tout type et, pour beaucoup d'entre elles, il n'est pas possible d'en faire une représentation en termes d'un modèle exécutable. Un BNF, par exemple, sur l'extensibilité d'un programme, est une contrainte dont le résultat n'est pas exécutable, mais qui définit une propriété de l'architecture. Un BNF de performance, fait référence à la structure d'un algorithme et non à une partie du code. On ne peut pas exécuter "la performance" séparément du modèle du monde.

- Si on limite les BNF à ceux qui peuvent se représenter dans un modèle exécutable, on trouve deux cas :

  - Les BNF dont la relation avec le monde du problème est si profonde que la seule solution est d'entremêler les instructions des deux modèles. Considérez par exemple la synchronisation d'un algorithme de plusieurs *threads* (*multi-threaded*). Il y a des cas dans lesquels le tissage ne peut même pas se faire au niveau instruction, mais il est nécessaire de le faire au niveau d'expression (ajouter une expression dans la condition d'une instruction conditionnelle). Ceci veut dire que, même si le BNF peut se représenter dans un modèle exécutable, la composition ne va pas pouvoir se faire, lors de l'exécution, par simple échange d'appels de méthodes entre les deux modèles.

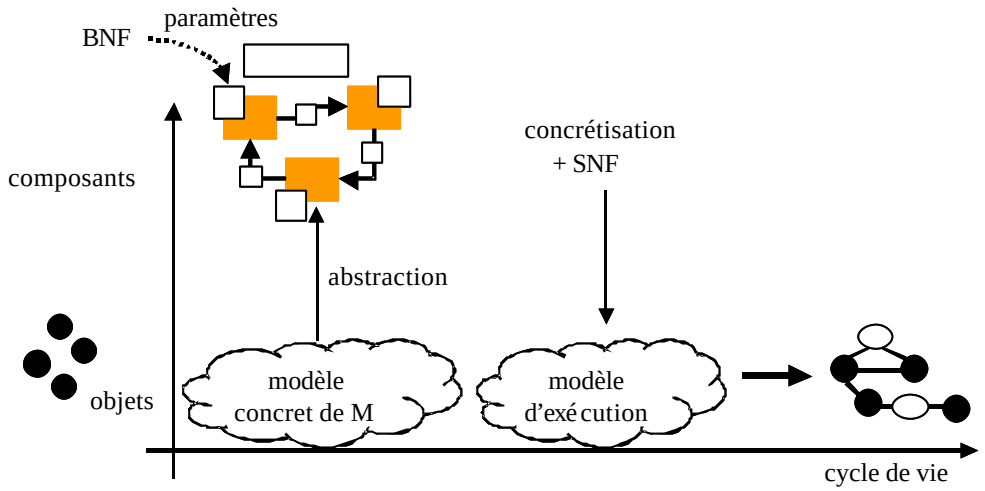
  - Dans le deuxième cas, les BNF peuvent se représenter comme un modèle exécutable, qui peut se composer avec le monde du problème lors de l'exécution. Ce dernier cas correspond au problème de coordination de mondes, discuté dans le chapitre précédent, mais il ne représente qu'une petite partie des BNF d'une application.

Nous allons supposer dans la suite qu'un monde NF tel que nous l'avons suggéré ci-dessus n'est pas généralisable, et qu'il faut trouver une manière différente d'inclure les aspects non-fonctionnels dans les programmes.

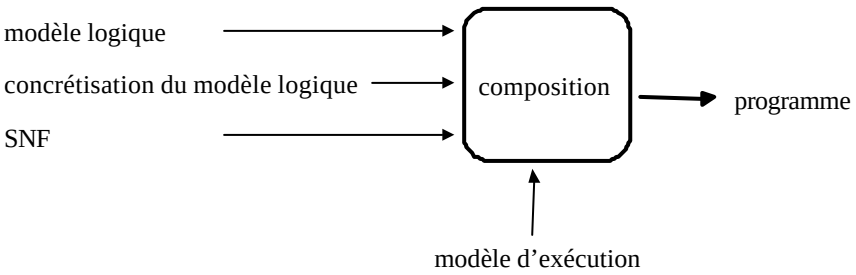
2.3. La composition de niveau boîte noire

Un modèle de composants a été défini comme étant un triplé : un modèle logique, un modèle d'exécution et un modèle non-fonctionnel. Le modèle logique permet de remonter le niveau d'abstraction du modèle du monde du problème. Le modèle d'exécution définit l'architecture finale du programme en termes des éléments du modèle logique. Le modèle

non-fonctionnel décrit la manière de concrétiser les éléments du modèle logique pour les assembler dans l'architecture du programme.

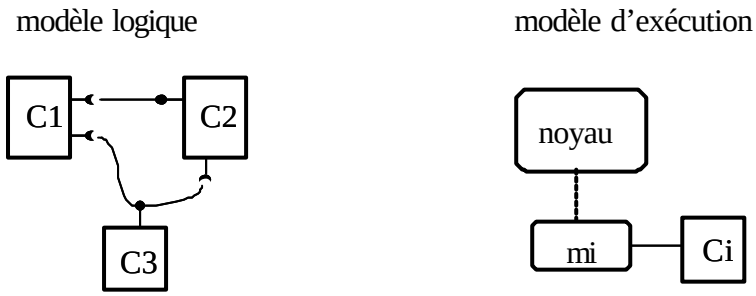


On peut imaginer la construction d'un programme sur un modèle de composants en six étapes : (1) création d'un modèle concret du monde du problème (i.e. la sortie de la phase de conception), (2) remonté du niveau d'abstraction pour définir un modèle abstrait du monde du problème (i.e. apparition des composants), (3) implémentation des éléments concrets du monde du problème (construction des classes), (4) "décoration" des éléments du modèle logique pour décrire la manière de les concrétiser, (5) construction de la solution non-fonctionnelle pour implémenter les BNF (nous reparlerons de cette étape par la suite), et (6) composition de tous les éléments précédents dans le modèle d'exécution pour obtenir un programme qui résout le problème, tout en respectant les BNF.

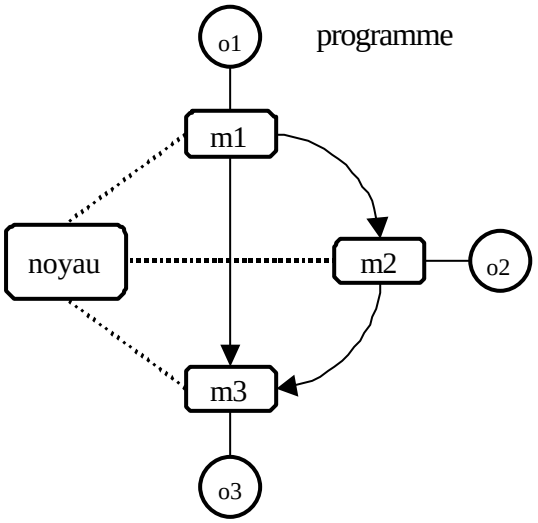


Le modèle d'exécution peut donc être considéré comme une architecture générique pour servir de base au processus de composition. Avec cette vision on peut dire qu'une SNF n'est qu'un programme écrit dans un langage compréhensible par les éléments du modèle d'exécution pour implémenter un ensemble de BNF.

Imaginez par exemple un modèle logique (trois composants C1, C2 et C3 connectés) et un modèle d'exécution (un noyau central et un mandataire entre chaque couple de composants du modèle logique) comme ceux suggérés dans la figure suivante :



Le programme résultant de la composition aurait la structure montrée dans la figure suivante, où m_k représente un mandataire et O_k l'objet (ou l'ensemble d'objets) qui implémente le composant C_k. Il est à noter qu'il y a deux types de connexions dans le programme : les connexions logiques (lignes pleines avec une direction) et les connexions du modèle d'exécution (lignes pointillées qui connectent les mandataires avec le noyau).



Dans l'exemple, une SNF pourrait comporter des paramètres pour l'instanciation des mandataires (la politique d'adaptation dynamique, par exemple), des informations pour le noyau (où trouver un serveur particulier lors de l'exécution), ou même un programme complet interprété par les éléments du modèle d'exécution (le noyau et les mandataires dans l'exemple).

De cette manière, on peut considérer le modèle d'exécution comme l'implémentation d'un monde non-fonctionnel, qui connaît la manière d'intégrer les éléments du monde du problème, et d'interpréter une SNF. Ceci nous permet d'affirmer que dans la composition de niveau boîte noire :

- Le modèle du monde (i.e. les composants) est intégré dans le modèle du monde non-fonctionnel (i.e. le modèle d'exécution)

- Le monde non-fonctionnel est prédéfini pour chaque modèle de composants, ce qui implique qu'il ne peut résoudre qu'un nombre limité de problèmes non-fonctionnels, mais que la solution peut s'exprimer dans des langages de haut niveau d'abstraction, ce qui simplifie son développement et sa maintenance.

- La composition se fait au niveau méthode, et non instruction.

2.4. La composition de niveau boîte grise

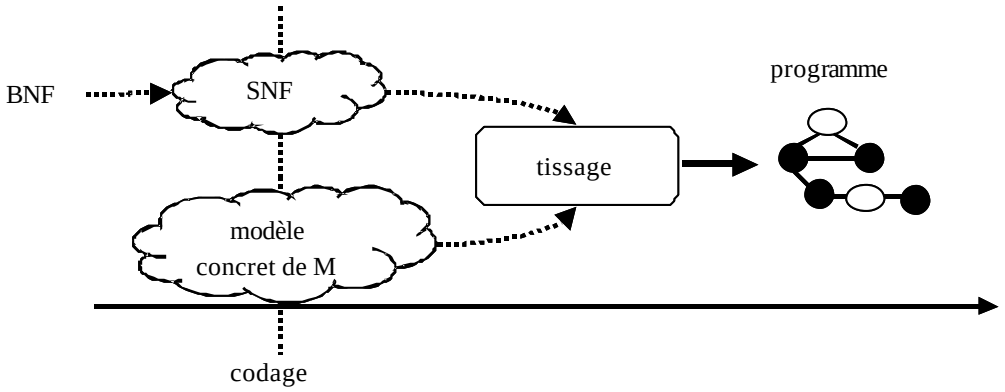
Dans la composition de niveau boîte noire les composants ne connaissent pas l'existence du modèle d'exécution, et ils sont alors réutilisables. Pour cette même raison la SNF doit s'exprimer séparément des composants ce qui facilite son adaptation.

Le problème avec les modèles de composants actuels est qu'ils ne fournissent pas un langage explicite pour exprimer la SNF, ce qui amène à construire des composants qui dans leur code interagissent directement avec le modèle d'exécution du modèle de composants pour résoudre les problèmes NF. Dans l'exemple de la section antérieure, ceci voudrait dire que dans les méthodes de l'objet o1, il y a des appels vers les services fournis aussi bien par le noyau que par son mandataire.

L'avantage est que la composition se fait au niveau des instructions (et même des expressions), ce qui permet de faire une composition beaucoup plus fine. L'inconvénient est que le code des composants est pollué et l'évolution et l'adaptation sont compromises.

2.5. La composition de niveau boîte blanche

Dans la composition de niveau boîte blanche, il n'y a pas de remontée de niveau d'abstraction, et l'inclusion des SNF se fait directement sur les structures du programme. Nous allons appeler "intégration" la composition de niveau boîte blanche.



On peut imaginer la construction d'un programme créé par intégration en trois étapes : (1) création d'un modèle concret du monde du problème (i.e. la sortie de la phase de codage), (2) traduction des BNF en solutions non-fonctionnelles (i.e. implémentation des SNF pour que le programme résultant satisfasse les BNF), (3) tissage des SNF avec le code du programme.

Il y a deux groupes de possibilités pour réaliser l'intégration :

- Par génération : les solutions se basent sur la manipulation du code source (ou du *bytecode*, dans le cas de java). Le mécanisme d'intégration génère une nouvelle application, avec les SNF intégrées en termes d'instructions additionnelles dans les classes et les méthodes du modèle du monde. Des exemples de cette solution sont la

programmation orientée aspects [KLM97, EAK0l, LLM99], AspectJ [KHH01], HyperJ [HJ], les hyperespaces [IBM, OT00, TOW99], la programmation adaptative [LOO01, LSX94, OL01], la programmation générative [CES97, EC00, LL94], les filtres de composition [ABS94, BA01, AWB93, AT98], la programmation subjective [HO93], et la machine à objets étendus [DES02].

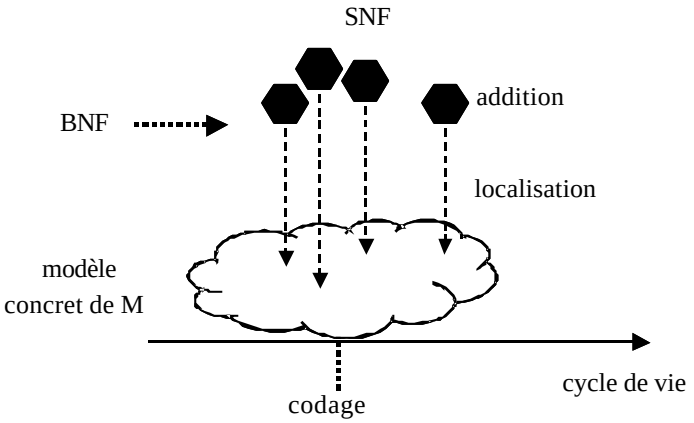
- Par modification de la machine virtuelle du modèle d'objets : au lieu de modifier le programme, qui s'exécute sur une machine virtuelle à objets, l'idée est d'agir au niveau de la machine virtuelle, et d'étendre sa sémantique pour prendre en compte la SNF. Un exemple de ce type de solution est la meta-programmation [KRB92, BH02, MMC95]. Cette approche ne peut être utilisée qu'avec des langages réflexifs comme Smalltalk [GR85] ou CLOS (*Common Lisp Object System*) [Tou90].

Par la suite nous considérerons seulement les solutions du premier groupe, et utiliserons la programmation par aspects comme l'élément le plus représentatif du groupe.

L'idée de fond des aspects est de supposer qu'une SNF (i.e. le code qu'il faut ajouter au modèle du monde du problème pour satisfaire un ensemble de BNF) peut s'exprimer comme une séquence de couples (addition, localisation), où :

- "addition" est un morceau de code qui représente une partie des instructions qu'il faut ajouter au programme qui implémente le modèle du monde (appelé dans la suite "le programme original") pour implémenter la SNF.

- "localisation" est le point dans la structure du programme original où il faut introduire le code de "l'addition".



Cette approche a les caractéristiques suivantes :

- La structure de base pour l'intégration est le programme original, auquel la SNF est ajoutée par couples (addition, localisation).

- La SNF n'est pas restreinte à un domaine non-fonctionnel spécifique.

- L'intégration entre le programme original et la SNF peut se faire à plusieurs niveaux, et dépend de la précision du formalisme de localisation. En particulier, on peut imaginer une intégration au niveau d'instructions.

Les principaux inconvénients de l'approche sont :

- Les additions et les localisations s'expriment en termes du programme original, ce qui implique une profonde connaissance de la structure de ce programme, et une lourde dépendance entre la SNF et les éléments de niveau syntaxique et lexical du programme original. Ceci implique tout type de problèmes d'évolution et d'adaptation. On ne peut pas changer le nom d'une classe du programme original, à risque de perturber la localisation d'une partie de la SNF.

- Il y a toujours un risque de conflit entre les parties de la SNF. Lors de l'évolution, par exemple, il faut connaître la localisation de toutes les additions déjà faites (et leur contenu), pour éviter les interférences entre les diverses additions, car il n'y a pas de relation conceptuelle entre les parties ajoutées (il n'y a pas de modèle de la solution à partir duquel on puisse prendre des décisions).

- Pour permette un tissage de bas niveau (au niveau d'instruction, par exemple), le système de localisation peut devenir très complexe (comment dire "ajouter dans la condition de la deuxième instruction conditionnelle de la méthode M de la classe C" de une manière simple?).

- La complexité de la SNF est très liée à la structure du programme original.

- Il est difficile de définir des méthodologies pour appuyer la construction d'applications par intégration, car tout dépend de la structure du programme original, et il n'y a aucune connaissance a priori à ce propos.

2.6. Composants avec un modèle d'exécution flexible

Plusieurs lignes de travail sont ouvertes autour des modèles de composants. En particulier, il commence à être évident que le modèle d'exécution est l'un des éléments névralgiques d'un modèle de composants. Il faut considérer surtout que les programmes écrits en utilisant un modèle de composants sont limités au monde non-fonctionnel que le modèle d'exécution implémente (sauf si les autres aspects non-fonctionnels sont gérés à la main par le programmeur dans le code des méthodes du modèle du monde).

Pour cette raison, en plus du développement de langages pour exprimer les SNF, il est important aussi de pouvoir étendre le modèle d'exécution d'un modèle de composants. Dans [Duc02] il y a un travail qui propose une manière d'étendre le modèle d'exécution, en utilisant la mécanique introduite par la programmation orientée aspects.

3. Conclusions et résumé

Dans ce chapitre nous avons montré l'intégration comme un troisième type de composition. L'intégration correspond à l'idée de construire une application par tissage entre le

programme qui implémente le modèle du monde et le programme qui implémente les besoins non-fonctionnels.

Avec l'intégration (utilisé par la programmation orientée aspects), la connexion et la coordination (utilisés par les composants), nous complétons les trois niveaux de composition, qui permettent la création d'applications par assemblage d'éléments.

Nous avons fait une comparaison des principales caractéristiques et limitations de chaque approche, et nous avons montré comment, actuellement, nous n'avons que des solutions partielles au problème, et que c'est un domaine très prometteur aussi bien pour la recherche que pour l'industrie.

Chapitre V

Composition de composants par coordination

1. Introduction

Avec ce chapitre nous commençons la deuxième partie de la thèse. Dans les trois précédents chapitres, nous avons étudié plusieurs problèmes liés au développement des logiciels. Nous nous sommes fournis d'un cadre conceptuel de travail et nous avons montré les principaux modèles commerciaux existants en utilisant ce cadre. Nous avons identifié une quantité de problèmes ouverts, qui méritent d'être étudiés. Nous avons aussi montré que le mot "composition" faisait partie des solutions proposées à plusieurs de ces problèmes, et que ce mot devait s'interpréter de façon différente dans chaque cas. En particulier, nous avons identifié trois niveaux différents de composition : l'intégration, la connexion et la coordination, comparés à la fin du chapitre précédent. De même, nous avons distingué trois types de composants : les composants de composition, les composants de coordination et les composants fonctionnels, chacun avec des caractéristiques qui font que la composition entre eux se fait de manière différente.

Maintenant, nous nous concentrons sur l'objet de cette thèse : la coordination comme mécanisme de composition.

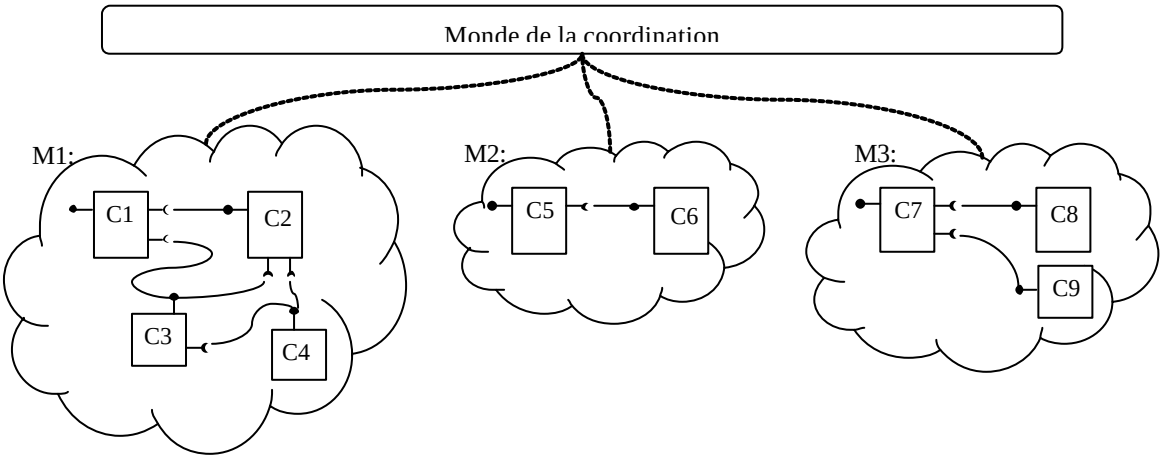
Nous procéderons en trois étapes : d'abord, dans ce chapitre, nous allons étudier chaque facette du problème de manière indépendante. Ceci va nous permettre de mieux gérer le niveau de complexité du problème global. Dans le chapitre suivant, nous allons montrer une façon d'intégrer les solutions dans une architecture que nous appellerons une fédération de composants. Dans le chapitre 7, nous montrerons l'expérimentation et les résultats pratiques obtenus.

Ce chapitre est organisé de la façon suivante : dans la section 2 nous faisons le point sur le problème que nous voulons étudier, en utilisant la terminologie introduite dans les chapitres antérieurs. Dans la section 3, nous étudions les contrats, et leur utilisation dans la problématique de la composition. Dans la section 4, nous traitons le cas particulier du monde du contrôle comme élément de composition. Dans les sections 5 et 6 nous étudions le problème de la composition du point de vue des mondes dépendants et des mondes émergents. Dans la section 7, nous étudions la gestion d'une application comme un composant. Dans la section 8, nous abordons la problématique de la composition de mondes actifs. Et finalement, dans la section 9 nous concluons et discutons les résultats.

2. Le problème à résoudre

Nous voulons étudier le problème de composition de composants par coordination, et nous voulons aussi pouvoir considérer le problème d'intégration d'applications (et d'outils) comme un cas particulier.

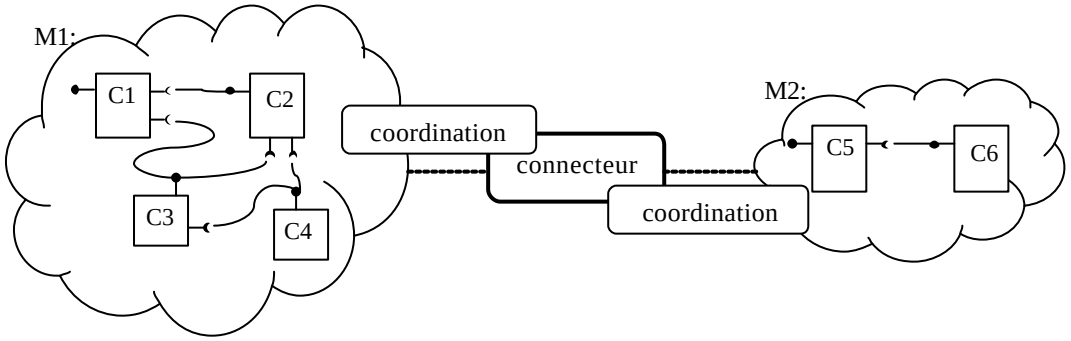
Le problème peut se résumer de la manière suivante : il y a N mondes modélisés en termes de composants, lesquels doivent travailler ensemble pour atteindre un objectif commun, sans aucun type de connexion directe (la composition par intégration et la composition par connexion sont écartées). Ces mondes peuvent être actifs, fonctionnellement dépendants et conceptuellement dépendants.



Les mondes, pour leur part, peuvent être exprimés en termes de composants de composition, ou en termes de composants fonctionnels. La seule supposition est qu'à l'intérieur de chaque monde il n'y a pas de mélange entre ces deux types de composants. Il est important aussi de dire, que les mondes peuvent connaître la présence du monde de la coordination, mais qu'ils ne se connaissent pas entre eux.

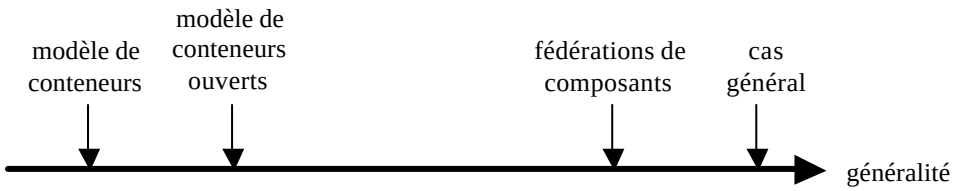
Dans la composition par coordination il y a un monde qui émerge, appelé le monde de la coordination. Pour aborder le problème de la composition par coordination, nous avons choisi d'étudier ce monde : sa structure, ses éléments, ses relations avec les mondes coordonnés, etc. Comme nous le verrons dans le chapitre 6, une fédération est une architecture pour représenter le monde de la coordination.

Cette approche se différencie de l'approche proposée par les EAI (*Enterprise Application Integration*) [FWK02, LSH03, SV02] où la coordination entre deux applications est obtenue en les connectant à travers des adaptateurs, et en distribuant les responsabilités d'interaction et de coordination entre les deux mondes. La figure suivante illustre le modèle de connexion des EAI.



Nous considérons que le problème qui nous concerne ici est complexe, sans solution simple apparente. Nous allons étudier chaque partie du problème de manière indépendante, pour essayer de proposer des solutions partielles. Ensuite, nous tâcherons de mettre ensemble ces solutions, pour obtenir un modèle multi-facettes du monde de la coordination. Nous courons le risque d'obtenir des solutions partielles "incompatibles" et impossibles de composer, ou d'arriver à un modèle trop complexe, inutilisable dans des applications réelles. C'est pour cela que dans le travail nous donnons autant d'importance aux évaluations pratiques, dont les résultats serons présentés dans le chapitre 7.

Notre travail vise le même problème des modèles de conteneurs, mais à un plus haut niveau d'abstraction et dans un contexte plus général. Dans les modèles de conteneurs la solution se base sur l'hypothèse que la coordination peut s'obtenir à partir de l'union d'un ensemble d'éléments isolables : les conteneurs. Par rapport aux travaux dans la même ligne, nous pouvons situer les fédérations de composants de la manière suivante :



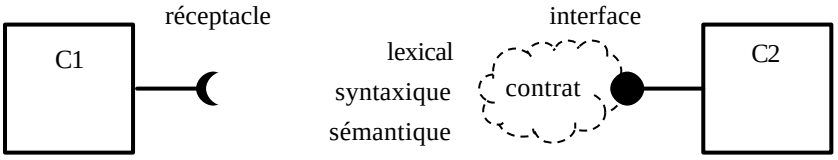
3. Contrats

Dans cette section nous allons étudier les contrats comme éléments de coordination. Pour l'instant, nous ne nous intéressons pas à la manière dont ils sont groupés (liés ou structurés entre eux) ou contrôlés (qui les déclenche et à quel moment) dans le monde de la coordination. Nous voulons ici seulement les caractériser individuellement.

3.1. Définitions et types de contrats

Un contrat est défini comme un "accord de volontés destiné à créer des rapports obligatoires entre les parties" [Dictionnaire Hachette – édition 2000].

Dans notre cas, une interface définit toujours un contrat que le réceptacle (la connexion côté client) doit respecter au moment de l'assemblage. Ce contrat a trois niveaux : un niveau lexical, un niveau syntaxique et un niveau sémantique.



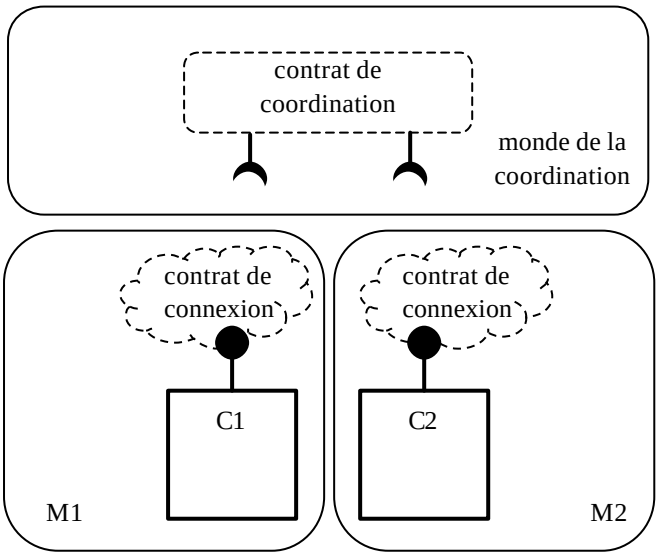
Le niveau lexical est défini par la correspondance des signatures des méthodes. C'est ce qui dans la littérature est connu sous le nom de contrat fonctionnel [BBB00]. Dans les langages de programmation, le niveau lexical correspond au concept de type, et le compilateur vérifie la compatibilité lexicale entre le réceptacle et l'interface : un réceptacle ne peut se connecter à une interface que si les types correspondants sont compatibles.

Le niveau syntaxique décrit le protocole d'utilisation des méthodes, l'ordre éventuel des appels ou les contraintes liées à l'utilisation des méthodes. Ce type de contrat est appelé aussi un contrat d'interaction [BBB00] ou un contrat de comportement [PV02, AF99]. Il ne suffit pas que les types soient compatibles, mais il est aussi indispensable de respecter une forme d'utilisation. Le niveau syntaxique peut imposer, par exemple, qu'une méthode M1 soit appelée seulement après une autre méthode M2, ou qu'une méthode M3 ne puisse être appelée que dans un état particulier du composant. Plusieurs types de formalismes ont été proposés pour modéliser ce type de contrat [FG03, FS02, MSZ01, BS02, SLM96]. Si le niveau syntaxique du contrat est exprimé explicitement, sa vérification pourrait éventuellement se faire dans le code du composant.

Le niveau sémantique explique les conséquences, pour l'état du composant, d'appeler chaque méthode. D'un point de vue objet, on peut dire que le niveau sémantique décrit le comportement du sous-monde du problème représenté par le composant. L'intention du contrat est souvent exprimée en termes d'assertions (préconditions et post-conditions) sur l'état du composant avant et après l'appel de la méthode. Il y a plusieurs formalismes pour exprimer l'intention du contrat (e.g. OCL [CW02, WK99a], Z [Dil94], etc.), destinés à documenter le composant et à guider le travail du programmeur. Dans le cas général, ces formalismes ne sont pas exécutables. Le niveau sémantique du contrat est donc matérialisé par l'implémentation du composant. Ceci a plusieurs conséquences : (1) la sémantique du contrat n'est pas complètement vérifiable, (2) l'assemblage se fait sur les intentions des contrats (donc, une personne doit toujours participer pour l'interpréter), (3) la composition de niveau sémantique (i.e. l'utilisation d'un composant C1 (le fournisseur) par un composant C2 (le client)) est câblée en dur dans l'implémentation de C2.

Ceci veut dire qu'avec les composants on peut rendre explicites les dépendances entre C1 et C2, mais qu'on est incapable de préciser la nature de cette dépendance, et que cela n'est exprimé que dans le code des implémentations. Si C1 et C2 appartiennent au même monde, on peut supposer que la sémantique de la composition est définie par la composition des comportements des sous-mondes, et que, même si la sémantique reste au niveau du code, l'évolution n'est pas compromise.

Si C1 et C2 appartiennent à deux mondes distincts, le problème est complètement différent. Nous distinguons donc deux types de contrats dans le problème qui nous concerne : les contrats de connexion et les contrats de coordination. On peut imaginer un contrat de coordination comme le suggère la figure suivante :



On peut faire le parallèle suivant entre les contrats de connexion et les contrats de coordination. Dans la section suivante, nous allons caractériser les contrats de coordination :

Contrat de connexion	Contrat de coordination
Les deux composants appartiennent à un même monde : le client a dans son code le niveau sémantique de la composition	Les composants qui participent appartiennent à des mondes différents : le contrat de coordination définit le niveau sémantique de la composition
Les contrats sont binaires. Même dans le cas où la cardinalité du réceptacle serait multiple, les contrats sont toujours définis un à un.	Les contrats ne sont pas nécessairement binaires. Il est courant que plusieurs composants participent dans un même contrat.
Les contrats sont fermés. Il n'y a que les deux participants.	Les contrats peuvent être ouverts, dans le sens où des participants additionnels doivent pouvoir arriver lors de l'exécution du contrat.
Un contrat n'a pas d'état. Une fois la connexion faite, l'état du contrat est implicite dans l'état des participants.	Un contrat a un état, additionnel à l'état des composants qui participent, et qui doit être géré par le monde de la coordination.
Le contrat doit toujours réussir.	Il existe le concept d'échec dans les contrats de coordination, non nécessairement lié à une exception ou à une erreur d'exécution. Simplement si les participants n'arrivent pas à se coordonner, le contrat échoue et le monde de la coordination doit réagir en conséquence.
Les contrats sont passifs, dans le sens où les deux participants sont prédéfinis	Les contrats peuvent être actifs (le contrat va chercher les participants à partir de sa description) ou passifs (les participants

	arrivent au contrat, et il attend que tous soient présents pour s'exécuter)
Les contrats sont statiques. Le contrat est le même tout au long de l'exécution d'une application	Les contrats peuvent être dynamiques. Un contrat peut s'adapter lors de l'exécution, en tenant compte du contexte. Il est même envisageable un processus de négociation entre les participants, pour définir une structure ou un ensemble de participants.
Les contrats ne sont pas une unité de composition	Si l'on veut composer deux mondes de la coordination, il est nécessaire de pouvoir composer les contrats.
Un contrat ne peut pas faire référence aux aspects non-fonctionnels des participants.	Un contrat peut éventuellement faire référence aux caractéristiques non-fonctionnelles des participants. En particulier, le contrat doit pouvoir imposer des contraintes NF aux participants.

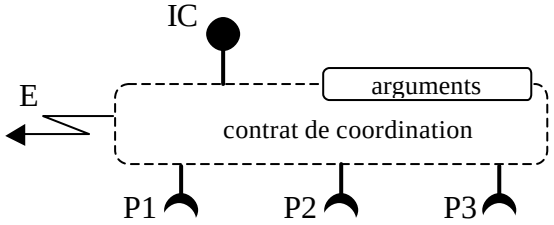
3.2. Caractérisation d'un contrat de coordination

Pour caractériser un contrat de coordination il faut considérer plusieurs dimensions, qui seront présentées ci-dessous.

3.2.1. Modèle externe du contrat

Vu de l'extérieur, un contrat de coordination (appelé simplement contrat dans la suite) est composé de :

- Un nom
- Une liste d'arguments : l'entité du monde de la coordination qui va contrôler les contrats va utiliser ces arguments pour décrire l'état de l'exécution, et pour paramétrer le contrat (i.e. décrire les participants, définir une politique, etc.)
- Un ensemble de participants abstraits (i.e. type des participants) avec un nom associé (P1, …, P3 dans la figure)
- Des événements émis par le contrat pour notifier son déroulement
- Un type de réponse : des informations pour l'entité qui contrôle le contrat, sur l'exécution du contrat (par exemple, le succès ou l'échec, les raisons d'un éventuel échec, d'autres objets nécessaires pour le monde de la coordination, etc.).
- Une interface de contrôle IC. Cette interface permet au monde de la coordination d'exécuter le contrat, de souscrire aux événements, d'ajouter ou de supprimer des participants, etc.



3.2.2. Le modèle interne du contrat

Le modèle interne du contrat a deux parties : la description concrète des participants et le corps du contrat.

La description concrète des participants explique la façon d'associer, lors de l'exécution, une instance de composant avec chaque participant abstrait. Dans le cas le plus simple, on peut supposer que les participants sont fournis en paramètre avec le contrat, et que la responsabilité de localiser les participants incombe à une autre entité du monde de la coordination. Dans un cas un peu plus général, on peut dire que le contrat permet la description des participants (à partir d'un ensemble de caractéristiques non-fonctionnelles, par exemple), et que l'interpréteur du contrat est chargé de localiser les participants à partir de leur description. Ceci est très utile dans le cas des composants fonctionnels.

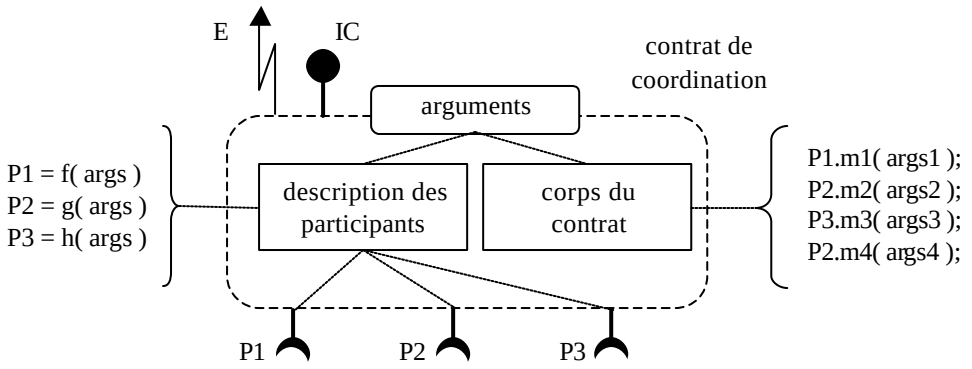
Pour illustrer l'idée expliquée, nous montrons un extrait d'un contrat d'une des applications qui seront présentées dans le chapitre 7.

```
nom =  wsProductOpen
arguments =  String machineName, String componentName

components
{
 workspaceManager: workspaceServerRole at machineName
 versionManager: versionManagerRole lookup componentName
}
```

Le contrat a deux arguments (`machineName` et `componentName`) et deux participants abstraits (`workspaceManager` et `versionManager`). Dans la description des participants, nous utilisons un langage qui explique que le participant appelé `workspaceManager` qui implémente l'interface `workspaceServerRole` doit s'exécuter sur la machine appelée `machineName`. De cette façon, à partir des arguments fournis par le monde de la coordination, le contrat est capable de localiser les participants concrets.

Si on imagine le corps du contrat comme une séquence d'appels de méthodes sur les participants, un contrat de coordination pourrait se visualiser de la façon suivante :



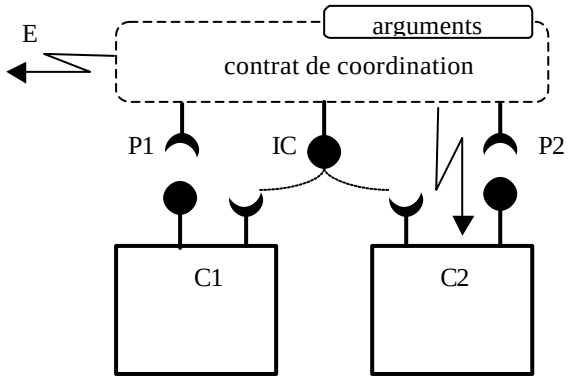
Exprimer le corps d'un contrat comme une liste d'instructions est une solution flexible et naturelle pour un programmeur, mais elle est difficile de composer, car les étapes du contrat et le protocole de coordination sont dilués dans la séquence d'instructions.

Dans le cas général, le corps du contrat peut s'exprimer de plusieurs façons possibles. Il y a des formalismes, comme les diagrammes d'état, très appropriés pour définir les étapes et la structure globale du corps du contrat. Il y a aussi des langages de coordination, comme CLF [APR00, AAP99], faits pour exprimer ce type de synchronisation entre plusieurs éléments. Dans l'implémentation que nous avons faite, et qui sera présentée dans le chapitre 7, nous avons laissé ce choix ouvert, et le moteur de la fédération accepte divers types de langages et de formalismes, à condition que son interpréteur satisfasse quelques règles et protocoles de communication. Dans la section 3.5 nous présenterons la structure que nous avons choisie dans l'implémentation actuelle pour exprimer un contrat.

3.3. Les mondes connaissent-ils les contrats ?

Est-il indispensable pour un composant de connaître les contrats auxquels il participe ? Le composant doit-il être programmé de façon particulière (en implémentant des méthodes de service, ou en lançant des exceptions particulières) pour être piloté par un contrat ? Quel type de service peut demander un composant à un contrat auquel il participe ? Le composant peut-il passer comme paramètre le contrat aux autres composants de son monde ?

Dans notre expérimentation, nous avons trouvé qu'il est extrêmement difficile d'exprimer certains aspects de la coordination, si les composants ne collaborent pas de manière étroite avec les contrats. Pour cette raison, malgré les éventuels problèmes de réutilisation des composants, nous considérons important de laisser ouverte la possibilité aux composants d'interagir avec les contrats. Le cas général d'un contrat de coordination serait donc celui montré dans la figure suivante, où les composants ont accès à l'interface de contrôle du contrat :



Cette nécessité est surtout évidente dans la gestion de l'échec du contrat, et dans le traitement de relations "profondes" entre les éléments de deux mondes distincts.

Le fait de connaître l'existence des contrats, et d'avoir accès à leur interface de contrôle, permet aussi aux composants d'écouter les événements sur le déroulement du contrat et de réagir en conséquence de manière asynchrone.

Avec cette vision, il est possible d'implémenter les divers modèles transactionnels en termes de contrats de coordination.

3.4. Composition de contrats

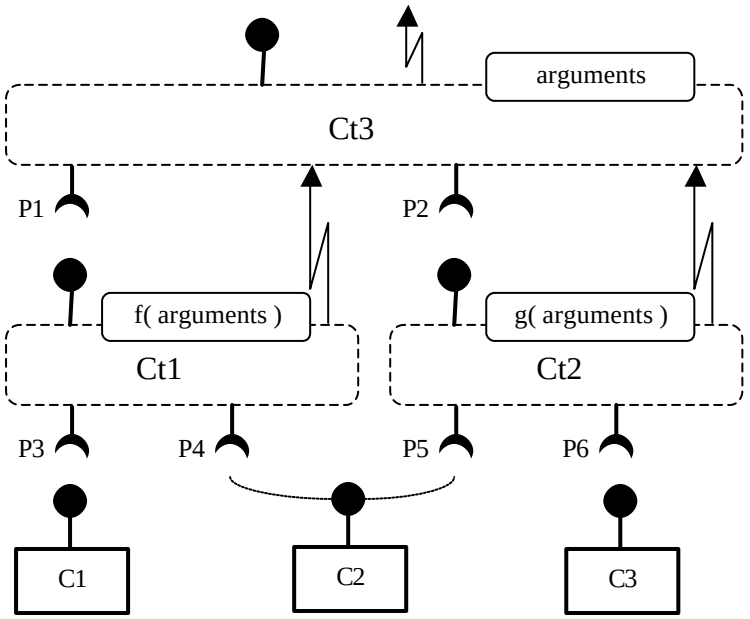
Si nous voulons composer deux mondes de la coordination, il est évident que cette composition doit passer par la composition de leurs éléments, en particulier, par les éléments de base : les contrats.

Le problème général peut se résumer de la manière suivante : étant donnés deux contrats Ct1 et Ct2 est-il possible de définir un contrat Ct3 comme étant la composition de Ct1 et Ct2 ?

Nous identifions 3 cas possibles de composition :

- Cas 1 : les contrats Ct1 et CT2 sont indépendants, car ils agissent sur des mondes différents. Ct3 correspond à l'exécution de Ct1 et Ct2 en parallèle (ou dans un ordre indéterminé).

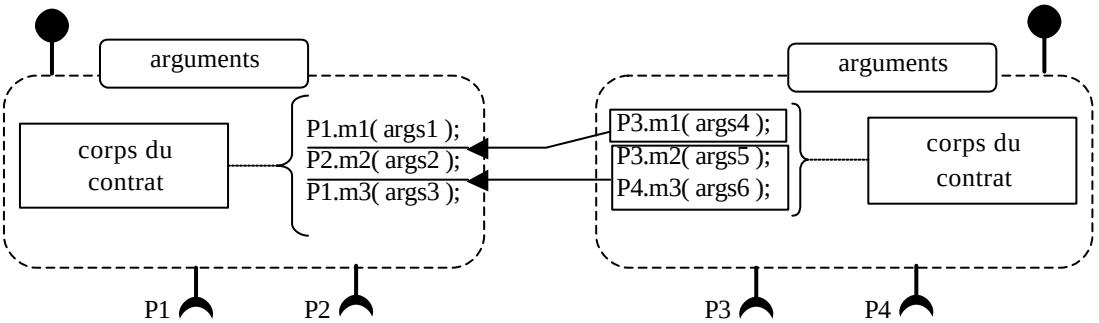
- Cas 2 : les contrats Ct1 et Ct2 coordonnent des composants appartenant aux mêmes mondes (pas nécessairement les mêmes composants), et la composition peut se faire par simple ordonnancement des contrats Ct1 et Ct2. Dans ce cas, Ct3 est un contrat de coordination entre Ct1 et Ct2, chargé de contrôler la localisation des instances partagées et d'exécuter les sous-contrats dans l'ordre convenable. On utilise pour exprimer Ct3 les mécanismes et la syntaxe des contrats de coordination.



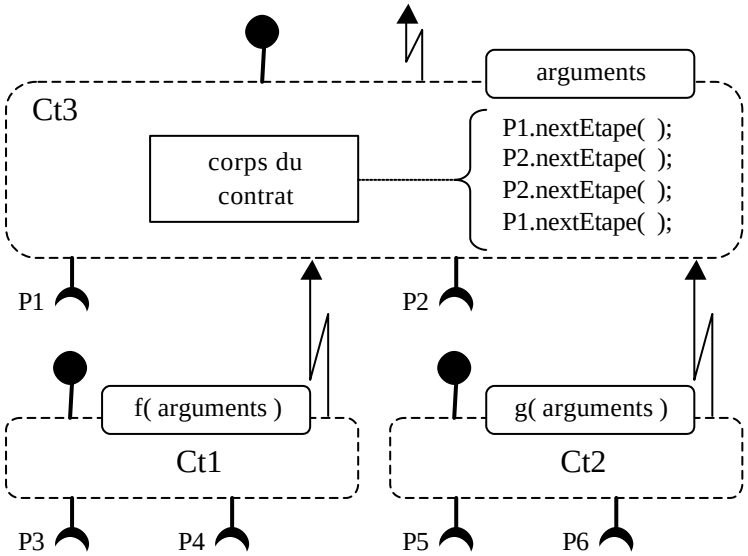
  La composition de contrats par hiérarchisation s'est montrée suffisante dans la plupart des cas dans notre expérimentation. Cette approche a l'avantage d'être assez simple à utiliser et à comprendre.

- Cas 3 : les contrats Ct1 et Ct2 coordonnent des composants appartenant aux mêmes mondes (pas nécessairement les mêmes composants), et la composition ne peut pas se faire par simple ordonnancement des contrats Ct1 et Ct2. Ceci implique que la composition doit se faire au niveau des corps des contrats. C'est ici qu'il est important

de disposer d'une structure claire du corps du contrat, car la composition des contrats doit se faire par intégration. Dans ce cas il est nécessaire d'introduire de nouveaux mécanismes et syntaxes dans les contrats. Dans la figure suivante, nous illustrons l'idée d'une intégration de contrats à niveau d'instruction :

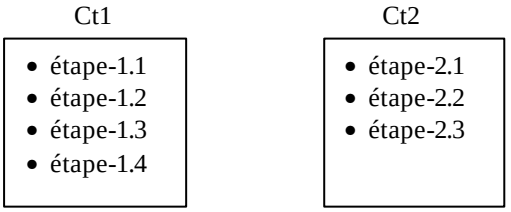


Dans notre implémentation, nous avons défini un langage textuel pour exprimer le corps d'un contrat, avec une syntaxe proche de celle de java, mais avec des informations structurelles additionnelles pour faciliter l'intégration de contrats. En ce moment, nous commençons les tests de ce type de composition de contrats. Nous avons introduit une syntaxe simple pour exprimer le corps du contrat résultant en termes des "étapes" des sous-contrats, et garder la même idée de hiérarchisation de contrats, tel que la figure suivante le suggère :



Pour obtenir cette capacité d'intégration des corps des contrats, on a ajouté à l'interface de contrôle la capacité d'exécuter un contrat étape par étape. Avec cette facilité, le contrat résultant ne fait que déterminer la manière de tisser les diverses étapes des sous-contrats, et il réutilise tout le reste des mécanismes disponibles.

Pour illustrer l'idée de composition des corps des contrats par le mécanisme montré ci-dessus, imaginez deux contrats Ct1 et Ct2 avec la structure suivante :

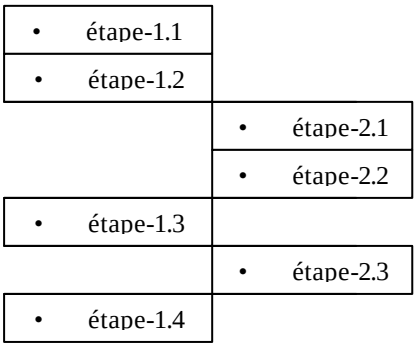


Le contrat Ct1 est composé de 4 étapes (chaque étape constituée d'une séquence d'instructions) et le contrat Ct2, de 3 étapes.

Si le contrat Ct3 est la composition de Ct1 et Ct2 décrite de la manière suivante :

- Le participant P1 de Ct3 est le contrat Ct1
- Le participant P2 de Ct3 est le contrat Ct2
- Le corps du contrat Ct3 est :
  - P1.nextEtape( );
  - P1.nextEtape( );
  - P2.nextEtape( );
  - P2.nextEtape( );
  - P1.nextEtape( );
  - P2.nextEtape( );
  - P1.nextEtape( );

On peut imaginer le contrat Ct3 tel que le suggère la figure :



Nous avons aussi considéré la possibilité d'inclure d'autres mécanismes pour faciliter la réutilisation des contrats, tels que l'héritage, mais pour l'instant nous avons décidé de rester dans l'étape d'expérimentation avec le modèle le plus simple possible.

3.5. Notre approche

Bien que nous n'ayons pas voulu le présenter de cette façon, il est évident que les contrats sont des composants du monde de la coordination, et qu'ils vont être au cœur de toute la synchronisation.

Dans cette section nous ne montrons que les caractéristiques les plus importantes des contrats que nous utilisons dans l'implémentation actuelle, mais il est sûr que ceci va évoluer au fur et a mesure que les expériences pratiques nous montrent d'autres besoins. Il

faut remarquer que l'implémentation actuelle est ouverte à tout langage de coordination pour exprimer les contrats, mais que les tests ont été faits seulement sur notre langage.

Les principales caractéristiques de notre langage sont :

- Le corps d'un contrat est structuré en étapes.
- Les étapes peuvent s'exécuter de manière séquentielle ou parallèle. Au moment de l'intégration de deux contrats, c'est le contrat de plus haut niveau dans la hiérarchie qui décide comment exécuter les étapes.
- Une étape peut être obligatoire ou optionnelle. L'échec d'une étape obligatoire engendre l'échec de tout le contrat.
- Il y a des instructions dans le langage pour faire échouer une étape.
- Toute exception lors de l'exécution d'une étape est interprétée comme un échec de l'étape.
- La composition de contrats se fait par hiérarchisation et par intégrations, tel qu'on l'a expliqué dans la section précédente.

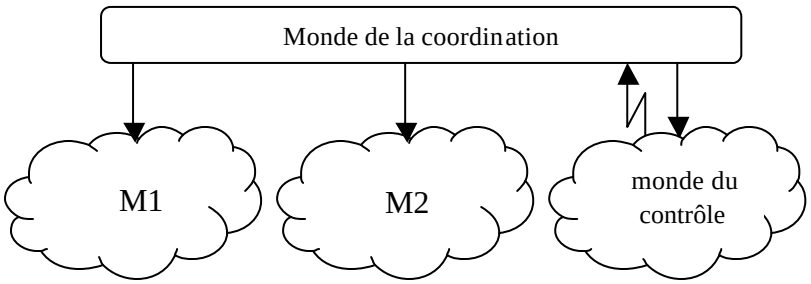
Dans le chapitre 7 nous montrerons en détail la syntaxe du langage d'expression de contrats utilisé.

Il est important de dire que méthodologiquement nous utilisons des diagrammes d'état pour visualiser les contrats, mais que, ensuite, nous les traduisons dans le langage textuel.

4. Le monde du contrôle

Dans cette section nous changeons de sujet, et nous passons à étudier le monde du contrôle, un monde chargé de guider un ensemble de mondes pour arriver à un but.

Ce que nous voulons envisager est une architecture semblable à celle exposée dans la figure suivante, où le monde du contrôle participe à la coordination avec les autres mondes, en leur imposant des objectifs globaux :

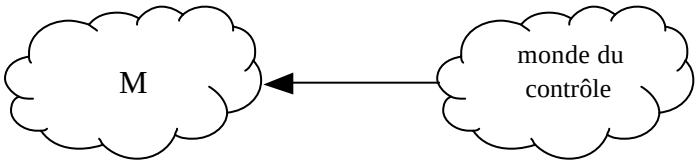


4.1. Définitions et vision globale

Dans le modèle d'objets, tel qu'il a été présenté dans le chapitre 2, il y a une séparation nette entre ce qu'on a appelé le monde du problème et la solution du problème. Le premier est chargé de représenter le monde où le problème a lieu, tandis que le deuxième exprime la façon de faire arriver ce monde à un état où le problème peut être considéré comme résolu. On a affirmé que la structure de la solution n'était pas définie par le modèle

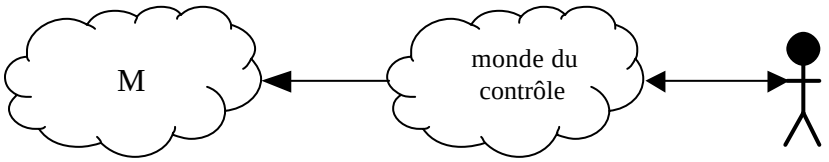
d'objets, et qu'il n'y avait pas de formalisme pour l'exprimer, différent du langage de programmation.

Dans une première approche, on peut dire que le monde du contrôle correspond au monde des solutions, où l'on peut exprimer le processus pour résoudre un problème. Autrement dit, le monde du contrôle doit être capable de guider le monde du problème pour arriver à un état où le problème est résolu.



Le monde M ne connaît pas l'existence du monde du contrôle. C'est le monde du contrôle qui impose une démarche au monde M : il est chargé de le guider.

Nous avons vu aussi dans le chapitre 2 que dans les applications interactives, il y a un monde chargé d'interagir avec l'utilisateur et de servir de médiateur pour que le client puisse contrôler le monde du problème. Dans ce cas, le monde du contrôle ne fait qu'interpréter les ordres du client et de guider le monde du problème en les suivant. C'est seulement un cas particulier de la définition antérieure du contrôle :

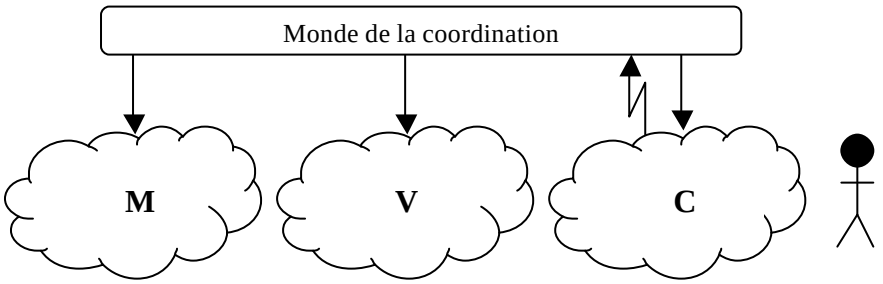


Ce que nous voulons faire dans cette section est étudier le monde du contrôle, ses éléments, sa structure, et montrer les *workflows* et les gestionnaires de procédés comme des outils de ce monde. Ceci va nous permettre d'exprimer "les solutions" avec des formalismes de haut niveau (explicites et externes), et de les faire interpréter, au lieu de laisser la solution dans le code des applications. Cette approche a l'avantage de faciliter l'évolution des solutions, et dans notre cas particulier, va nous permettre d'utiliser ce monde comme une structure de coordination.

D'autres formalismes tels que les grammaires [FHS92, GS96] ont été proposés pour exprimer le contrôle, mais ils ont l'inconvénient d'être difficiles à visualiser et d'avoir une syntaxe parfois peu intuitive.

Il est intéressant de noter que l'architecture MVC est une architecture où le monde de la coordination a été réparti entre les mondes de la visualisation et du contrôle. Tous ceux qui ont implémenté une application avec une architecture MVC savent que les mondes V et C terminent fortement pollués par M, parce qu'on leur a imposé la responsabilité de gérer les contrats de coordination et les autres structures du monde de la coordination.

Si notre hypothèse s'avère vraie, une meilleure architecture serait une architecture MVC+C (Monde - Visualisation – Contrôle + Coordination), où la coordination est exprimée indépendamment des mondes V et C, en facilitant leur évolution :



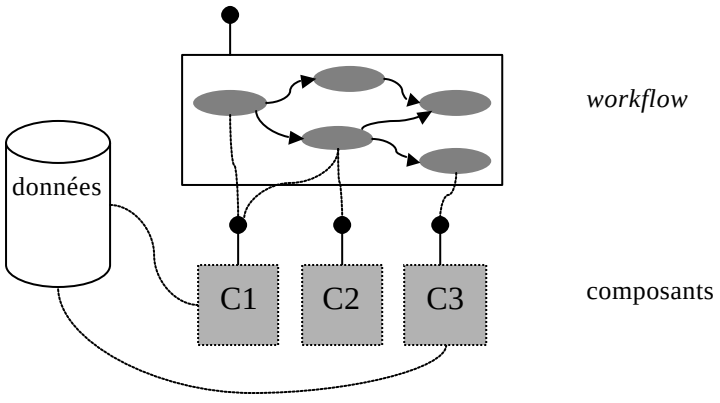
Actuellement, nous sommes en train de faire des tests pour vérifier si nos outils nous permettent de développer des applications avec l'architecture MVC+C, et surtout, de mesurer l'impact du point de vue performance, un aspect clé dans les applications graphiques interactives.

4.2. Composition de composants par le contrôle

Avant de présenter une caractérisation du monde du contrôle, nous voulons montrer comment, dans plusieurs approches, il est possible de coordonner un groupe de composants en utilisant comme structure de base le monde du contrôle.

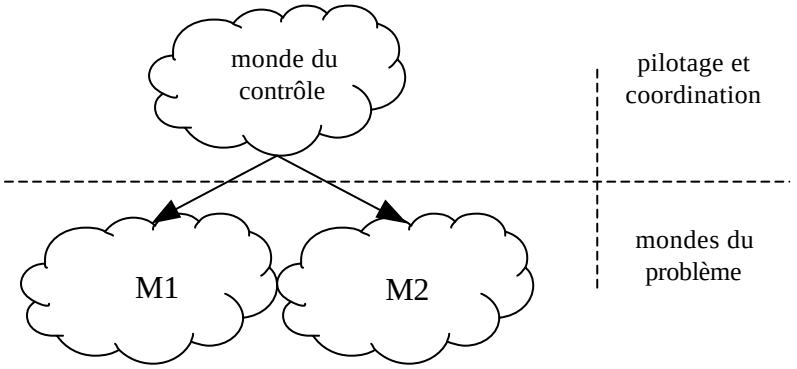
Si on considère le monde du contrôle comme la simple matérialisation d'un algorithme, il est évident que ce monde est un moyen idéal pour la composition de composants fonctionnels. On ne fait que remplacer l'expression d'une procédure dans un langage de programmation par un modèle de contrôle. C'est l'idée des BPM (*Business Process Management*) [SF02], et l'approche suivie par BPEL4WS (*Business Process Execution Language for Web Services*)
[WC02].

L'exécution d'une opération correspond au parcours d'un modèle de contrôle, entre un état initial et un état final, tel qu'il est montré dans la figure suivante :



Le monde du contrôle peut aussi être utilisé pour la coordination entre deux mondes. Dans cette architecture à deux niveaux, la coordination entre les mondes M1 et M2 se fait par le contrôle, qui guide les deux mondes de manière cohérente: à chaque fois qu'il envoie à un monde un stimulus, il envoie à l'autre les stimulus nécessaires pour que la cohérence entre les mondes soit maintenue. Le contrôle est donc responsable de faire avancer ensemble les

deux mondes vers la solution du problème, en prenant en compte toutes leurs relations et contraintes.



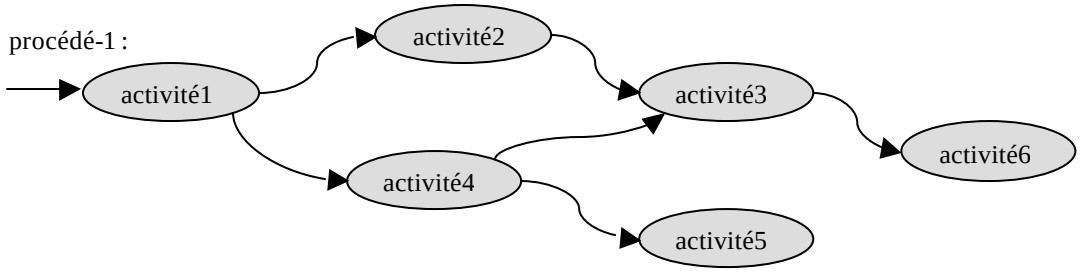
Avec cette approche, le monde du contrôle est chargé aussi de la coordination entre les mondes, ce qui le rend complexe et peu dynamique. Nous voulons pouvoir considérer le monde du contrôle comme un autre monde à coordonner et séparer ainsi les responsabilités de guider et de coordonner sur deux mondes à part.

Le monde du contrôle est le premier exemple d'un monde actif, dont le concept sera généralisé dans la section 8 de ce chapitre. Dans le cas général, on peut imaginer un ensemble de mondes actifs guidant les actions d'une application complète.

4.3. Caractérisation du monde du contrôle

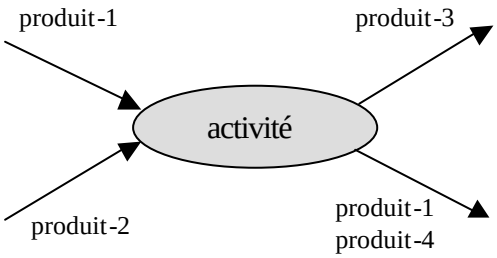
Il y a plusieurs formalismes exécutables disponibles pour exprimer un modèle du monde du contrôle. L'approche privilégiée actuellement consiste à changer de modèle de programmation : passer d'un modèle impératif (séquence d'instructions à une machine abstraite) à un modèle de flot de données (*dataflow*) –expliqué ci-dessous–, beaucoup plus expressif du point de vue graphique, et par conséquent, plus simple à maintenir et à développer.

Dans les modèles de flots de données, le concept d'activité (appelé aussi étape, tâche, pas ou instruction) est la base de la structure, et la topologie du contrôle est exprimée par les connexions entre les diverses activités. Cette vision a l'avantage de garder une étroite et naturelle correspondance avec les concepts du monde du contrôle, ce qui la rend très appropriée comme moyen d'expression. Un modèle du monde du contrôle est appelé un "procédé".

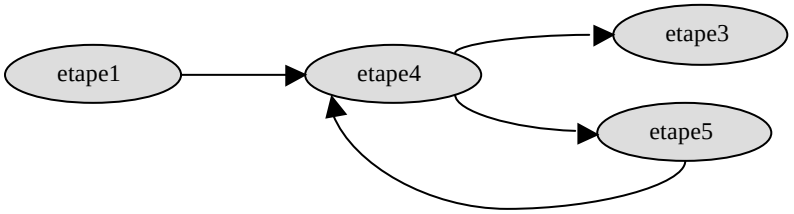


Un procédé a toujours une activité initiale et une ou plusieurs activités finales. L'exécution d'un procédé consiste à commencer l'exécution par l'activité initiale et arriver jusqu'à une activité finale. Les liens entre les activités s'appellent des connecteurs. Leur but est de définir l'ordre d'exécution des activités. Pour passer des informations entre les activités, les connecteurs sont chargés aussi de communiquer "les données", que nous allons appeler "les produits".

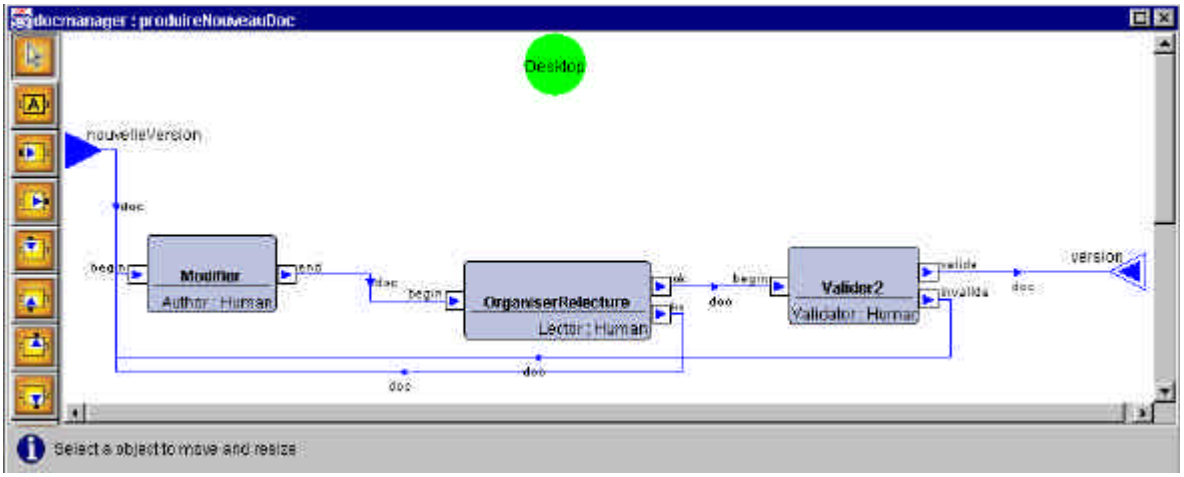
Avec cette vision, on peut dire que l'objectif d'une activité est de créer les produits nécessaires pour les activités suivantes, à partir d'un ensemble de produits donnés en entrée. Une fois les produits de sortie obtenus, l'activité se termine. Les produits sont typés et nommés.



Une activité peut se terminer de plusieurs façons. Chaque connecteur en sortie exprime une façon de terminer, avec l'ensemble de produits associés dans chaque cas. Avec les éléments montrés et avec la mécanique de flot de produits décrite, il est possible d'exprimer les structures conditionnelles et itératives des langages impératifs.



Dans le chapitre 7 nous montrerons l'outil APEL [DEA98] que nous avons utilisé pour exprimer le contrôle dans nos applications d'expérimentales. Un exemple de la syntaxe graphique d'APEL est montré dans la figure suivante:

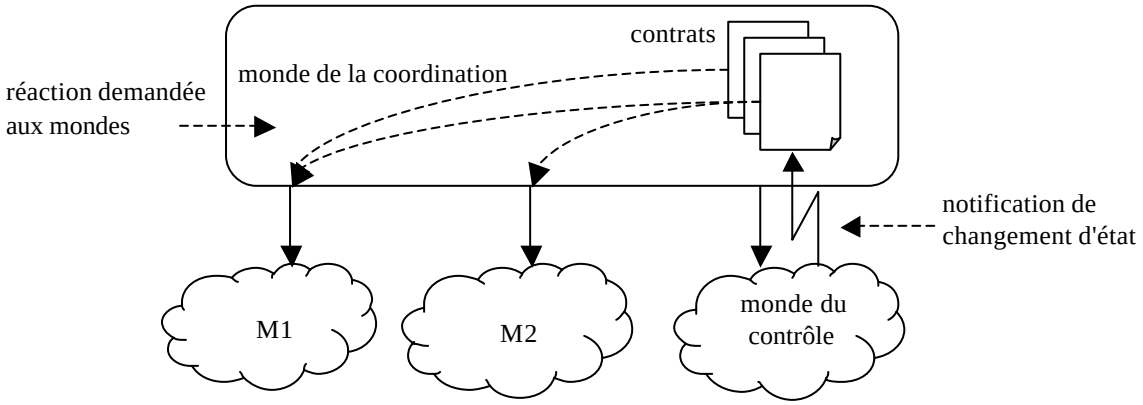


4.4. Le contrôle et le monde de la coordination

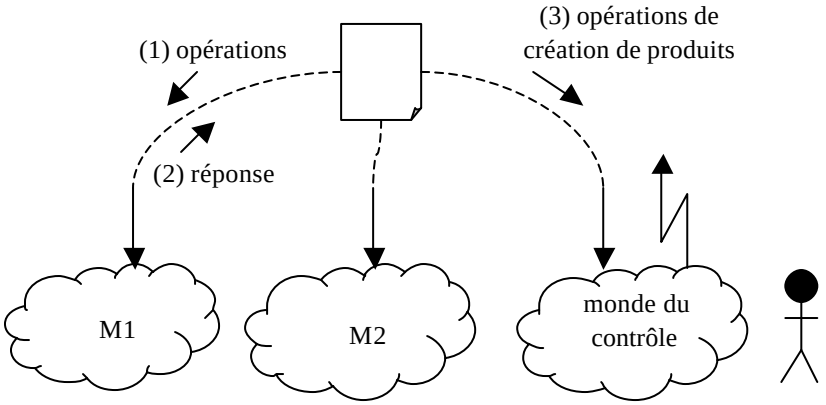
La question, maintenant, est de savoir quelle est la relation entre le monde du contrôle et le monde de la coordination ? Comment va-t-on associer des contrats de coordination avec les activités ?

Si nous voulons considérer le monde du contrôle au même niveau que les autres mondes, nous ne pouvons pas associer de manière directe des contrats de coordination avec les activités d'un procédé. Ceci équivaudrait à l'architecture de contrôle hiérarchique que nous voulons éviter. Nous devons alors passer par les mécanismes généraux de communication entre un monde actif et le monde de la coordination.

Pour donner une idée initiale (qui sera développé tout au long de ce chapitre et du chapitre suivant), nous pouvons dire que les mondes actifs (et en particulier le monde du contrôle) doivent informer le monde de la coordination de leurs changements d'état. Certains de ces changements d'état peuvent être associés à des contrats de coordination.



Les contrats, typiquement, vont demander aux mondes d'exécuter quelques opérations, de récupérer éventuellement une ou plusieurs réponses, et de communiquer cette information au monde du contrôle en termes de produits, pour qu'il puisse avancer dans le procédé qu'il interprète.



Le monde du contrôle peut aussi interagir avec un usager (dans certaines applications), de telle façon que le contrôle est réellement partagé entre les contrats et le client.

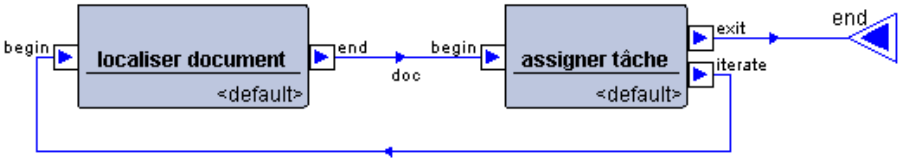
Pour illustrer le protocole décrit, nous pouvons utiliser l'exemple suivant :

- Le monde M1 est un monde où l'on trouve un ensemble de documents, chacun avec un nom et un type (une espèce de gestionnaire de documents).

- Le monde M2 est un monde avec un ensemble de personnes, avec un identifiant, un niveau et une liste de tâches a faire (une espèce de gestionnaire de ressources).

- Le but de l'application est d'assigner la lecture des documents aux personnes, en respectant une série de politiques (par exemple, garder l'équilibrage de charges, ou de limiter la lecture d'un type de document à un niveau de personnes).

- Le control de l'application peut s'exprimer avec le modèle de contrôle suivant, dans la syntaxe graphique d'APEL :

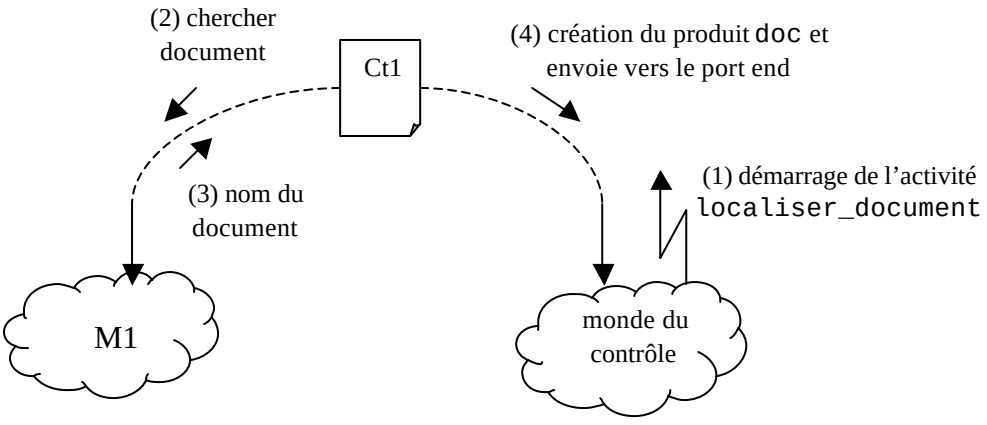


  - Une première activité pour localiser un document. Il est à noter que le document choisi (nommé `doc` dans le modèle graphique) est communiqué par le flot de données vers la deuxième activité.

  - Une deuxième activité où la tâche de lire le document est assigné à une personne. Cette activité peut redémarrer l'itération (s'il y a des personnes qui peuvent accepter un autre document en lecture) ou terminer le procédé.

- Dans le monde de la coordination, on peut imaginer au moins deux contrats :

  - Contrat Ct1 : déclenché avec le démarrage de l'activité `localiser_document`. Son objectif est de trouver un document non-lu, de créer le produit `doc` et d'inclure dans ses attributs le nom du document sélectionné.



- Contrat Ct2 : associé avec le démarrage de l'activité `assigner_tâche`. Ce contrat est responsable de trouver une personne pour lire le document. Sa structure peut se résumer avec le pseudo-algorithme suivant :

```
personne = M2.localiserPersonne( doc )
M2.assignerTache( personne, "lecture de " + doc )
M1.marquerDocumentLu( doc )
si( M2.personnesDisponibles( ) )
     activer le port de sortie "iterate"
sinon
     activer le port de sortie "exit"
```

En général, on peut dire que la mission du monde du contrôle est de définir les objectifs globaux de l'application, tandis que le monde de la coordination (en utilisant ses contrats) se charge de piloter les mondes en leur imposant un ensemble de tâches.

4.5. La cohabitation de plusieurs sources de contrôle

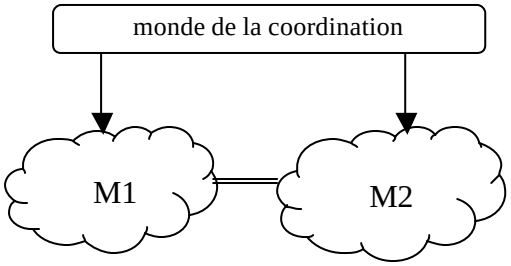
Un cas possible (et même normal) est l'existence de plusieurs sources de contrôle dans l'application globale : chaque monde actif va essayer de la guider avec sa vision "partielle" du problème. Comment va-t-on garantir une cohabitation "amiable" entre toutes ces sources de contrôle ? Autrement dit, si les contrats ne se connaissent pas entre eux, est-il possible d'éviter les interférences entre eux ? La réponse n'est pas simple. Nous allons trouver une première partie de la réponse dans la section suivante.

5. Mondes dépendants

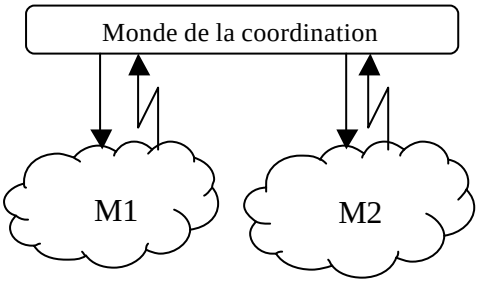
Dans cette section nous aborderons une autre facette du problème de la coordination : les mondes avec des dépendances fonctionnelles ou des dépendances conceptuelles.

5.1. Mondes fonctionnellement dépendants

D'après la définition qu'on a donnée dans le chapitre 3, deux mondes M1 et M2 sont fonctionnellement dépendants s'ils ont des connexions directes qui leur permettent de travailler ensemble. Ces connexions peuvent correspondre à un partage de données ou de contrôle.



Le problème pour le monde de la coordination est qu'à chaque demande de changement sur un monde, l'autre monde peut réagir en modifiant son état. Cette situation peut être représentée, vis-à-vis du monde de la coordination, comme deux mondes indépendants, mais actifs.



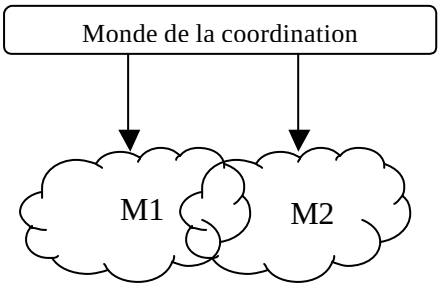
Avec cette vision, nous évitons d'introduire des éléments additionnels dans le monde de la coordination, et traitons le problème des mondes fonctionnellement dépendants comme un cas particulier d'un autre problème déjà existant : les mondes actifs.

5.2. Mondes conceptuellement dépendants

5.2.1. Le problème et les hypothèses de travail

Deux mondes M1 et M2 sont conceptuellement dépendants s'ils partagent une partie de l'espace conceptuel. Ceci veut dire que :

- M1 et M2 gèrent des concepts partagés, pas nécessairement identiques (possiblement deux facettes d'un même concept)
- M1 et M2 ne se connaissent pas et, par conséquent, toute relation doit passer par le monde de la coordination



Pour illustrer cette situation, considérons l'exemple suivant :

- Le monde M1 est le monde d'une société de transports, où les concepts gérés correspondent par exemple à des employés, des colis, et des véhicules.
- Le monde M2 est le monde de la sécurité sociale, où les concepts gérés sont par exemple des assurés, des médecins, des médicaments, etc.

Etant donné que les concepts d'employé du monde M1 et le concept d'assuré du monde M2 correspondent à deux facettes du même concept (le concept de personne), si M1 et M2 veulent modéliser la même réalité, ils doivent garder une cohérence dans leur état. Si par exemple, un employé est renvoyé de la société, le monde de la sécurité sociale doit réagir pour que l'assuré qui représente la même personne change son état d'accord à ses nouveaux droits.

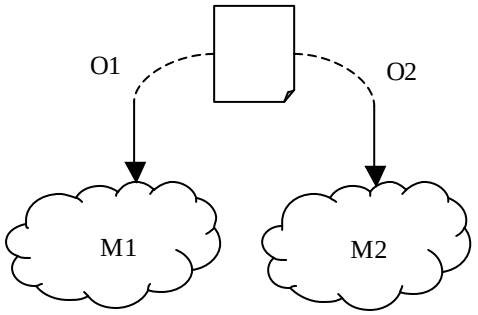
En général, on peut dire que si deux mondes M1 et M2 sont conceptuellement dépendants, les modifications dans un de ces mondes peuvent impliquer des modifications dans l'autre, malgré le fait qu'ils ne se connaissent pas et qu'ils ne peuvent pas communiquer directement entre eux. Nous supposons dans cette section que M1 et M2 sont passifs. Le cas des mondes actifs avec des dépendances conceptuelles sera traité plus loin dans ce chapitre.

Nous supposons que les concepts partagés sont contrôlables depuis l'extérieur du monde, à travers les services fournis par les interfaces des composants de M1 et M2. Si ceci n'est pas vrai, il est impossible de garantir une cohérence entre les concepts partagés. Dans l'exemple précédent, s'il n'y a pas de service dans le monde de la sécurité sociale pour mettre à jour l'état d'un assuré, il est impossible d'actualiser son état pour le rendre cohérent avec le nouvel état de l'employé de la société.

5.2.2. Les approches possibles

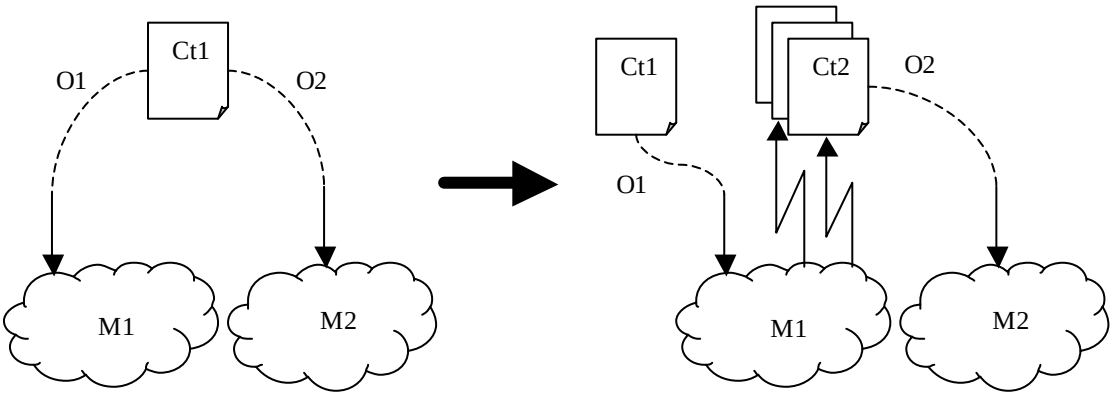
La question de fond est de savoir qui connaît la relation conceptuelle entre M1 et M2, et qui est responsable de garantir la cohérence entre ces mondes ? Il est évident que ceci ne peut pas être la responsabilité des mondes. Nous identifions au moins trois possibilités, qui seront présentées par la suite :

- Option 1 : Coordination entre M1 et M2 en utilisant les contrats de coordination. Dans chaque contrat où participe M1, M2 doit également participer, et les dépendances conceptuelles entre les deux mondes sont exprimées en termes des opérations qui garantissent une évolution cohérente des mondes. Autrement dit, à chaque opération O1 (ou groupe d'opérations) exécutée sur les éléments de M1, il faut associer une opération O2 (ou groupe d'opérations) qui fait que M2 arrive à un état cohérent avec celui de M1.



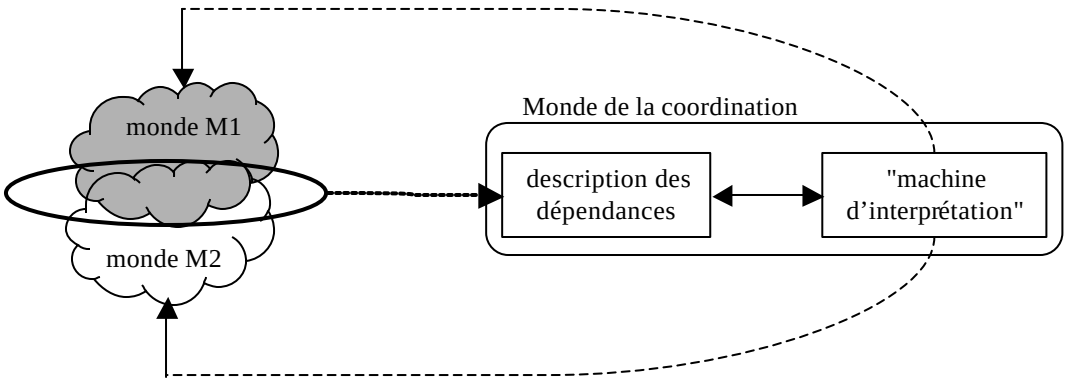
  L'avantage de cette approche est qu'on réutilise des mécanismes qui sont déjà présents dans le monde de la coordination. L'inconvénient est que les dépendances conceptuelles sont exprimées à un niveau opérationnel dans les contrats (i.e. un protocole pour maintenir cohérents les mondes) et non à un niveau structurel (i.e. quel est le concept partagé et en quoi consiste cette dépendance). Ceci a comme conséquence que l'évolution devient difficile, et les contrats, complexes à développer et à maintenir.

- Option 2 : Modifier les mondes M1 et M2 pour qu'ils se comportent comme des mondes actifs (ce sujet sera abordé plus tard dans ce chapitre), et définir des contrats de coordination liés à chaque changement d'état d'un de ces mondes qui peut concerner l'autre. La différence avec l'option précédente est qu'on spécialise les contrats : un contrat Ct1 qui a été déclenché par un monde actif avec des instructions pour résoudre le problème sur un monde M1, et un contrat (ou plusieurs contrats) Ct2 pour réagir aux changements de M1 au fur et à mesure que s'exécute le contrat Ct1.



L'avantage de cette option est la séparation des contrats par intention : les contrats dont l'objectif est de faire avancer un monde M1 dans la solution d'un problème, et les contrats qui maintiennent la cohérence entre deux mondes dépendants M1 et M2. Ceci peut guider la construction de méthodologies et de formalismes, et peut simplifier l'évolution. Toutefois, cette option a deux inconvénients principaux : d'une part les dépendances restent encore exprimées sous forme opérationnelle dans les contrats et, d'autre part, on introduit des problèmes de synchronisation entre les contrats. Ce deuxième point sera traité dans la section qui présente les monde actifs.

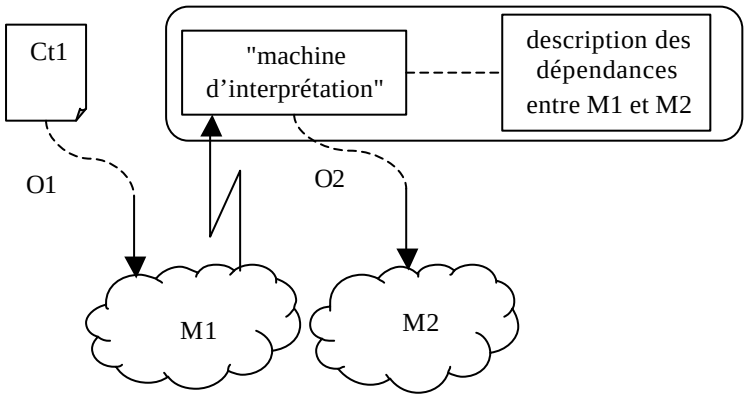
- Option 3 : Rendre explicites les dépendances conceptuelles entre les mondes M1 et M2, et passer la responsabilité de maintenir leur cohérence à un "module" du monde de la coordination, qui se base sur la description des dépendances pour intervenir.



Cette troisième option a l'avantage d'être plus claire et plus proche de la philosophie du monde de la coordination. Elle va nous permettre de réfléchir et d'exprimer les dépendances au niveau du meta-modèle des mondes concernés, et ceci a un impact profond sur les facilités de développement et de maintenance. Le risque que nous

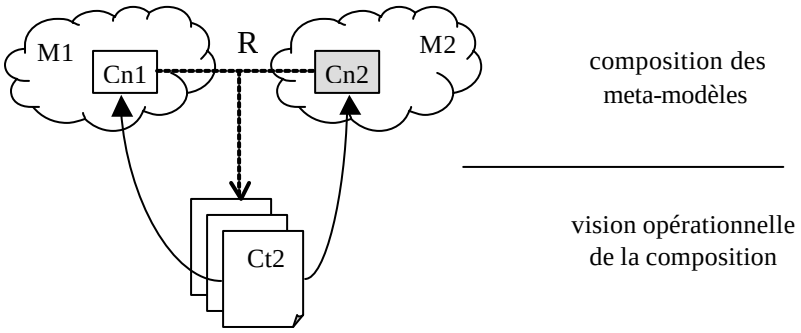
courons est de rendre trop complexe le monde de la coordination : il faut définir une manière d'exprimer les dépendances et une manière de les interpréter, deux concepts jusqu'ici étrangers au monde de la coordination

On peut imaginer le processus de coordination de la manière suivante :



(1) un contrat Ct1 est démarré. Le contrat exécute l'opération O1 sur le monde M1. Cette opération modifie un concept partagé avec M2. (2) Le monde M1 notifie son changement d'état au monde de la coordination. (3) Le monde de la coordination consulte la description des dépendances entre M1 et M2. (4) Le monde de la coordination décide d'appliquer l'opération O2 sur M2 pour assurer la cohérence.

Une idée, par exemple, pourrait consister en l'association des contrats de coordination avec chaque relation entre les concepts, pour définir ainsi la façon de maintenir la cohérence entre les concepts partagés des deux mondes.



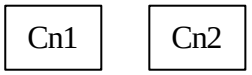
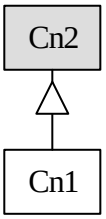
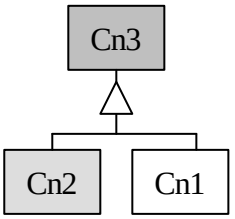
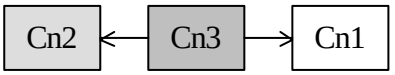
Nous avons décidé d'utiliser la troisième option, et elle sera développée dans la suite de cette section. Nous appellerons cette option "coordination par meta-composition des mondes".

Il reste trois points ouverts : (1) Comment exprimer les relations entre les concepts de deux mondes ? (2) Comment "traduire" la déclaration d'une dépendance entre deux mondes en opérations concrètes sur ces mondes (3) Est-ce qu'il y a une taxonomie des dépendances possibles entre deux concepts, pour guider les choix des points 1 et 2 ?

5.2.3. Types de relation entre concepts

Nous commençons par le troisième point de la discussion précédente. Nous ne prétendons pas développer une théorie complète autour de la meta-composition, ni une classification de toutes les relations imaginables entre deux concepts. Nous voulons simplement montrer les cas les plus importants que nous avons rencontrés lors de l'expérimentation, et qui nous ont guidés dans la construction des outils et des méthodologies.

Si M1 et M2 sont deux mondes, Cn1 est un concept de M1 et Cn2 est un concept de M2, on a, au moins, les cas suivants :

Cn1 et Cn2 sont deux concepts identiques : il y a une correspondance directe entre les concepts	
Cn1 est une spécialisation de Cn2 : les attributs de Cn2 sont un sous-ensemble des attributs de Cn1 (une espèce d'héritage entre les concepts)	
Cn1 et Cn2 sont des spécialisations d'un concept abstrait Cn3 : Cn1 et Cn2 partagent un sous-ensemble de ses attributs	
Un troisième concept Cn3 fait la relation entre Cn1 et Cn3. Ce cas sera traité dans la section des mondes émergeants, et il ne sera pas considéré dans la suite de cette section.	

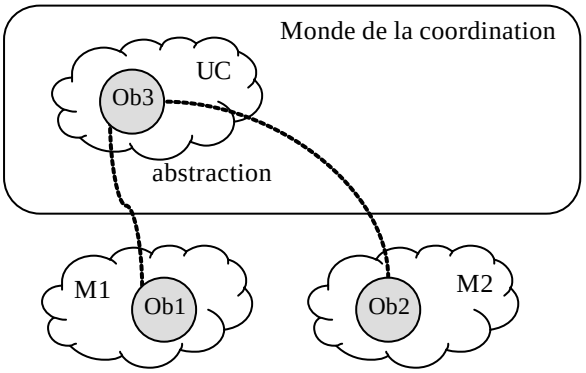
La composition entre les mondes M1 et M2 consiste alors à identifier les concepts communs, et le type de relation qu'il y a entre eux. Cette relation doit s'exprimer de manière explicite. Du point de vue méthodologique, le type de relation va guider la classe de contrat qu'il faut associer à la relation.

5.2.4. Matérialisation des concepts partagés

De toutes les solutions possibles pour modéliser et représenter la composition entre deux mondes, nous avons choisi la matérialisation des concepts partagés. Ceci veut dire que, dans le monde de la coordination, on va avoir un lieu avec l'état de l'espace conceptuel partagé entre les mondes. C'est ce que nous appellerons l'univers commun (UC).

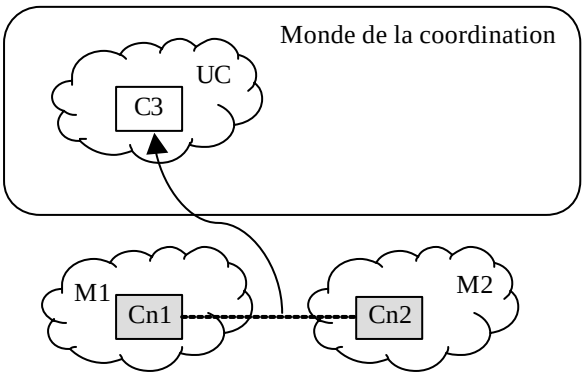
On peut imaginer l'univers commun, lors de l'exécution, comme un modèle d'objets où l'on peut trouver une instance Ob3 pour chaque couple d'instances Ob1 (de M1) et Ob2 (de

M2) avec des dépendances entre elles. Cet objet Ob3 garde assez d'information dans son état pour identifier les objets concernés par la relation.



L'idée est alors de faire passer toute la coordination entre Ob1 et Ob2 par l'objet de l'UC Ob3.

De même, lors de la phase de définition, on peut imaginer l'univers commun comme un ensemble de classes où il y a une classe C3 pour chaque couple de concepts dépendants Cn1 (de M1) et Cn2 (de M2). Cette classe C3 de l'UC doit exprimer toutes les dépendances entre les concepts Cn1 et Cn2. La question est alors de déterminer comment exprimer les dépendances entre deux concepts en termes d'attributs et de méthodes d'une classe ? Ou plus précisément, comment modéliser le concept de relation pour pouvoir le matérialiser dans l'univers commun ?



Dans l'exemple de la société de transports et de la sécurité sociale, on aurait : Cn1 = concept d'employé, Cn2 = concept d'assuré, C3 = classe `Personne`. Très probablement, dans la classe C3 on ne trouverait que le numéro de la carte d'identité de la personne, et quelques autres attributs pour définir son état partagé.

Cette approche a plusieurs avantages :

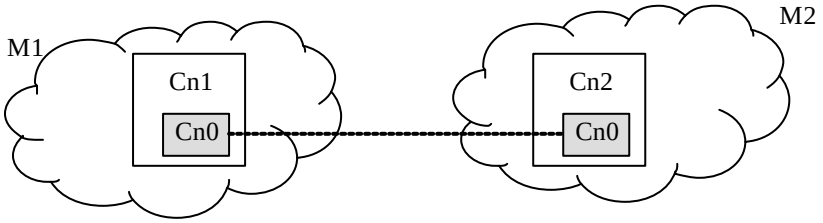
- On peut manipuler les concepts communs à un niveau abstrait, au lieu de manipuler directement le code qui garanti la cohérence entre ces concepts.
- On évite de passer par des syntaxes additionnelles pour décrire les dépendances entre concepts. On va les exprimer en termes de classes. On n'a pas besoin, non plus, d'un interpréteur des règles de cohérence, car on peut se baser sur les méthodes des classes de l'UC pour exprimer les dépendances.

- Du point de vue du développement et de la maintenance il y a beaucoup d'avantages, car cette vision est plus naturelle et facile de composer. En particulier, il est possible de généraliser cette matérialisation de concepts communs à plusieurs mondes.

Dans le chapitre 7 nous allons montrer des exemples concrets de construction d'un univers commun à partir d'un ensemble de mondes.

Pour préciser un peu plus l'idée d'univers commun, il est indispensable de définir ce qu'on veut modéliser avec une classe de cet univers. L'idée n'est pas de copier l'état complet des concepts partagés, et encore moins de passer (ou copier) une partie de la sémantique des concepts dans les classes de l'univers commun.

Pour commencer, on peut affirmer que si deux concepts Cn1 (de M1) et Cn2 (de M2) sont dépendants, il est possible d'identifier un concept Cn0 correspondant à la partie commune de Cn1 et Cn2 (sauf dans le cas des mondes émergents, qui seront traités dans une autre section). Cette partie commune doit toujours garder une cohérence, tel qu'il est suggéré dans la figure suivante :

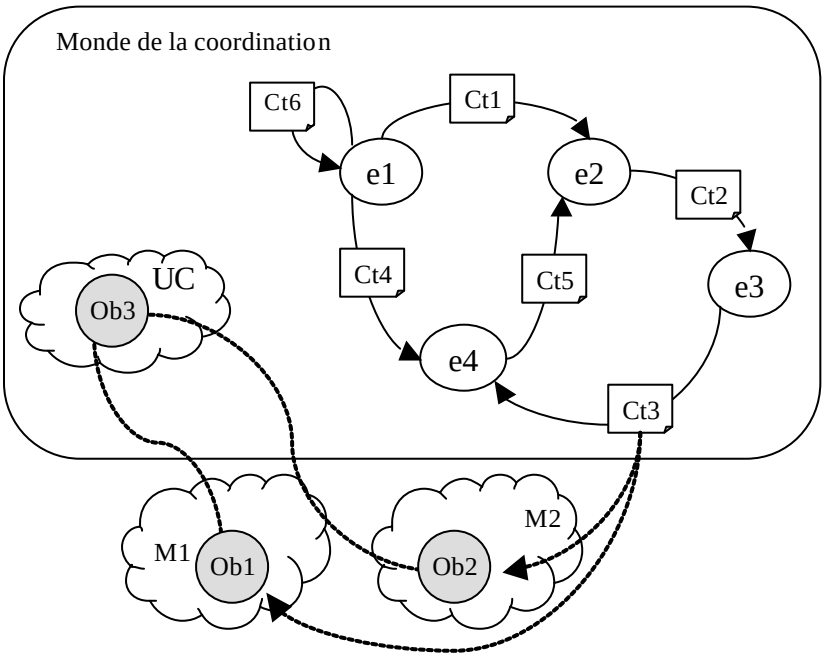


La première possibilité à explorer serait de définir une classe C0 dans l'UC pour représenter Cn0. Cette classe aurait un ensemble de méthodes pour copier le comportement de ce concept dans les deux mondes (si c'est le même concept, il doit avoir le même comportement dans les deux mondes). Cependant, même si cette idée peut paraître intéressante, elle n'est pas très pratique (ni réaliste), car on finirait possiblement par ré-implémenter une bonne partie des applications, et par dupliquer leur état : dans la plupart des cas, pour exprimer la sémantique de Cn0, il faudrait inclure dans C0 toutes les relations de Cn0 avec les autres concepts de M1 et M2. Nous considérons qu'il n'est pas viable de généraliser l'idée d'isoler un concept et sa sémantique pour les représenter dans une classe à part.

Mais, en réalité, on n'est pas intéressé par l'état de Cn0 (ni par son comportement), mais seulement par ses changements d'état. Imaginez, un instant, qu'on est capable de modéliser le "cycle de vie" du concept Cn0 et d'associer des contrats de coordination avec chaque transition de ce modèle. Dans ce cas, on laisse la sémantique et l'état dans les mondes, et on se concentre dans la modélisation de l'évolution des concepts partagés. Si on suit cette idée, une classe de l'UC comporterait :

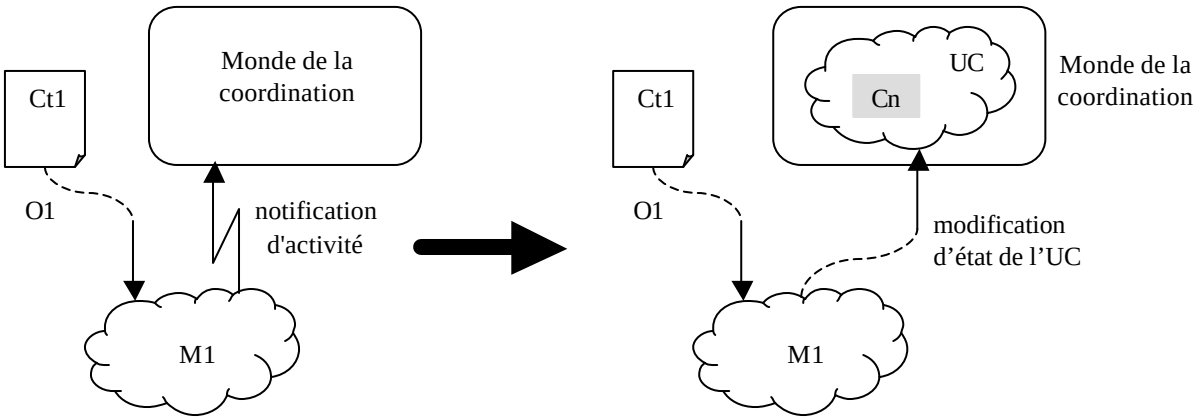
- Des attributs pour pouvoir "localiser" les instances de M1 et M2 concernés par la dépendance.
- Des attributs pour modéliser l'état dans le cycle de vie du concept partagé.
- Des méthodes pour modéliser l'évolution dans le cycle de vie du concept.
- Des contrats de coordination associés aux transitions du cycle de vie du concept.

On peut visualiser l'approche avec l'exemple montré dans la figure suivante :



- Ob3 est l'objet de l'univers commun qui modélise la relation entre Ob1 (de M1) et Ob2 (de M2). Les attributs d'Ob3 lui permettent de localiser Ob1 et Ob2 et de les passer comme paramètres aux contrats de coordination.
- e1,…, e4 sont les états du cycle de vie des objets de la classe d'Ob3.
- Les contrats de coordination Ct1,…,Ct6 sont associés aux transitions du cycle de vie de d'Ob3. Ces contrats sont déclenchés à chaque fois que l'objet de l'univers commun change d'état. Les participants du contrat peuvent être Ob1, Ob2 ou les deux.

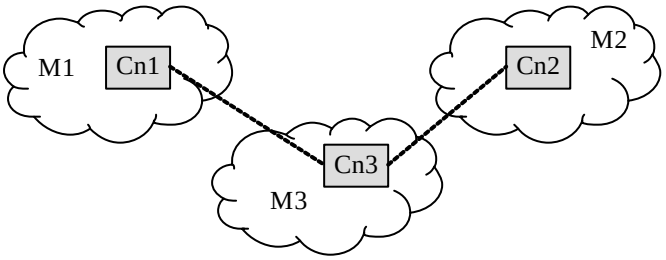
La notification d'activité d'un monde va se transformer en appels vers les objets de l'univers commun :



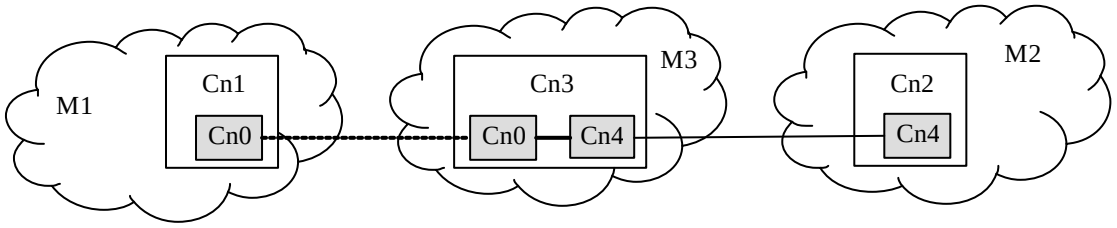
Pour l'instant nous nous arrêtons ici dans la description de l'univers commun. Nous approfondirons les idées dans la section qui traite les mondes actifs et dans le chapitre 6, car nous voulons pouvoir considérer les mondes dépendants et les mondes actifs comme deux cas particuliers d'une même problématique.

6. Mondes émergents

Nous étudions maintenant le problème suivant : soit M1 et M2 deux mondes qui ont des dépendances conceptuelles indirectement via des concepts qui appartiennent à un monde M3.

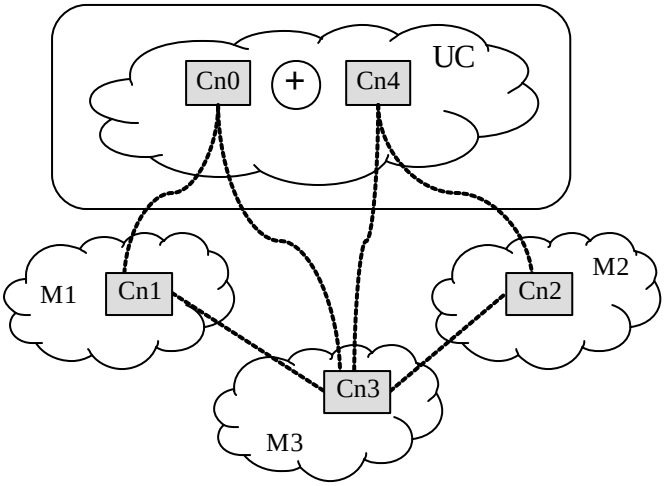


En utilisant la syntaxe graphique de la section précédente pour montrer la partie commune des concepts, nous obtenons la figure suivante, où le monde M3 connaît la relation entre Cn0 et Cn4 :



Si M3 n'est introduit que pour exprimer cette relation "indirecte" entre M1 et M2, on dit que M3 est un monde émergent.

Indépendamment du fait que M3 soit un monde émergent ou non, la solution est toujours la même : composer M1 avec M3 comme deux mondes dépendants (pour obtenir le concept Cn0 dans l'UC), composer M2 avec M3 de la même façon (pour obtenir le concept Cn4 dans l'UC), et, finalement, composer les deux résultats dans l'univers commun (composition des concepts Cn0 et Cn4 dans l'UC).



La composition de compositions sera un sujet abordé dans le chapitre suivant. Ce qu'on peut dire pour l'instant est que pour gérer les mondes émergents et les dépendances simultanées entre concepts de plusieurs mondes, il faut pouvoir composer les concepts de l'univers commun.

7. Une application vue comme un composant

7.1. Le problème

Si nous voulons considérer le problème d'intégration d'applications comme un cas particulier du problème de composition de mondes par coordination, il est nécessaire de pouvoir gérer les applications comme des composants (éventuellement actifs). Ceci implique plusieurs choses :

- Une description fonctionnelle abstraite de l'application (concept de rôle, section 7.2) pour découpler sa fonctionnalité de son implémentation.

- La capacité de modifier l'application pour qu'elle implémente un rôle, et pour qu'elle puisse se comporter comme un monde actif (concept d'enveloppe (*wrapper*), section 7.3)

- La possibilité de créer et de détruire des instances du composant (section 7.4)

- Une description non-fonctionnelle du composant, qui décrit les aspects physiques de l'application (section 7.5)

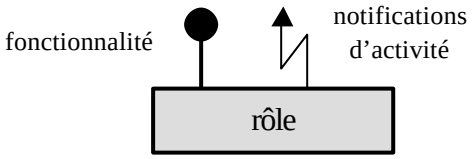
- La matérialisation d'un sous-ensemble du monde implicite dans l'application, de telle façon qu'il soit possible de traiter l'application comme n'importe quel autre monde, au moment de la composition par coordination (section 7.6).

Dans cette section nous introduirons aussi la nuance que nous faisons entre "application" et "outil", et qui sera utilisée dans le chapitre suivant au moment de parler de composition de domaines.

7.2. L'abstraction de la fonctionnalité : le concept de rôle

Dans notre modèle, nous avons introduit le concept de rôle pour modéliser les aspects fonctionnels des applications. Un rôle est un ensemble d'interfaces avec un nom. D'un point de vue conceptuel, on peut considérer le rôle comme une unité fonctionnelle de plus haut niveau qu'une interface. D'un point de vue pratique, on peut l'utiliser comme une facette d'un composant (i.e. une fonctionnalité accessible à travers un nom).

Si le rôle représente un monde actif, en plus de la fonctionnalité, il doit aussi déclarer le type "d'activité" de l'application. Pour ce faire, le rôle doit indiquer les changements d'état que l'application peut produire par sa propre initiative.



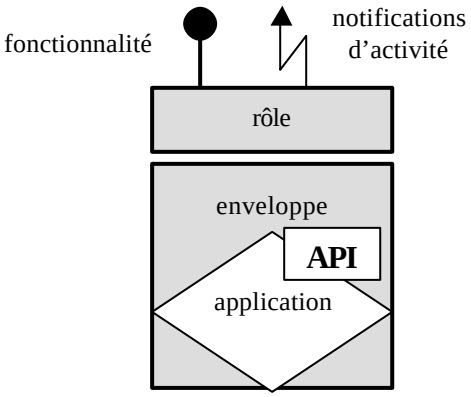
7.3. Le concept d'enveloppe (*wrapper*)

L'objectif de l'enveloppe d'une application est de :

- Implémenter le rôle. Ceci implique l'adaptation de l'application (en utilisant son API, par exemple) pour satisfaire le contrat exigé par le rôle.

- Faire toutes les adaptations technologiques entre l'application et le monde de la coordination.

- Informer le monde de la coordination de toutes les modifications d'état faites par initiative propre de l'application. Ceci équivaut à dire que, en cas de besoin, l'enveloppe est chargée d'implémenter le protocole des mondes actifs.



La construction de l'enveloppe (*wrapper*) peut se faire de trois façons, notamment : par l'implémentation d'un adaptateur, par instrumentation de l'application, ou par un mélange des options précédentes.

- Un adaptateur : si l'application dispose d'une API (*Application Programming Interface*) la construction de l'adaptateur est simple, à condition que l'API et le rôle soient sémantiquement compatibles. Si l'application n'a pas d'API, il faut voir s'il est possible d'utiliser un autre mécanisme de communication pour contrôler l'application. Dans le chapitre 7 nous montrerons plusieurs adaptateurs qui ont été développés pour des applications qui n'ont pas d'API. Il est à noter qu'il y a des applications pour lesquelles il est très difficile d'écrire un adaptateur pour implémenter un rôle, même si l'application et le rôle sont sémantiquement compatibles.

- Instrumentation de l'application : cette option est utilisable si on dispose des sources de l'application, ou de si l'application est écrite dans un langage dont les binaires sont modifiables (comme c'est le cas de Java et son *bytecode*). Dans notre cas, avec la MOE

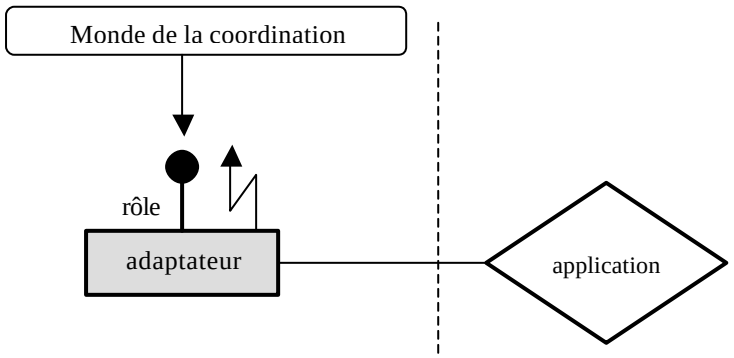
(machine à objets étendus) [DES02], nous sommes capables d'adapter aisément les applications écrites en java, à partir de son code binaire (*bytecode*).

- Un adaptateur et l'instrumentation de l'application : parfois, il est nécessaire de mélanger les techniques précédentes pour créer l'enveloppe d'une application.

7.4. La création d'instances

Par simplicité, dans cette section, nous allons supposer que l'enveloppe de l'application a été créée avec un adaptateur.

Dans ce cas, créer une instance d'un composant correspond simplement à exécuter l'application, à créer une instance de son adaptateur, et à le connecter. Ceci implique que dans le modèle non-fonctionnel du composant, en plus d'une fabrique pour l'adaptateur, il faut définir aussi la commande qui démarre l'application.



D'après notre définition, une application représente un monde avec un état et un comportement particulier.

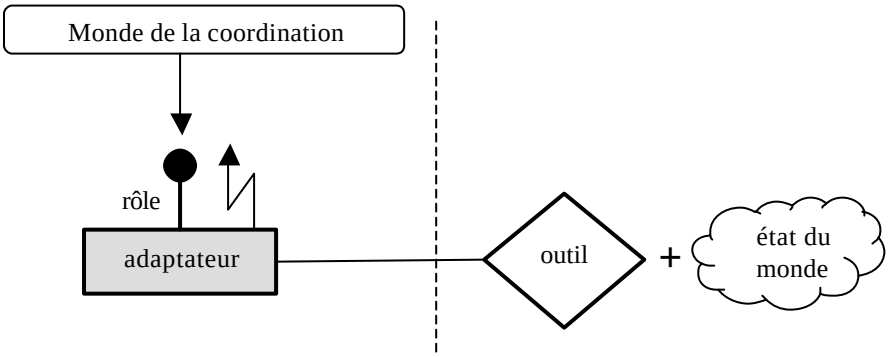
Un outil, pour sa part, peut être considéré comme un patron d'application, qui implémente le comportement d'un monde, mais qui n'a pas d'état (nous appellerons "modèle" un état du monde). On peut imaginer Microsoft Excel comme un exemple d'outil. Dans ce cas, un modèle est un fichier .xls et l'exécution d'Excel avec un fichier .xls est une application.

La définition que nous avons donnée d'outil implique que :

- Pour créer une instance d'un composant implémenté par un outil, il est indispensable de définir le modèle que l'outil doit utiliser.

- Les modèles sont externes à l'outil, explicites et nommés.

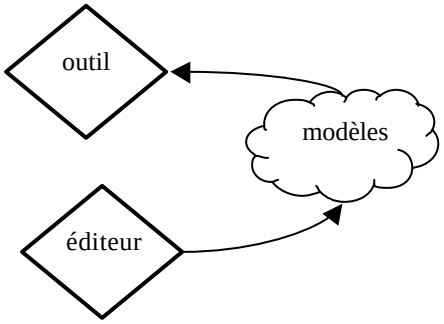
- Chaque couple (outil, modèle) est une instance différente du composant. Ceci va nous permettre d'identifier une instance de composant par le nom du modèle.



- Un outil est remplaçable par n'importe quel autre outil ayant le même rôle, à condition de partager le même modèle.

- La mobilité des instances de composant peut se baser sur la mobilité des modèles, et pas nécessairement sur la mobilité du code.

Pour un outil, il est normal de compter sur un autre outil (un éditeur) pour définir les modèles. Nous avons utilisé beaucoup d'outils dans notre expérimentation, et dans la plupart des cas, il y a une séparation entre l'outil d'édition de modèles et l'outil qui implémente le comportement du monde.



7.5. Le modèle non-fonctionnel

Dans le modèle non-fonctionnel du composant, il faut tenir compte, entre autres, de :

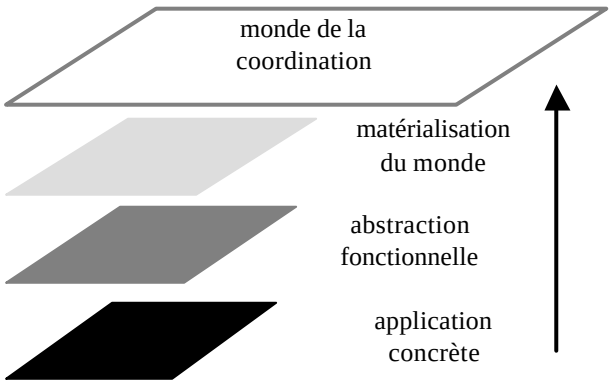
- Où est-ce que l'application est installée (sur une seule machine, sur plusieurs machines?).
- Comment démarrer l'application sur chaque machine où l'application est installée.
- S'il s'agit d'un outil ou d'une application.
- La fabrique de l'adaptateur
- S'il est possible de démarrer plusieurs fois la même application sur la même machine.

7.6. La matérialisation du monde

Avec l'abstraction fonctionnelle d'une application, obtenue à travers son rôle, on est capable de faire participer l'application dans les contrats de coordination, et d'informer l'activité de l'application au monde de la coordination. Si le monde représenté par l'application est indépendant des autres mondes, cette solution est suffisante. Par contre, s'il

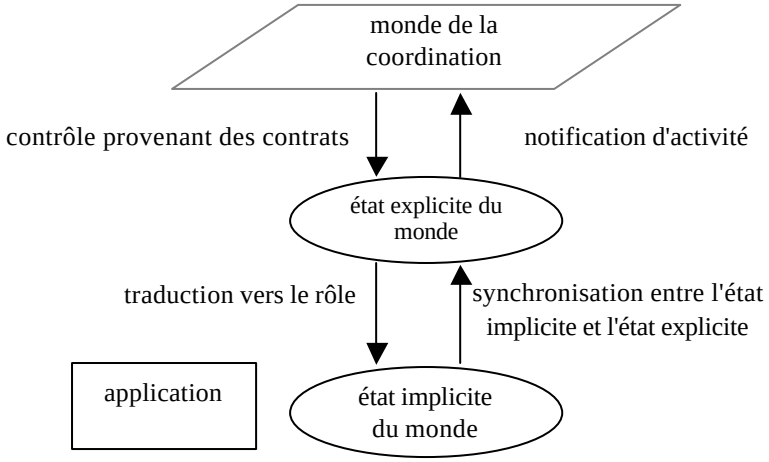
y a de dépendances conceptuelles, nous sommes incapables de les exprimer. Où sont les concepts de l'application ?

Pour cette raison, et afin de pouvoir utiliser une application comme un monde, il reste une couche entre l'abstraction fonctionnelle (obtenue par le rôle) et les mécanismes de coordination de la fédération : la matérialisation des concepts partagés avec d'autres mondes. La figure suivante illustre la couche additionnelle, qui va permettre de considérer une application comme n'importe quel autre monde.

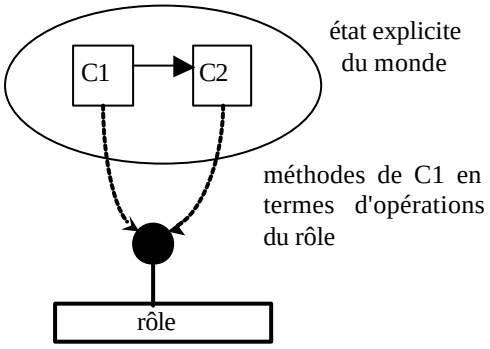


D'abord, il faut dire qu'une application appartient à un monde, dont l'état est implicite dans l'état de l'application. Nous n'avons accès à cet état qu'à travers l'API de l'application, et seulement en termes fonctionnels. Ceci veut dire que nous sommes incapables de remonter à l'univers commun une relation entre un de ces concepts et un concept d'un autre monde, car le concept n'existe que dans l'état de l'application.

L'idée est alors de rendre explicite une partie de l'état des concepts du monde, et d'utiliser cette matérialisation comme base de coordination avec les autres mondes, tel qu'il est suggéré dans la figure suivante :



Dans la matérialisation (d'un sous-ensemble) du monde nous allons créer des objets pour représenter ces concepts, et nous allons définir la sémantique de ces objets, par traduction en termes d'opérations du rôle.



Dans les applications développées dans la phase d'expérimentation, et qui seront présentées dans le chapitre 7, nous avons utilisé cette technique de matérialisation des concepts du monde, pour plusieurs outils dans le domaine de la gestion de procédés.

8. Mondes actifs

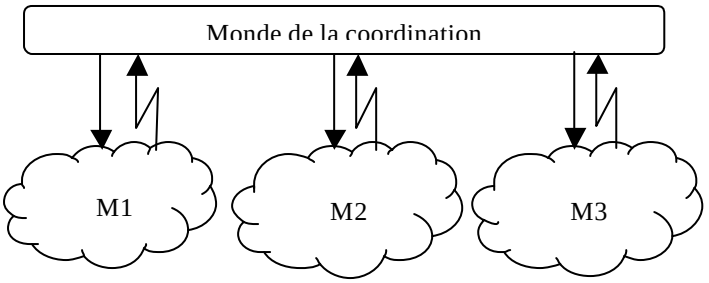
8.1. Le problème

Nous allons étudier dans cette section le problème des mondes actifs.

Initialement, nous avons défini un monde actif comme un monde ayant d'initiative. C'est-à-dire, comme un monde qui n'attend pas uniquement des ordres de la part du monde de la coordination pour changer d'état, mais, qui, pour une raison quelconque, est capable d'évoluer de son propre gré. C'est le cas du monde du contrôle (i.e. un *workflow* guidant vers une solution) ou d'une application interactive.

Dans les sections précédentes, nous avons montré aussi qu'une partie du problème des mondes dépendants (et des mondes émergents) peut s'exprimer en termes des mondes actifs.

Pour considérer tous les cas, nous allons définir un monde actif comme un monde dans lequel certains changements d'état doivent être notifiés au monde de la coordination. Ces modifications d'état vont engendrer une source de "contrôle" pour le monde de la coordination, qu'il va traduire en l'exécution de certains contrats de coordination. L'un des objectifs du monde de la coordination est alors de permettre la cohabitation cohérente de toutes les sources de contrôle.



Le monde de la coordination est alors responsable "d'observer" les mondes actifs et de réagir à leurs notifications. Les questions qu'il faut se poser alors sont : (1) Comment notifier le monde de la coordination ? En quoi consiste cette notification ? (2) Qui décide

quels sont les changements d'état d'un monde qui doivent être notifiées ? (3) Qui, dans le monde de la coordination, est concerné par la notification d'un changement d'état d'un monde ? (4) Quelle est la relation avec l'univers commun ?

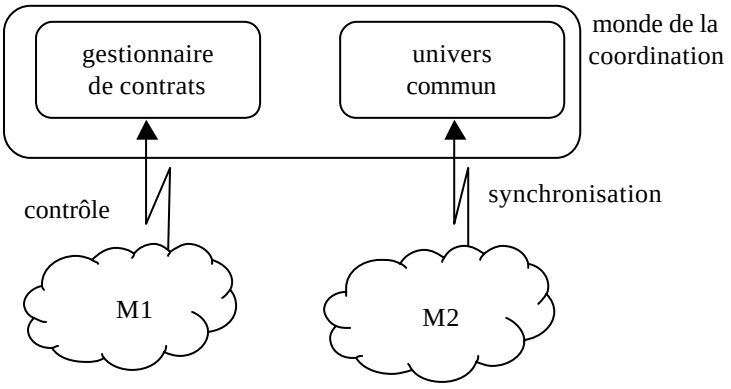
Dans la suite nous allons aborder ces quatre points.

Il faut remarquer que nous utilisons les termes "notification" et "observation", qui ont la plupart du temps une connotation "d'asynchrone", dans un sens beaucoup plus large, comme nous le verrons dans la suite de cette section.

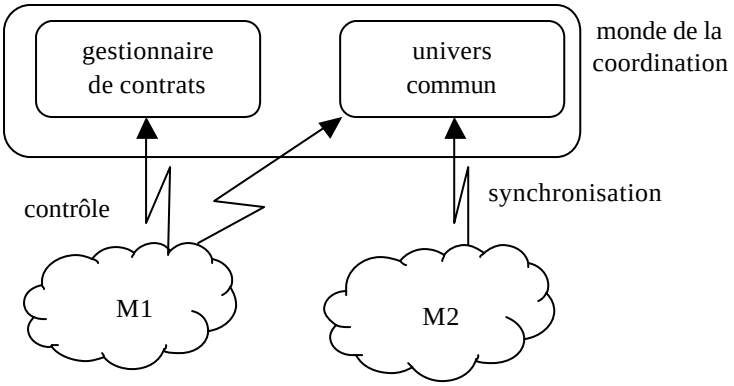
8.2. Relation avec le monde de la coordination

Il y a deux types "d'activité" des mondes actifs qui fournissent des notifications au monde de la coordination : l'activité qui provient du contrôle (qui guide les mondes dans la recherche de la solution) et l'activité qui provient de la synchronisation entre mondes (qui garantit la cohérence entre les divers mondes). Dans le chapitre suivant nous verrons qu'il y a aussi une troisième source d'activité provenant de la composition des mondes.

D'après ce qu'on a esquissé dans les sections précédentes, l'activité de synchronisation est "écoutée" par l'univers commun (qui gère les contrats de synchronisation) et l'activité de contrôle (générée par le monde du contrôle) est traitée par le gestionnaire de contrats (qui gère les contrats de contrôle).



Un cas possible, et souvent utilisé dans notre expérimentation, est de faire "remonter" l'activité de contrôle directement dans l'univers commun, pour éviter ainsi le protocole de contrôle en plusieurs étapes : un monde contrôlé directement, et tous les mondes dépendants contrôlés indirectement comme réponse à la synchronisation.



De cette façon, le monde du contrôle peut interagir directement avec l'univers commun, et gérer le contrôle et la synchronisation dans un même contrat. Les avantages de cette possibilité sont la simplicité et la performance. Le principal inconvénient est le mélange des deux types de contrats. Une solution de compromis est d'utiliser la mécanique de composition de contrats pour les maintenir séparés, mais de les assembler comme des unités dans l'univers commun.

Il y a plusieurs structures possibles pour le gestionnaire de contrats : la palette de possibilités va d'un simple tableau d'association entre "activités" et contrats, à un système de règles pour choisir le contrat adéquat en accord avec un état du monde de la coordination. Dans le prochain chapitre nous en reparlerons.

8.3. Les notifications d'activité d'un monde

Trois aspects doivent être considérés en ce qui concerne les notifications d'activité d'un monde :

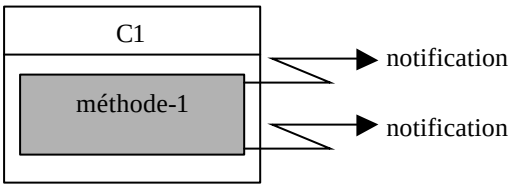
- L'identification ou la localisation de l'objet de l'univers commun concerné. La notification doit contenir assez d'informations pour permettre au monde de la coordination de trouver le destinataire. Il est nécessaire alors d'associer un identifiant avec chaque objet de l'UC, ou d'introduire le concept de racine dans l'UC, qui permet de trouver un point d'entrée dans le modèle d'objets, et de naviguer jusqu'à l'objet concerné. Dans notre modèle, nous avons introduit ces deux options.

- La description de l'activité. Pour "expliquer" l'activité au monde de la coordination il est nécessaire (1) d'identifier le type de changement d'état du monde et (2) de passer assez d'information pour pouvoir définir (paramétrer) les contrats associés. Pour faire cela, il y a deux options principales :

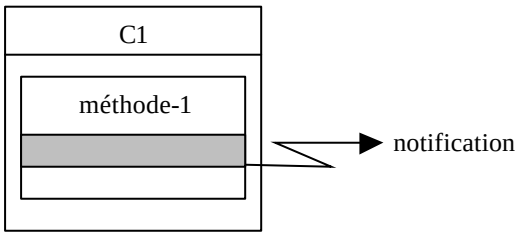
  - Transmettre seulement l'information de la cause du changement (le nom de la méthode appelé dans le monde et les arguments de l'appel) et laisser le monde de la coordination "interpréter" le changement d'état que cela va produire, et réagir en conséquence.

  - Définir une liste de changements possibles d'état du monde, et associer à chacun l'information nécessaire pour le décrire.

  Du point de vue d'extensibilité, la première option est peut-être la meilleure solution, car les modifications d'un monde pour le rendre actif peuvent être automatisées (par instrumentation du code binaire, par exemple). Les principaux problèmes sont : (1) la complexité d'interpréter (sans tout le contexte) le changement d'état produit par l'appel, (2) l'unité de synchronisation est la méthode (un grain trop gros dans certains cas), (3) Les dépendances de types générées par l'interprétation dans le monde de la coordination des arguments de l'appel.



Dans la deuxième option, il est possible de travailler à un niveau de granularité plus fin, puisque l'unité n'est pas la méthode mais l'instruction. L'inconvénient est que rendre un monde actif implique la participation d'un programmeur, et l'accès au code source du programme.



Nous verrons dans le chapitre 6 que de cette décision dépend la sélection des méthodes des classes de l'univers commun. En tout cas, du point de vue méthodologique, il est convenable de rendre explicites les changements d'état du monde indépendamment des appels de méthodes.

Dans notre modèle, nous avons laissé les deux possibilités ouvertes.

- La synchronisation entre le monde actif et la réaction du monde de la coordination. Dans chaque cas, il est important de définir si le monde doit attendre la fin de l'exécution des contrats de coordination pour continuer (exécution synchrone), ou s'il se contente d'informer le changement mais qu'il continue son travail (exécution asynchrone).

9. Conclusions et discussion

Dans ce chapitre nous avons étudié le problème de la coordination de mondes, en utilisant comme moyen une compréhension de ce que pourrait être la modélisation du monde de la coordination. Il y a déjà une structure de base, quelques idées à développer mais, aussi, beaucoup de questions ouvertes. Nous considérons qu'avec les bases fournies ici, nous allons pouvoir dans le chapitre suivant concrétiser les idées et montrer les fédérations comme une architecture qui va permettre de matérialiser les concepts du monde de la coordination.

Il est évident qu'il y a encore une énorme quantité de décisions à prendre, pour passer des idées à une modélisation concrète (et ensuite à une implémentation), et que de ces décisions va probablement dépendre la viabilité de l'approche. C'est pour cette raison que nous avons choisi cette manière de présenter le travail : (1) le problème, (2) les idées, (3) un modèle concret, (4) une implémentation et un ensemble d'outils de supports, et (5) des applications réelles sur l'implémentation. Ceci nous permet de garder la trace entre le problème et les solutions, et de revenir aux étapes plus abstraites en cas de difficulté.

Chapitre VI
Fédération de composants

1. Introduction

Dans le chapitre précédent on a étudié le problème de composition de composants par coordination dans ces diverses facettes. On a expliqué que l'approche choisie consistait à modéliser et étudier les mécanismes d'un monde imaginaire de la coordination. On a avancé dans les idées générales, mais il y a encore beaucoup de points ouverts, qu'on a considéré plus liés à une solution particulière qu'au problème en général.

Dans ce chapitre on tentera de mettre tout ensemble et de choisir, entre toutes les options disponibles, celles qui semblent les plus appropriées du point de vue pratique. L'objectif est d'arriver à une concrétisation simple du monde de la coordination, afin de pouvoir réaliser les expérimentations nécessaires et de raffiner ainsi les idées de base. Nous montrerons qu'une fédération est une architecture logicielle qui va permettre la construction d'applications à partir d'un ensemble de composants appartenant à des mondes indépendants.

Ce chapitre est structuré comme suit : dans la section 2 nous présentons la fédération, dans la section 3 nous discutons sur le problème de composition de mondes, et dans la section 4 nous mentionnons quelques conclusions.

2. Une fédération comme une architecture logicielle

2.1. Le concept de fédération

D'après la définition qu'on trouve dans le dictionnaire Hachette, une fédération est "un regroupement, sous une autorité commune, de plusieurs sociétés".

Nous avons choisi le terme fédération pour nommer ainsi l'architecture logicielle qui matérialise le monde de la coordination. A un niveau abstrait, on peut dire qu'une fédération a des caractéristiques très proches des caractéristiques du monde de la coordination, et qui sont énoncées dans la suite :

- Une fédération n'a pas été construite pour résoudre un problème particulier, mais pour permettre aux membres d'agir ensemble, afin de satisfaire les intérêts individuels, tout en respectant l'autorité commune.

- L'autorité commune est exprimée en termes de règles de comportement et de protocoles entre les membres. Cet ensemble de règles est appelé souvent la constitution.

- Chaque élément fédéré garde une autorité locale et un certain degré d'indépendance, tant que cela ne contredit pas la loi commune.

- Tous les membres d'une fédération peuvent avoir une influence sur les autres membres au travers de la fédération. Les membres peuvent travailler ensemble de cette manière.

- Une fédération connaît ses membres et chaque membre est conscient qu'il fait partie de la fédération. Il est possible que les membres aient des relations directes entre eux. C'est une autre manière de travailler ensemble.

- Dans une fédération il est possible qu'un des membres impose ses objectifs aux autres (des membres passifs). C'est ce qu'on pourrait appeler une fédération dictatoriale.

2.2. Une fédération de composants

Nous imaginons une fédération de composants comme une architecture basée sur les principes énoncés précédemment.

Notre objectif est de définir une fédération comme étant une architecture logicielle ouverte et dynamique, adaptable aisément à divers types de problèmes et de mondes, et qui s'appuie, pour arriver à son objectif, sur la collaboration d'un ensemble de composants participants.

Dans notre vision, une fédération de composants est un triplet :

F = < CP, CT, UC >

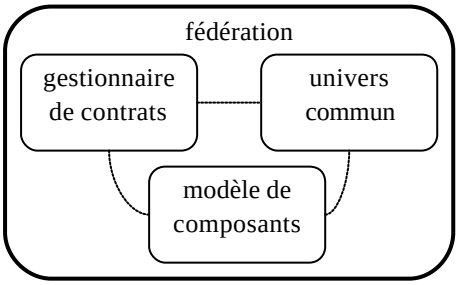
où :

- CP est un ensemble d'instances de composant : les membres de la fédération. Tous ces membres sont décrits et gérés par un modèle de composants MC. En utilisant la terminologie du monde de la coordination, on dirait que chaque monde est décrit comme un ensemble de composants, tous exprimés en termes du même modèle de composants MC.

- CT est un ensemble de contrats entre les membres. Ces contrats représentent les règles de fonctionnement de la fédération, et les protocoles à respecter. Il faut remarquer que les contrats doivent être exprimés en termes compréhensibles par tous les membres.

- UC est l'univers commun : une vision abstraite des relations et des éléments communs entre tous les membres de la fédération.

Graphiquement, on peut imaginer une implémentation d'une fédération de la manière suivante :



Dans les sections suivantes nous allons présenter les trois éléments de la structure de support d'une fédération : le modèle de composants, le gestionnaire de contrats et l'univers commun. Nous n'allons pas entrer dans les détails d'implémentation. Nous allons préciser uniquement les décisions que nous avons prises pour concrétiser les concepts du monde de la coordination.

2.3. Le modèle de composants

Le modèle de composants conçu pour exprimer les participants de la fédération sera décrit en détail dans le chapitre 7. Pour l'instant, nous n'allons énoncer que les principales caractéristiques de ce modèle :

- Le modèle est utilisable pour faire de la composition structurelle (composants de composition) et de la composition fonctionnelle (composants fonctionnels). Le modèle comporte les mécanismes nécessaires pour la création et la localisation d'instances, par exemple.

- Le modèle est implémenté sur une machine virtuelle de composants (CVM) qui est réflexive et distribuée. Ceci nous permet de gérer la distribution au niveau de composition au lieu de la gérer au niveau de la coordination. Ceci veut dire qu'on ne va pas retrouver le monde de la distribution comme l'un des mondes à coordonner, mais que le concept de distribution sera géré comme une caractéristique du modèle de composants de base.

- Le modèle supporte la définition d'applications et d'outils comme des composants. Il permet de définir, dans le modèle non-fonctionnel du composant, les caractéristiques physiques des applications.

2.4. Le gestionnaire de contrats

Le gestionnaire de contrats est chargé de gérer les contrats de contrôle et les contrats de synchronisation. Il est composé de deux parties : une partie pour définir les contrats (y compris sa composition), et une partie pour les interpréter lors de l'exécution.

Par rapport aux contrats, nous avons pris les décisions suivantes :

- Nous allons associer un nom unique à chaque contrat. Ceci veut dire que les mondes actifs (avec des liens directs vers les contrats de contrôle) et l'univers commun vont faire référence directe aux contrats par leur nom. Ceci implique qu'il ne va pas avoir un processus de sélection du contrat, basé sur un système permettant de considérer l'état

de la fédération dans le choix du contrat (un système de règles ou un système expert, par exemple). Probablement, dans une prochaine version, nous allons étudier la viabilité d'incorporer ce type de mécanismes à la fédération.

- Au lieu de définir un langage unique pour exprimer les contrats, nous avons décidé de laisser les possibilités d'évolution ouvertes, et de définir plutôt les caractéristiques de son interpréteur pour qu'il puisse être connecté au reste de la fédération. Tout langage avec un interpréteur qui suit les règles de connexions définies peut être utilisé.

- La manière d'exprimer la composition entre contrats est un problème du langage, et elle n'est pas spécifiée par la fédération.

- Nous allons utiliser un langage textuel, proche de Java, pour exprimer les contrats. Avec l'aide d'un compilateur, nous allons générer le code exécutable du contrat, pour éviter ainsi toute interprétation lors de la phase d'exécution. Le langage permet la composition de contrats, au niveau expliqué dans le chapitre antérieur.

2.5. L'univers commun

Sur l'univers commun, il y a beaucoup de décisions qui ont été prises pour concrétiser les idées du chapitre antérieur. Nous allons grouper ces décisions en deux sections pour faciliter la lecture.

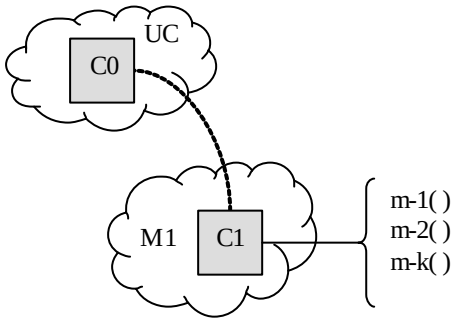
Il est important de dire que nous avons laissé la plupart d'options ouvertes, en espérant clarifier avec l'expérimentation ce qui doit être imposé (par l'architecture ou par la plate-forme) et ce qui est du ressort de la méthodologie.

2.5.1. Les attributs

Les attributs des classes de l'UC doivent permettre la localisation des instances des composants concernées par la synchronisation. Nous laissons ici l'option de stocker dans l'UC des références directes aux instances de composant, ou de stocker l'ensemble des informations nécessaires pour les retrouver.

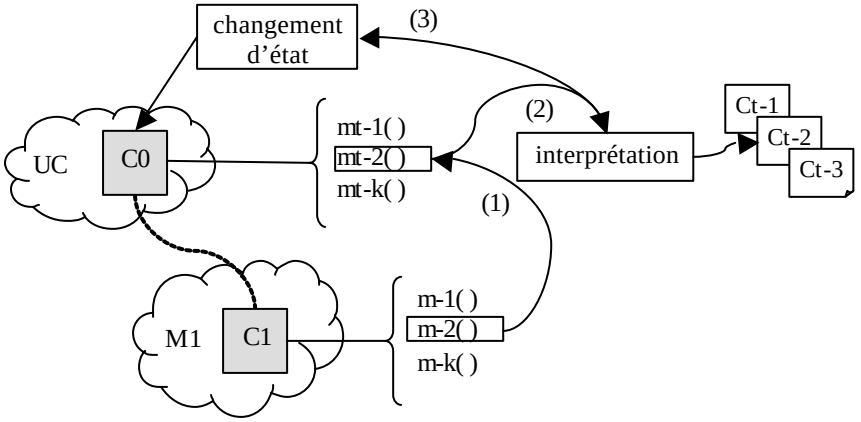
2.5.2. Les méthodes et les contrats de synchronisation

On a mentionné dans le chapitre antérieur deux options, à propos des méthodes des objets de l'UC. Pour les résumer, nous allons supposer la notation suivante : Soit C1 une classe active d'un monde M1, et soit C0 le concept abstrait de l'univers commun associé à C1. Soit $m_1,\ldots,m_k$ les méthodes actives de C1 (les méthodes qui produisent une modification d'état dont l'univers commun doit être informée).



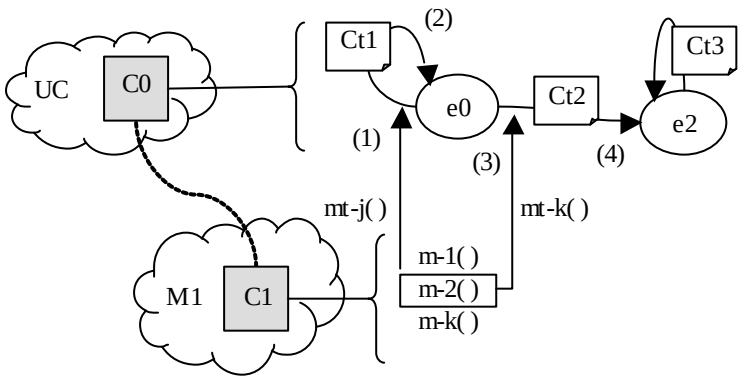
- Option 1 : La classe C0 a une méthode mt-k pour chaque méthode active m-k de C1. L'interprétation du changement d'état produit par m-k se fait dans mt-k, et selon le changement d'état produit, mt-k va lancer le contrat associé et va obliger l'objet de l'univers commun à changer d'état pour refléter l'évolution.

  Cette option a plusieurs avantages : (1) il est facile d'adapter un monde actif pour interagir avec la fédération, (2) l'association des contrats peut se faire par instrumentation des méthodes de l'univers commun, et (3) les contrats peuvent se lancer avant et/ou après le changement d'état dans le monde M1.



- Option 2 : La classe C0 a une méthode mt-k pour chaque changement d'état dans le monde M1. L'interprétation du changement d'état se fait dans M1, et par conséquent il n'y a pas de relation un à un entre les méthodes de la classe C0 et les méthodes actives de C1. Chaque méthode mt-k correspond à un changement d'état dans l'univers commun (i.e. une transition).

  Cette option a l'avantage de permettre la synchronisation à un niveau plus fin, mais implique la modification manuelle du monde actif. Dans l'exemple suivant, la méthode m-2 génère, pendant son exécution, deux changements d'état : d'abord une transition de l'état e0 vers lui-même (en déclenchant le contrat Ct1), et ensuite une transition de e0 vers e2 (en exécutant le contrat Ct2).



Nous avons utilisé dans l'expérimentation les deux options, selon les besoins. En particulier, dans le monde du contrôle, où une synchronisation plus fine est indispensable, nous avons choisi la deuxième option.

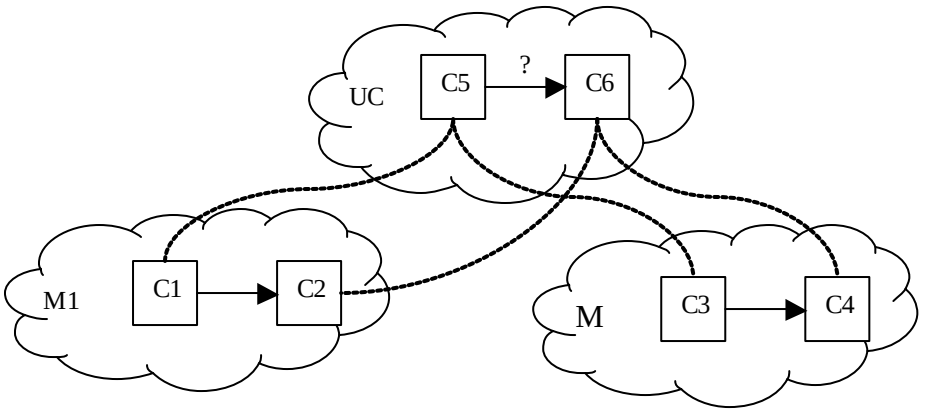
3. La composition dans une fédération

D'après notre définition, une fédération est une architecture pour faire travailler ensemble un groupe de mondes. Nous avons considéré jusqu'ici deux cas possibles : (1) les mondes sont indépendants ou (2) les mondes ont des dépendances conceptuelles. Pour traiter ce dernier cas nous avons définit un mécanisme de synchronisation entre les mondes.

Dans cette section nous voulons étudier le problème général de composition dans la fédération. Dans le chapitre 5 nous avons montré la composition de contrats comme un moyen de composition, mais deux points fondamentaux n'ont pas été étudiés : la composition dans l'univers commun et la composition entre mondes coordonnés.

3.1. Les associations dans l'univers commun

Soit deux mondes M1 et M2 avec des dépendances conceptuelles. Si ces dépendances ne se limitent pas à un seul concept mais à un groupe de concepts dans chaque monde, faut-il représenter les relations entre les concepts (au niveau des mondes) comme des associations dans l'univers commun ?



En principe, la réponse est non, car du point de vue de flot de contrôle cette relation n'est pas nécessaire. Les modifications de C6 ne proviennent pas de C5, mais de C2 :

- Normalement on aurait : C1 modifie C2 et C5, C2 modifie C6, C6 modifie C4. Ceci veut dire que toute relation entre C5 et C6 est un reflet de la relation entre les concepts dépendants des mondes. Il n'y a pas de sémantique additionnelle pour cette relation dans l'univers commun.

- Le flot de contrôle "C1 modifie C5, C5 modifie C6, et C6 modifie C2 et C4" ne correspond pas à notre idée de synchronisation, et ceci pourrait engendrer une profonde interférence sémantique dans les mondes.

Malgré le fait que cette relation semble inutile dans l'univers commun, elle peut être utilisée dans certains cas, surtout si :

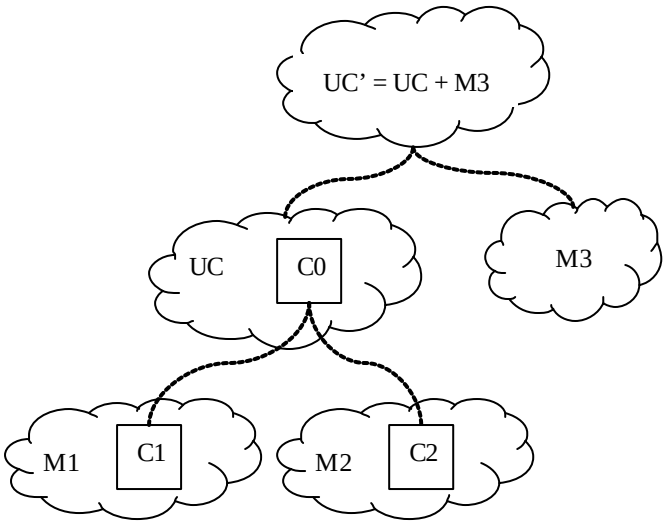
- C'est la responsabilité de l'univers commun de faire l'interprétation des changements d'état d'un monde actif. La relation stockée dans l'univers commun peut donner des informations additionnelles sur l'état du monde (on peut utiliser l'état de C6 pour interpréter l'impact dans l'état de C1).

- Il est important de permettre la navigation à l'intérieur de l'univers commun (localisation des objets de l'univers commun à partir de sa "description").

En résumé, la composition entre objets de l'univers commun ne sert qu'à maintenir une copie d'une partie de la structure des mondes, et elle n'apporte rien au niveau sémantique.

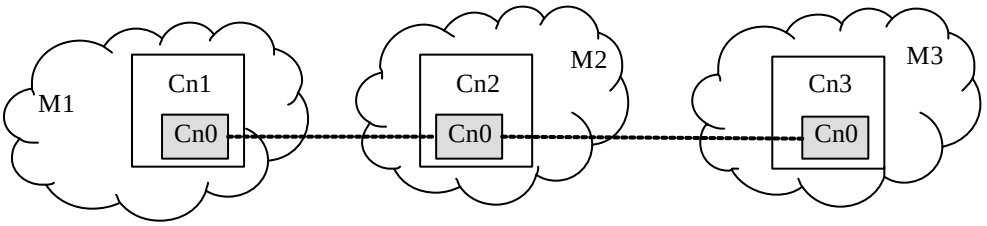
3.2. L'univers commun comme unité de composition

Soit M1 et M2 deux mondes avec des dépendances conceptuelles, avec un univers commun déjà calculé représentant les concepts communs. S'il y a un monde M3 avec des dépendances conceptuelles avec M1 et/ou M2, est-il possible de calculer le nouvel univers commun comme une "composition" entre M3 et l'ancien univers commun?



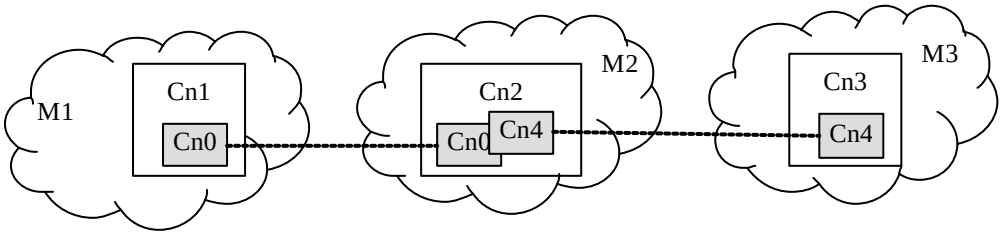
Autrement dit, l'univers commun est-il une unité de composition, qui peut être composée avec autres mondes ?

La réponse dans le cas général est non. Ceci n'est vrai que dans le cas très particulier dans lequel le concept partagé est le même entre les trois mondes, et la composition avec M3 implique seulement un raffinement des contrats d'intégration.



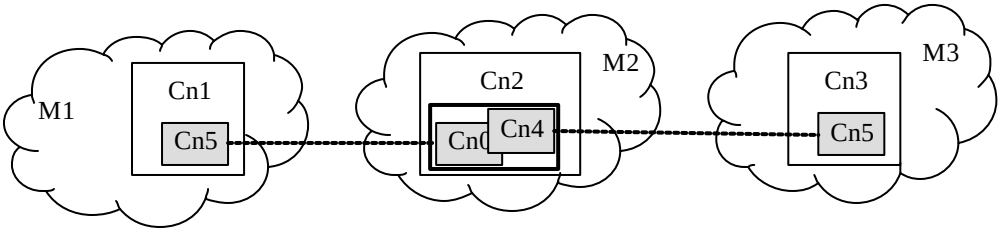
Dans le cas général, il est nécessaire de faire la composition entre M1 et M3, et la composition entre M2 et M3.

Le seul cas un peu différent qu'il faut prendre en compte, c'est le cas dans lequel la composition par couples n'est pas suffisante, et qu'il faut considérer à la fois la composition des trois mondes. Cette situation est illustrée dans la figure suivante :



Dans l'exemple, certaines modifications de Cn0 causées par Cn1 devraient générer un changement d'état de Cn3, si la modification impacte l'intersection entre Cn0 et Cn4.

L'option la plus simple serait alors de trouver un concept Cn5 qui intègre Cn0 et Cn4 à la fois, et de le représenter dans l'univers commun. Les contrats de synchronisation de l'intersection entre Cn0 et Cn4 vont faire référence aux trois mondes à la fois.

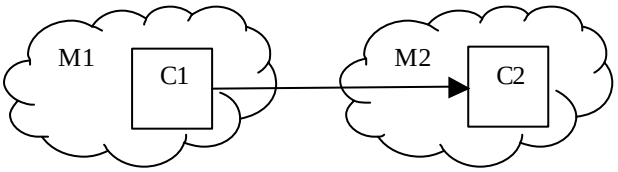


En conclusion, on peut dire que, pour la synchronisation de mondes, l'univers commun ne peut pas être utilisé comme une unité de composition. Cependant, il semble possible de définir des méthodologies de composition pour guider ce processus.

3.3. Composition de mondes

Dans les deux cas précédents il n'était pas nécessaire de garder les associations dans l'univers commun, parce que les relations étaient exprimées dans les mondes, et cela suffisait pour maintenir la cohérence des concepts. Supposez que nous voulons construire

une application dans laquelle existent des relations additionnelles à celles déjà présentes dans les mondes. Une espèce de composition de mondes, en définissant des relations additionnelles entre les concepts, pour exprimer plus que "l'union" simple des mondes. Ce type de mécanisme serait d'un grand intérêt car cela pourrait nous permettre d'enrichir les mondes en établissant des relations entre concepts de deux mondes indépendants.



Imaginez par exemple un monde M1 qui représente une société de transports (pour utiliser le même exemple du chapitre 5), et un monde M2 qui représente une banque. Dans M1 on trouve des concepts comme employé, véhicule, route, etc. Dans M2 on a des concepts comme compte, somme, prêt, etc. Même s'il n'y a aucune dépendance conceptuelle entre les deux mondes, pour une application qui veut intégrer les deux mondes, il serait important (même indispensable) de pouvoir dire qu'un employé peut avoir un compte bancaire.

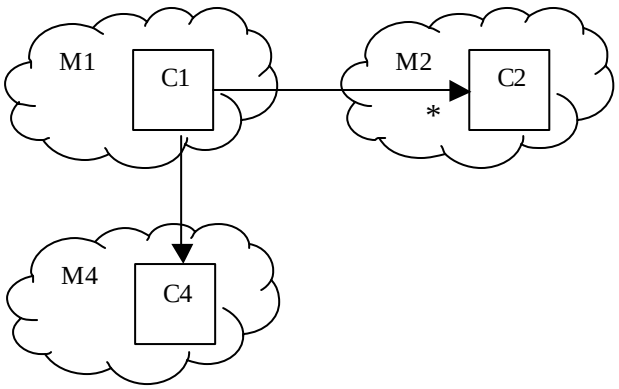
Il est à noter que le problème de composition de mondes est différent du problème de synchronisation. Si nous considérons le monde M3 comme étant le monde de la sécurité sociale (avec des concepts comme assuré, médecin, hôpital, etc.), nous avons :

- La relation entre employé et assuré est dans les deux directions, et la relation est toujours l'identité (synchronisation).

- La relation entre employé et compte bancaire a une direction, et la relation a une sémantique qui n'est précisée dans aucun des mondes. Le compte bancaire ignore l'existence de l'employé. Chaque méthode de la classe `Employé` doit être "étendue" pour intégrer dans sa sémantique le nouveau concept (composition).

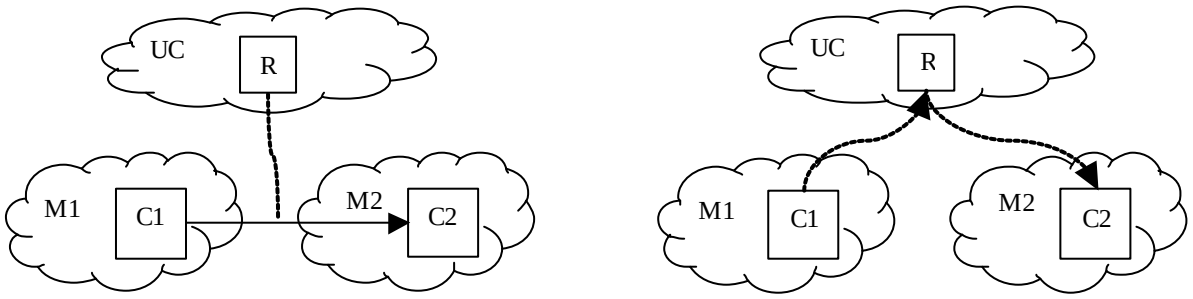
- La relation entre le concept de société et le concept de prêt peut avoir aussi une cardinalité (composition).

Dans le cas général, on peut imaginer des relations comme celles suggérées dans la figure :



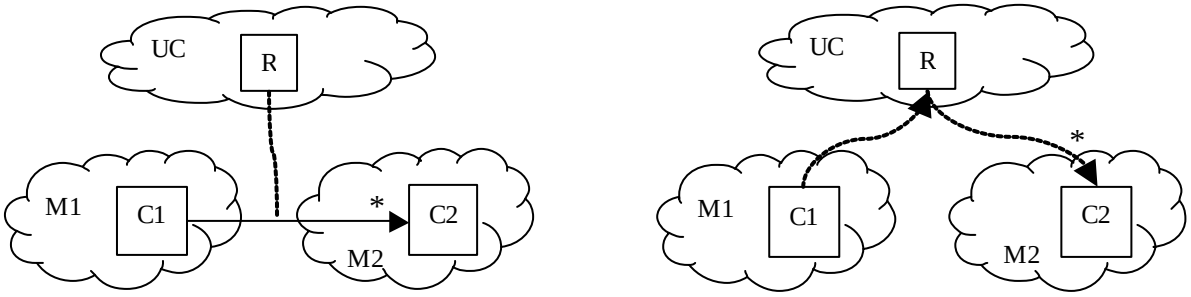
En partant de l'hypothèse qu'un monde ne peut pas connaître l'existence des autres mondes, il est naturel de penser que la responsabilité de gérer la composition incombe au monde de la coordination.

L'idée serait alors de matérialiser, dans l'univers commun, le concept de relation :

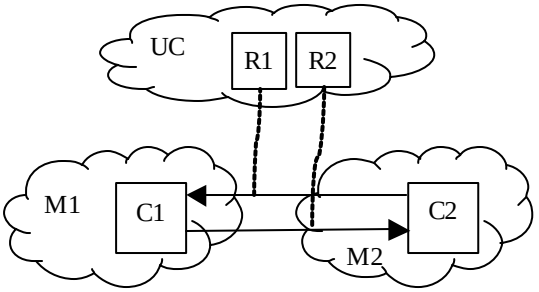


De cette façon on réutilise tous les mécanismes déjà disponibles dans le monde de la coordination. En réalité c'est une généralisation des concepts dépendants, où, au lieu de se restreindre à la relation d'identité, on considère tout type la relation entre les concepts. Autrement dit, la synchronisation n'est qu'un cas particulier de composition, avec des contraintes.

Dans les relations de cardinalité multiple, la multiplicité doit être gérée dans l'univers commun, tel qu'illustré dans la figure suivante :



Et si la relation est dans les deux sens, il faudrait matérialiser chaque relation séparément :



Avec cette vision, on peut dire que le monde de la coordination "écoute" trois types d'activité provenant des mondes : une activité de contrôle (la solution du problème), une activité de synchronisation (la gestion de la cohérence des concepts communs) et une activité de composition (la gestion des relations "ajoutées" aux mondes).

Dans notre expérimentation, en particulier dans l'application de gestion documentaire, nous avons utilisé aussi bien la synchronisation que la composition pour exprimer les relations entre les divers mondes. Dans le chapitre suivant nous montrons des exemples de chaque cas.

Il faut faire quelques précisions sur les objets de l'univers commun dans le cas de la composition :

- Les attributs des objets de l'univers commun (qui représentent une relation) doivent permettre la localisation de l'objet destination de cette relation (l'objet de la classe C2 du monde M2 dans l'exemple).

- Si la relation a un état, cet état peut se représenter en termes d'attributs des objets de l'univers commun. En particulier, les mondes émergents simples peuvent être gérés à l'intérieur de l'univers commun.

- Les méthodes des objets de l'univers commun doivent utiliser la première approche proposée dans la section 2.5.2 de ce chapitre. C'est-à-dire, les méthodes doivent garder une relation un à un avec les méthodes de la classe active. Ceci permet d'associer directement les extensions sémantiques des méthodes.

  Les méthodes des objets de l'univers commun doivent dire ce que les méthodes de la classe active diraient si elles connaissaient la relation avec le nouveau concept.

Une différence avec l'univers commun généré par la composition de mondes est qu'il est additif, à la différence de l'univers commun généré par la synchronisation. Ceci veut dire que l'univers commun de deux mondes M1 et M2 ne va pas être impacté par l'apparition d'un monde M3, et que la composition de M3 se fait par simple addition à l'univers commun existant.

4. Conclusions et discussion

Dans ce chapitre nous avons présenté une concrétisation possible du monde de la coordination : la fédération de composants. Dans le chapitre suivant nous présenterons l'implémentation concrète que nous avons réalisé, et les applications que nous avons développé pour évaluer la viabilité de l'approche.

Il est important de noter qu'il y a plusieurs façons de concrétiser le monde de la coordination, et plusieurs manières d'implémenter une concrétisation.

Pour l'implémentation, par exemple, on peut imaginer un serveur qui est capable d'interpréter une description du monde de la coordination, ou bien un compilateur capable de générer le code de l'application définie par cette description.

Nous avons choisi un mélange de ces deux options, comme nous le montrerons par la suite.

Chapitre VII
Expérimentation et résultats pratiques

1. Introduction

Dans ce chapitre, nous résumons les diverses implémentations réalisées pour tester et valider l'approche proposée dans cette thèse. Dans la dernière section du chapitre, nous ferons un bilan de toutes les expériences pratiques et discuterons les principales difficultés rencontrées.

Avec ce chapitre nous commençons la troisième et dernière partie du document. Dans la première partie, nous avons établi le problème à résoudre et les différentes approches existantes. Dans la deuxième partie, nous avons présenté la fédération de composants, une architecture logicielle pour la composition par coordination d'un ensemble de composants appartenant à différents mondes. Dans cette troisième partie nous montrerons les résultats pratiques du travail, et les leçons apprises.

Les objectifs que nous nous sommes fixés pour cette dernière partie du travail peuvent se résumer comme suit :

- Vérifier la viabilité technologique des fédérations pour une utilisation industrielle. Ceci a à voir, par exemple, avec la possibilité d'intercepter les appels des méthodes, la compatibilité entre plusieurs technologies et produits commerciaux, les systèmes de sécurité (e.g. *firewalls*), etc.

- Vérifier s'il est réellement possible de construire de vraies applications industrielles (performantes, stables, extensibles, etc.) en utilisant comme architecture de base une fédération.

- En sachant que le problème abordé est assez compliqué, et que la solution proposée est complexe, le troisième objectif est donc de vérifier si à travers un ensemble d'outils, de formalismes et de méthodologies on est capable de rendre gérable pour une personne ce niveau de complexité.

Le travail présenté dans ce chapitre est le résultat de l'effort de plusieurs personnes, pendant plusieurs années, dans le cadre de plusieurs projets de recherche et de thèses (terminées et en cours) [Ami99, CCD00a, CCD00b, DEA98, ECB98, EFS02, EVC01a, EVC01a, EVC01b, EVC01b, Ver01]. Dans la section 8 de ce chapitre, je précise ma participation directe dans l'implémentation et l'expérimentation, bien qu'il soit parfois difficile de faire la distinction à cause de notre travail en équipe, et à cause aussi des multiples versions développées de chaque outil et application.

Pour donner une idée de la taille de l'expérimentation, nous avons calculé les données suivantes sur l'implémentation courante :

- 1.100 classes java
- 180.000 lignes de code

Quelques expériences et tests pratiques ne seront pas répertoriés dans ce chapitre, soit parce qu'on considère qu'ils n'ont pas eu un impact direct sur le développement des fédérations, soit parce qu'on les a utilisés uniquement pour des tests de viabilité et ils n'ont pas été intégrés dans les outils ou les applications finales. C'est le cas du modèle OMI dont on parlait dans le chapitre 3, et le cas des tests de connectivite à la fédération avec différentes technologies et produits : CORBA (ORBacus [ORBacus] et openCCM [MM01]), EJB (JOnAS [JOnAS]), WEB-services (Apache-SOAP [SOAP]), C++ par JNI [JNI] et ObjectStore [OBS].

Tous les développements ont été faits en `java` (version 1.4.1). On a utilisé `castor` [CAS] (version 0.9.4.3) pour le stockage en XML, `ant` [ANT] (version 1.5.2) pour la création de *makefiles*, et `javaCC` [JCC] pour la génération des compilateurs des différents langages introduits.

Le chapitre est organisé comme suit : la section 2 montre le modèle de composants de la fédération, et son implémentation. La section 3 présente l'implémentation de l'univers commun utilisée actuellement. La section 4 présente un résumé des outils de composition développés. La section 5 montre APEL, un outil de gestion de procèdes utilisé comme l'outil du monde du control de la fédération. Les sections 6 et 7 présentent plusieurs applications développées (ou en cours de développement) en utilisant la plate-forme des fédérations. La section 8 fait le point sur ma participation directe dans cette partie pratique du travail. Et finalement, la section 9 conclut avec un bilan de toute l'expérimentation, et avec quelques recommandations pour le travail futur.

2. Le modèle de composants

Dans cette section, nous présentons le modèle de composants de la fédération, en utilisant comme guide le cadre conceptuel introduit dans le chapitre 3. Il faut signaler que le *framework* développé supporte aussi bien les composants de composition que les composants fonctionnels. Ceci veut dire qu'on trouve des mécanismes pour la composition structurelle (fabriques, création d'instances de composants, connexion entre instances de composants) et pour la composition fonctionnelle (mécanismes de découverte de services, mécanismes d'adaptation dynamique, etc.)

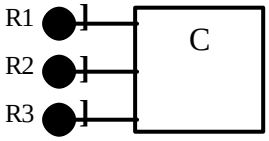
N'oublions pas que les composants de ce modèle sont les briques de base de la fédération, et que l'objectif est de les faire travailler ensemble pour atteindre des objectifs communs. Nous montrons d'abord ce modèle, pour bien séparer ce qui appartient au domaine de la composition de ce qui appartient au domaine de la coordination.

2.1. Le modèle logique

2.1.1. Vision externe

Le modèle logique du modèle de composants de la fédération est très proche du modèle de composants de java. La principale différence est que les composants implémentent un ensemble de facettes au lieu de simples interfaces.

Nous utilisons le terme rôle pour faire référence à une facette d'un composant.



2.1.2. Connexions

Il n'y a pas un modèle explicite de connexion entre les composants. Les composants de composition peuvent se composent en utilisant une fabrique. Pour les composants fonctionnels les connexions peuvent se faire dynamiques lors de l'exécution.

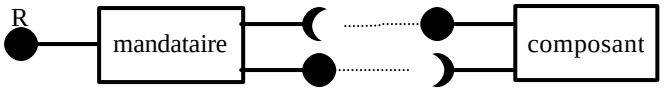
2.1.3. Vision interne

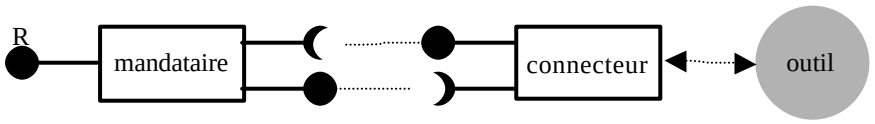
Il n'y a pas une description interne des composants. Le modèle NF va associer une ou plusieurs fabriques à chaque composant, selon le type de composant, pour permettre la création d'instances.

2.2. Le modèle NF

Le modèle NF permet de définir les caractéristiques non-fonctionnelles des divers éléments et relations du modèle logique. Dans la version actuelle du modèle de composants de la fédération, les caractéristiques non-fonctionnelles des composants, dont le modèle d'exécution a besoin sont :

- Type du composant: étant donné que le modèle de composants de la fédération gère la distribution au niveau composition et non pas à niveau coordination, chacun des composants appartient à un des types suivants. Pour chaque type, les informations associés sont différentes.

local	Un composant local s'exécute sur la machine de la fédération. Son modèle NF comporte un nom et une fabrique.
distant	Un composant distant s'exécute sur une machine différente de celle du moteur de la fédération.  Son modèle NF comporte un nom, une fabrique pour le mandataire et une fabrique pour le composant distant.

application	Une application s'exécute dans un processus différent de celui du moteur. Sa principale caractéristique est qu'elle a été construit pour travailler de façon indépendante, et qu'on n'a pas accès à ses sources pour modifier son comportement. Pour cette raison, pour la faire travailler dans le contexte de la fédération, il faut un connecteur. Ce connecteur a notamment deux responsabilités: d'une part, fournir les services attendus par le mandataire, à fin d'implémenter son rôle, et d'autre part, d'écouter les événements ou actions effectuées par l'application, les interpréter et les communiquer vers la fédération sous forme d'appel sur son mandataire.  Son modèle NF comporte un nom, une fabrique pour le mandataire, une fabrique pour le connecteur, une façon de démarrer l'application (la commande de démarrage, par exemple) et, optionnellement, la liste des machines où l'application est disponible.

- Le type démarrage du composant : Définit le moment où une instance de composant doit être créée. Il y a trois instants définis dans le modèle actuellement : au moment de démarrer le moteur de la fédération, au moment de démarrer l'activateur d'une machine qui participe à la fédération ou lorsqu'un autre composant en a besoin.

- Un ensemble de contraintes globales : Définissent un ensemble de conditions globales qui doivent être respectées pendant l'exécution. Notamment, le modèle de composants de la fédération en utilise actuellement deux :

  - Un seul composant dans toute la fédération peut jouer un rôle donné.
  - Une seule instance d'un composant donné peut s'exécuter à la fois.

  Ce type de contraintes permet de déclarer les configurations valides du modèle de composants de la fédération pendant son exécution. Il est envisageable, par exemple, de définir un langage déclaratif de contraintes, avec lequel on pourrait programmer à un très haut niveau des BNF non triviaux, comme par exemple l'équilibrage de charge.

2.3. Le modèle d'exécution

2.3.1. Modèle de base

Le modèle d'exécution est basé sur une machine virtuelle de gestion de composants (CVM – *Component Virtual Machine*) avec deux caractéristiques principales : elle est réflexive et elle est distribuée.

Le fait d'être réflexive lui permet de gérer tout le modèle de composants de manière dynamique : il est possible, par exemple, de déclarer un nouveau composant pendant l'exécution, de modifier son modèle NF, de changer les connexions entre les composants

en cas de besoin, etc. Ces caractéristiques font du modèle de composants de la fédération une excellente plate-forme de travail pour l'adaptation dynamique.

Le fait d'être distribuée, assigne au *framework* du modèle la responsabilité de la gestion de cet aspect non-fonctionnel, en évitant que la distribution soit gérée de manière individuelle par chaque composant. Ceci a plusieurs avantages :

- Le code des composants n'est pas pollué par la gestion de la distribution, ce qui facilite sa réutilisation.

- Les protocoles de communication et les mécanismes de récupération d'erreurs sont partagés par tous les composants, ce qui facilite la programmation et l'évolution.

- Les BNF d'équilibrage de charge, migration de composants, etc. sont la responsabilité du *framework* en non pas de chaque composant, ce qui permettrait de les gérer à niveau déclaratif comme une contrainte globale du modèle NF.

- Comme le *framework* a une vision globale des caractéristiques de la connexion de chaque machine qui participe à la fédération, il peut utiliser des techniques différentes de communication dans chaque cas particulier (pour passer un *firewall*, par exemple), ou gérer des concepts comme le niveau de sécurité d'une connexion.

Ceci ne veut pas dire que tous les composants soient obligés à utiliser la même technologie, ou qu'on impose des restrictions sur son implémentation interne.

Pour faciliter l'adaptation dynamique des participants de la fédération, le modèle d'exécution émet un ensemble d'événements que les composants peuvent écouter pour réagir en conséquence et s'adapter au changement. En particulier, le modèle d'exécution génère un événement à chaque modification du modèle de composants (un nouveau composant, une modification du modèle NF d'un composant), et à la création ou destruction d'une instance de composant.

2.3.2. Représentation d'un composant

Un composant est représenté par un objet chargé d'implémenter les rôles. Son implémentation interne est cachée derrière cet objet.

2.3.3. Navigation

La navigation d'interfaces (rôles) se fait par simple *casting* sur le composant.

2.3.4. Création et localisation d'instances

Dans le modèle d'exécution, à chaque instance de composant est associé un groupe d'attributs physiques (la machine sur laquelle il s'exécute, le type de connexion, etc.) et un groupe d'attributs logiques gérés par le programmeur (un nom, par exemple).

Au moment de vouloir localiser un composant dans la fédération pour une tâche particulière, il est nécessaire de spécifier son rôle, et de donner toutes les caractéristiques

NF attendues, en termes de valeurs pour les attributs physiques et logiques. Si le modèle ne trouve pas une telle instance, il essaie de la créer.

Pour créer une instance, le modèle d'exécution consulte le modèle NF pour établir son type. Avec le type, le modèle utilise les fabriques pour créer l'instance et les éventuels mandataires ou connecteurs. Une fois que ces objets ont été créés et connectés, le modèle assigne des valeurs aux attributs physiques de l'instance de composant, et garde une référence.

En tout moment, le modèle d'exécution garantit le respect des contraintes globales définies dans le modèle NF.

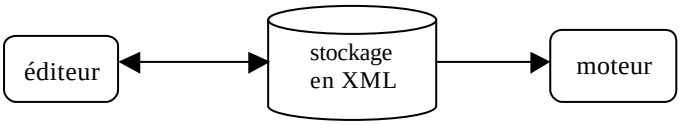
2.3.5. Opérations de service

Tout composant distant est obligé d'implémenter un ensemble d'opérations de service pour gérer la synchronisation avec le modèle d'exécution (e.g. `ping`, `stop`, `synchronize`, etc). Pour faciliter sa programmation, dans le *framework* il y a une implémentation par défaut, dont le composant peut hériter.

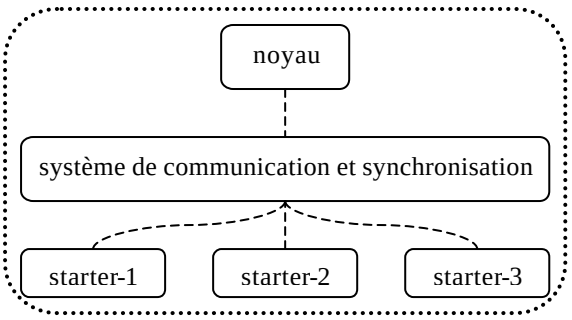
Les composants locaux et les applications n'implémentent pas d'opérations de service. Dans le cas des applications, le connecteur est chargé de les implémenter.

2.4. Le *Framework*

Le *framework* du modèle a deux parties : un éditeur et un moteur. L'éditeur est un outil qui permet au programmeur de définir tous les aspects du modèle de composants (i.e. le modèle logique et non-fonctionnel de chaque composant). Le moteur est une implémentation du modèle d'exécution. Le stockage du modèle de composants se fait en XML.



Du point de vue architecture, le moteur est composé d'un noyau, d'un système de communication et d'un ensemble de programmes client, appelés les starters, qui s'exécutent sur les machines qui participent à une fédération. Chacun de ces éléments sera présenté par la suite:

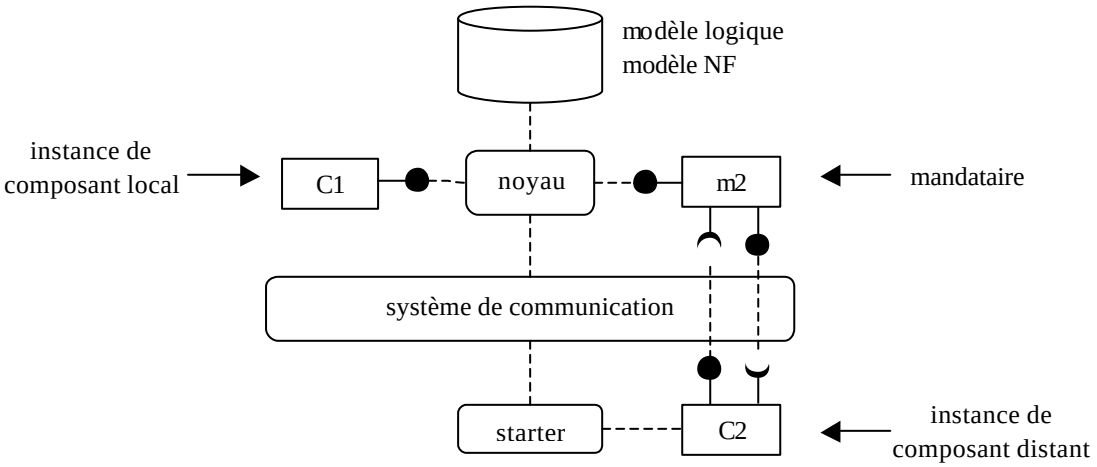


2.4.1. Le noyau du moteur

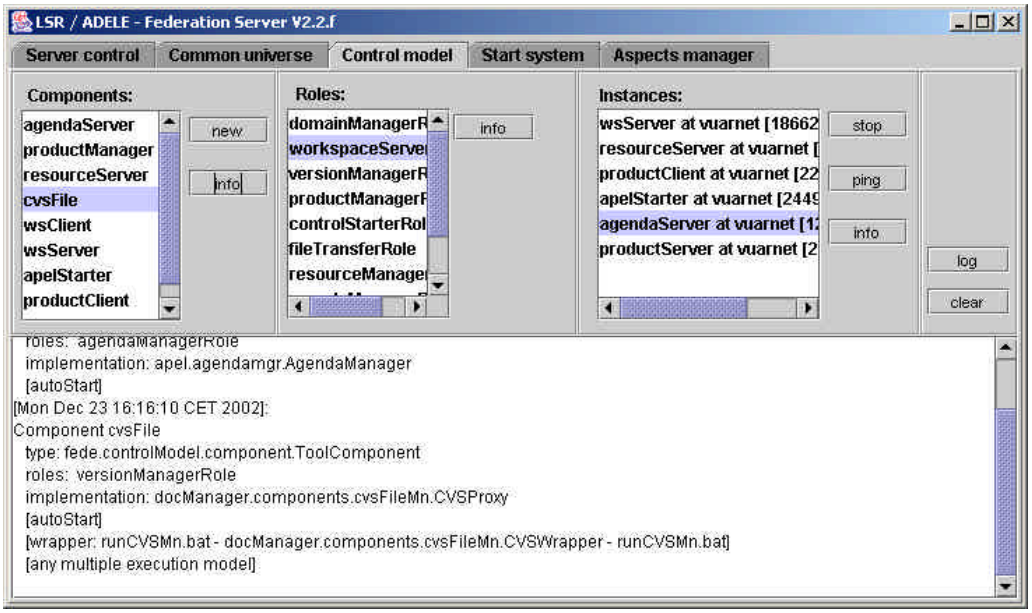
Le noyau du moteur a deux responsabilités principales :

- Gérer le modèle logique et le modèle NF
- Gérer le cycle de vie et la recherche des instances des composants

Dans le cas des composants distants, la responsabilité de gérer le cycle de vie des instances de composants est partagée entre le noyau et le starter. Le noyau contrôle les mandataires, et le starter les instances des composants qui s'exécutent sur cette machine. Il faut préciser que toute la communication entre le mandataire et l'instance de composant est contrôlée par le moteur.



Le noyau dispose d'une interface graphique qui permet l'interaction directe d'un usager avec le modèle de composants (modèle logique et modèle d'exécution), facilitant de cette façon le processus de développement des programmeurs.



Une caractéristique important du noyau est sa capacité de se récupérer en cas de panne. Pour ce faire, le noyau part de l'état de chaque starter pour recréer les mandataires associés, et continuer ainsi l'exécution.

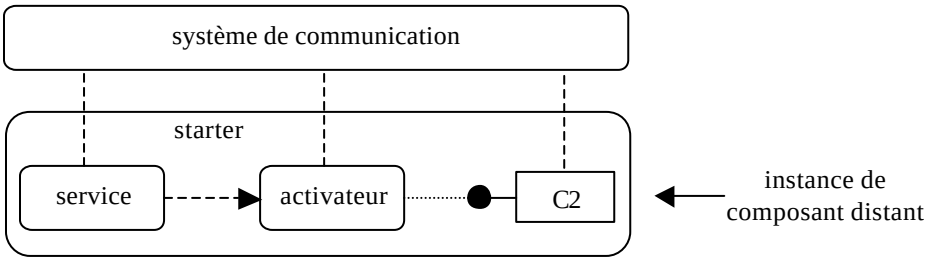
2.4.2. Les starters du moteur

Un starter est le représentant du moteur dans une machine client. Le starter est responsable de :

- Se présenter au noyau, en lui fournissant les informations nécessaires pour sa participation au modèle.
- Gérer le cycle de vie des instances de composant qui s'exécutent sur la machine.
- Dans certaines situations, collaborer avec le système de communications, pour communiquer un mandataire avec son instance.

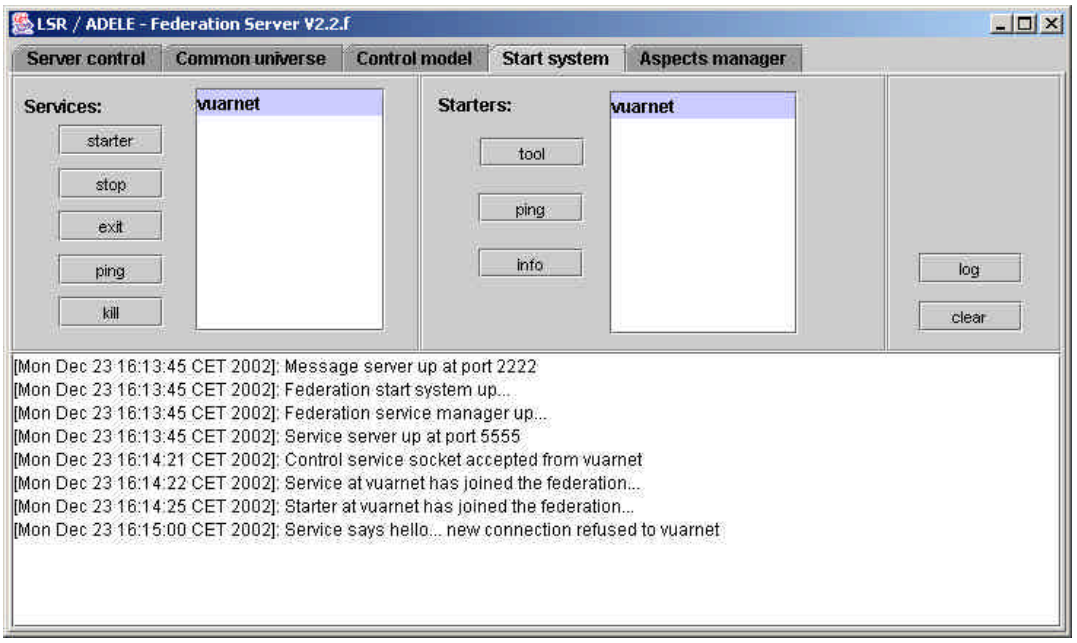
Un starter est constitué de deux applications synchronisées, pour faciliter la récupération en cas de problème. La première application, appelée le service (destinée à s'exécuter sur la machine cliente), est un petit programme connecté au noyau, dont sa seule responsabilité est de gérer le cycle de vie de la deuxième application. Le service a une logique très simple et il a été développé pour que les risques de panne soient infimes.

La deuxième application, appelée l'activateur, assume toutes les responsabilités fonctionnelles du starter.



En cas de problème, le service est capable d'arrêter l'activateur et de se synchroniser avec le noyau pour que l'activateur récupère son état (se redémarrer, et récupérer sa connexion avec les instances de composant qui se exécutent sur la machine). L'activateur utilise les opérations de service du composant pour le localiser et se re-synchroniser.

Le client peut interagir directement avec le système de communication et synchronisation de la fédération, en utilisant son interface graphique : il est possible de démarrer / arrêter un activateur sur une machine, vérifier l'état de la connexion, créer une instance d'un composant sur une machine distant, et, en particulier, exécuter les outils qui font partie d'une fédération.

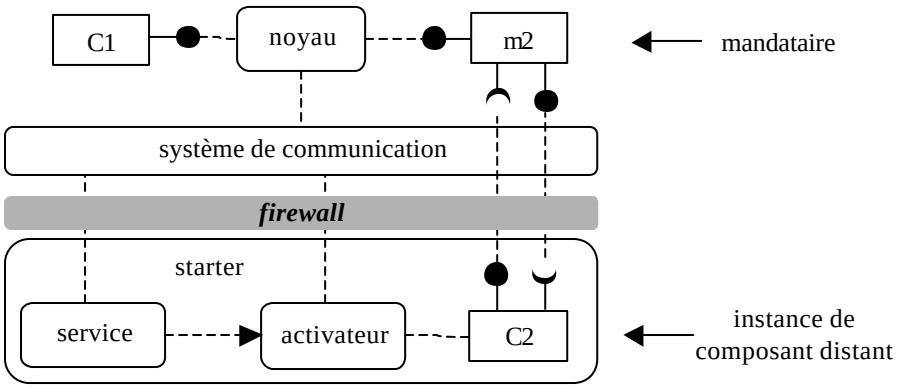


L'activateur peut aussi s'exécuter avec une interface graphique pour permettre au programmeur de visualiser son état, et d'interagir avec lui.

2.4.3. Le système de communication

Le système de communication et de synchronisation est chargé de connecter le noyau avec les starters, et les mandataires avec les instances de composant. Dans le cas le plus simple, lorsqu'il n'y a pas de *firewall* entre le noyau et le starter (le cas typique d'un Intranet), toute la communication se fait par RMI. Dans cette situation, la synchronisation se fait en utilisant le registre (*registry*) de la machine et par des appels des opérations de service des composants.

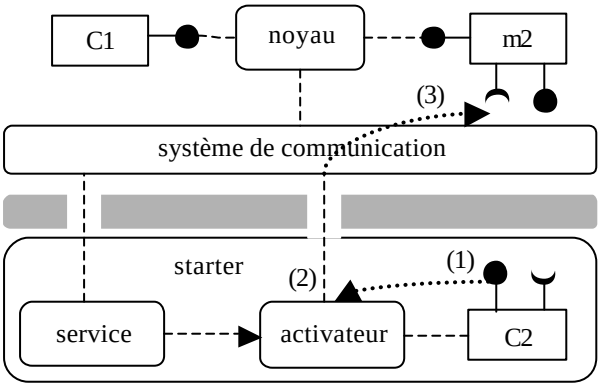
Le deuxième cas possible est qu'il y ait un *firewall* du côté de la machine cliente.



Pour affronter cette possibilité, le système de communication utilise un protocole qui peut se résumer de la manière suivante :

- Le service et l'activateur ouvrent des tunnels dans le *firewall*.

- La communication du noyau vers l'activateur passe par le tunnel, en utilisant un système de messagerie proche de SOAP [SOAP], mais qui n'utilise pas XML par de raisons de performance. Le système de messagerie se base sur la sérialisation de Java. La communication de l'activateur vers le noyau se fait par RMI.



- La communication d'un mandataire vers son instance de composant passe aussi par le tunnel de l'activateur. Dans ce cas, le système de communication et l'activateur font le multiplexage du tunnel pour gérer multiples interactions simultanées. La communication de l'instance de composant vers son mandataire se fait par RMI.

Pour le cas d'un *firewall* dans le serveur et un *firewall* dans le client, nous avons développé un prototype basé sur le protocole HTTP, implémenté en utilisant des `servlets` [SSJ02] chargés de faire la liaison entre le noyau et le starter. Ce prototype fournit la même fonctionnalité au niveau communication, et n'utilise que la connexion standard du port 80 pour un serveur HTTP. Les tests ont été faits avec TOMCAT et Apache comme serveurs. Le prototype n'a pas été intégré au moteur du modèle.

Le système de communication peut afficher sur l'interface graphique du moteur, toutes les communications et les réponses, afin de faciliter le monitoring de l'exécution.

2.4.4. Les bibliothèques de base

Pour simplifier le développement des composants distants, des mandataires et des connecteurs, on a construit un ensemble de clases qui intègrent déjà les fonctionnalités de base et les protocoles de synchronisation et de récupération. Ceci permet au programmeur d'ignorer la plupart des détails de bas niveau, et de se concentrer sur la fonctionnalité du composant.

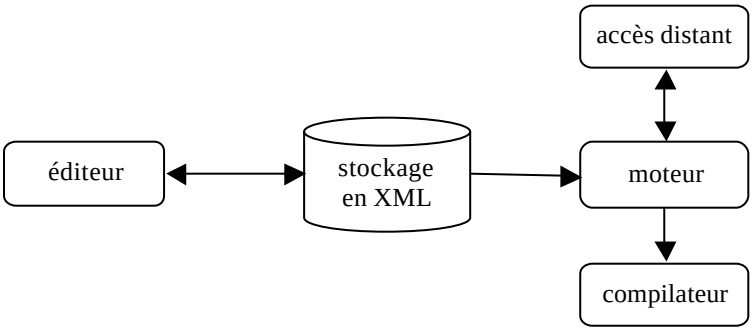
3. L'univers commun et les contrats de coordination

3.1. Architecture

Du point de vue architecture, l'univers commun a été implémenté comme quatre outils indépendants : un éditeur, un compilateur de contrats, un moteur et un connecteur pour l'accès distant.

L'éditeur permet au programmeur de définir les classes qui vont s'exécuter dans l'univers commun, et d'associer les contrats de coordination avec les méthodes de ces classes. Le

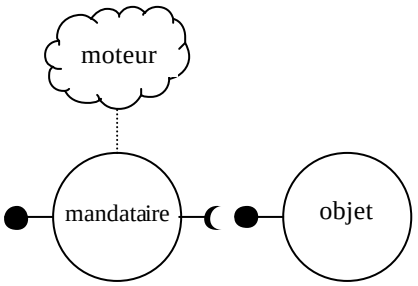
compilateur est utilisé par le moteur, pendant l'exécution, pour traduire du langage des contrats en java. Le moteur, pour sa part, est chargé de matérialiser la description de l'univers commun faite avec l'éditeur. Le connecteur pour l'accès distant est un module qui se connecte au moteur pour gérer la relation entre l'univers commun et les composants qui ne sont pas contrôlés par la fédération.



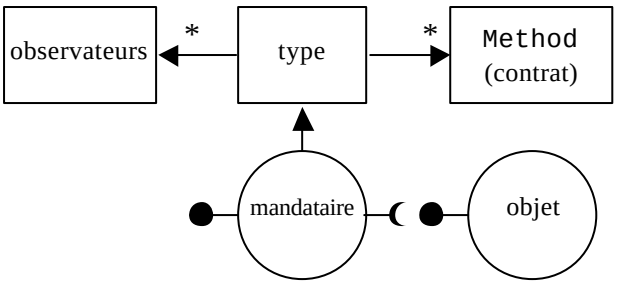
3.2. Le moteur de l'univers commun

Comme nous l'avons montré dans le chapitre 4, il y a plusieurs stratégies pour intercepter les appels des méthodes vers un objet. Cette interception va permettre au moteur d'introduire, dans la logique d'exécution de l'application, les appels aux contrats de coordination.

Nous avons choisi une implémentation qui utilise le mécanisme de mandataires dynamiques de java (*proxies dynamiques* [DPC]). Le moteur, au moment de recevoir la demande de création d'une instance dans l'univers commun, installe un mandataire généré dynamiquement entre le client (celui qui demande la création de l'instance) et l'objet créé.



A chaque appel d'une méthode, le mandataire passe le contrôle au moteur pour vérifier s'il faut exécuter un contrat de coordination et s'il y a des observateurs à notifier. Le moteur, pour sa part, a déjà traduit les contrats en termes de méthodes Java, et a laissé des objets de la classe *Method* prêts à être exécutés. Ceci évite tout processus d'interprétation.

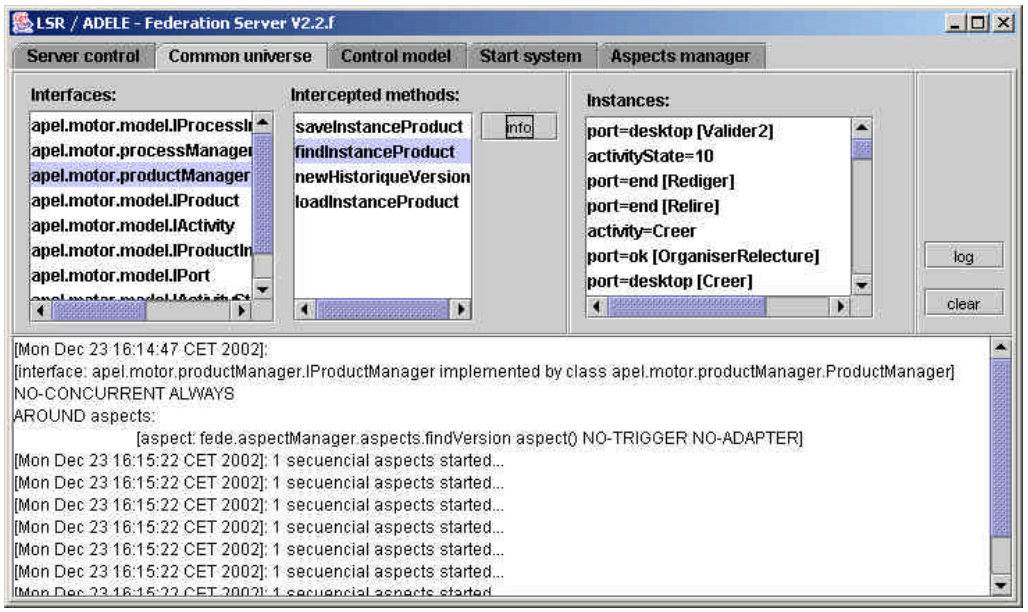


Le moteur émet deux types d'événements pour ses observateurs: D'une part il notifie les appels des méthodes sur les objets de l'univers commun, et, d'autre part, notifie la création et destruction d'instances.

Le moteur gère le concept de racine de l'univers commun, ce qui permet d'associer un nom avec une instance particulière. Ceci permet à un composant de localiser un objet par son nom, et de se synchroniser de cette façon avec d'autres composants.

L'implémentation actuelle de l'univers commun n'est pas persistante, ce qui implique que la responsabilité de persister appartient à chaque classe. Pour faciliter cette tâche aux classes de l'univers commun, le moteur implémente un ensemble de services de base.

Le moteur a une interface graphique, qui permet en tout moment au programmeur de visualiser le contenu et l'exécution de l'univers commun (classes, contrats et instances) et d'interagir avec lui.

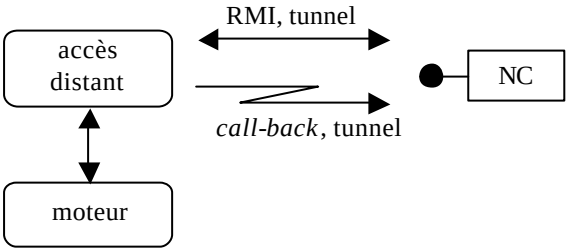


Il y a une autre implémentation disponible de l'univers commun, qui utilise le processus de modification du code de la machine à objets étendus [DES02]. Au moment d'écrire cette section cette implémentation est en phase finale de tests.

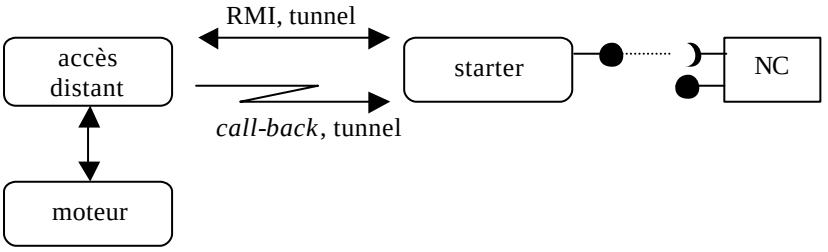
3.3. L'accès distant à l'univers commun

Pour permettre l'accès à l'univers commun de composants qui ne font pas partie du modèle de composants (appelés composants non-contrôlés), on a introduit un module additionnel. Ce module se connecte au moteur, et agit comme le mandataire de tous les composants non-contrôlés.

Les composants peuvent communiquer avec ce module soit par RMI, soit par des *sockets*, de telle façon que, même si le composant se trouve derrière un *firewall*, le composant a accès aux objets et aux notifications de l'univers commun.



Pour éviter les problèmes d'implémentation de ce type de composants (surtout s'ils sont distants), on a rajouté au starter du moteur du modèle de composants un ensemble de services pour permettre au composant d'accéder "localement" aux services d'accès distant de l'univers commun.



Chaque objet de l'univers commun a un identifiant unique assigné, de telle façon que les composants non-contrôlés puissent interagir avec eux, sans la contrainte d'avoir à gérer des objets distants dans l'univers commun.

3.4. Le langage et le compilateur de contrats

Nous avons développé un langage simple pour exprimer les contrats de coordination, basé sur la syntaxe de java. Ce langage a deux objectifs principaux :

- D'abord, rendre indépendants le moteur de la fédération et le langage utilisé pour définir les contrats. Ceci va nous permettre d'utiliser d'autres langages à condition d'écrire le compilateur nécessaire.

- Le deuxième objectif est d'étudier le problème de composition de contrats. Le langage a été défini pour qu'il soit possible de faire un tissage de contrats à bas niveau, en suivant la stratégie montrée dans le chapitre 5.

Une partie du langage est exprimé de manière graphique dans l'éditeur de contrats. Pour le reste, en particulier en ce qui concerne la description des participants et le corps des contrats, il y a une syntaxe textuelle. Le modèle global du langage proposé peut s'expliquer avec la pseudo-BNF suivante :

<instrumentation> ::= <règles, `before`: {contrat}, `after`: {contrat}, `around`: {contrat}>

L'instrumentation correspond à "l'extension" d'une méthode de l'univers commun par un ensemble de contrats de coordination. Nous appellerons "contrat global" cette instrumentation.

<règles> ::= < acceptation, exécution >

La règle d'acceptation définit quand le contrat global est satisfait :

<acceptation> ::= `AND` | `OR` | `ALWAYS`

- `AND` : Tous les contrats doivent être respectés pour satisfaire le contrat global
- `OR` : Au moins l'un des contrats doit être respecté pour satisfaire le contrat global
- `ALWAYS` : Le contrat global est toujours satisfait

La règle d'exécution définit comment le contrat global doit se dérouler (ordre d'exécution):

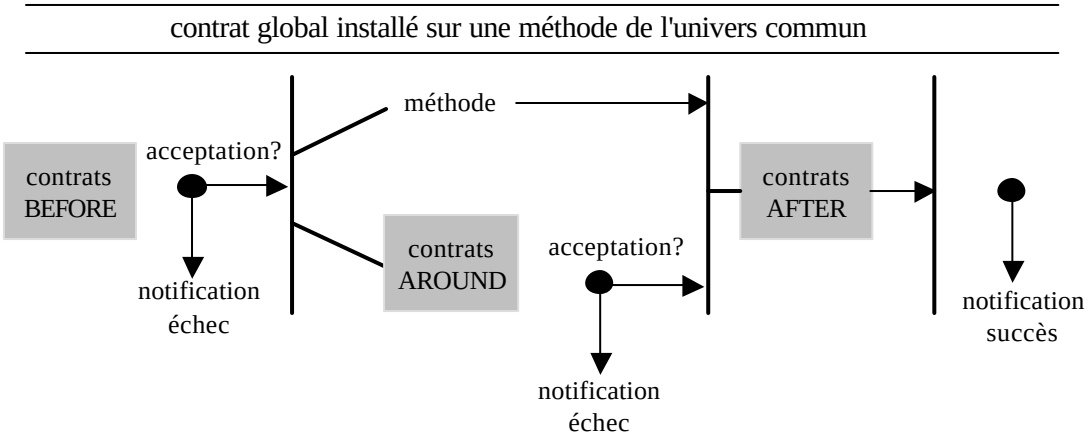
<exécution> ::= `parallel` | `sequential`

- `parallel`: Tous les contrats s'exécutent de façon concurrente
- `sequential` : Les contrats s'exécutent l'un après l'autre, en suivant l'ordre de déclaration

Un contrat global s'exécute de la façon suivante:

- Exécution de tous les contrats `before` en mode <exécution>, dans l'ordre dans lequel les contrats apparaissent dans la liste.
- Si ! <acceptation>
  notification de l'échec et fin du contrat global
- S'il y a des contrats `around`
  { exécution de tous les contrats `around` en mode <exécution>
  si !<acceptation>
  notification de l'échec et fin du contrat global
  }
  sinon
  exécution de la méthode interceptée
- exécution de tous les contrats `after` en mode <exécution>
- notification aux observateurs de l'appel de la méthode

La figure suivante résume le protocole implémenté pour l'exécution d'un contrat global :



Les contrats sont définis avec une syntaxe textuelle, et se stockent dans des fichiers indépendants. Un contrat représente une facette du contrat global, et il est définit par un ensemble de participants et par un groupe d'actions que chacun d'entre eux doit exécuter.

<contrat> ::= < nom, type, [adaptateur], [trigger], { participant }, corps >

<type> est le type d'objet de l'univers commun dont la méthode va être interceptée

L'adaptateur est une fonction qui explique la manière de traduire (si nécessaire) la signature de la méthode interceptée dans la signature du contrat. Ceci permet la réutilisation des contrats sur différentes méthodes de l'univers commun.

Soit $m(p_1, \ldots, p_k)$ la méthode interceptée, et soit $contrat(q_1, \ldots, q_n)$ la signature d'un contrat. Un adaptateur est une fonction $P_1 \times \ldots \times P_k \rightarrow Q_1 \times \ldots \times Q_n$, où $p_i \in P_i$ et $q_i \in Q_i$.

Le trigger est un prédicat exprimé en termes des paramètres du contrat et en termes du contexte d'exécution (voir section 3.5), qui explique les cas dans lesquels le contrat doit être exécuté. Si le trigger évalue à faux, le contrat n'est pas lancé, mais il est considéré comme satisfait. On dit alors que le contrat ne s'applique pas à la situation actuelle.

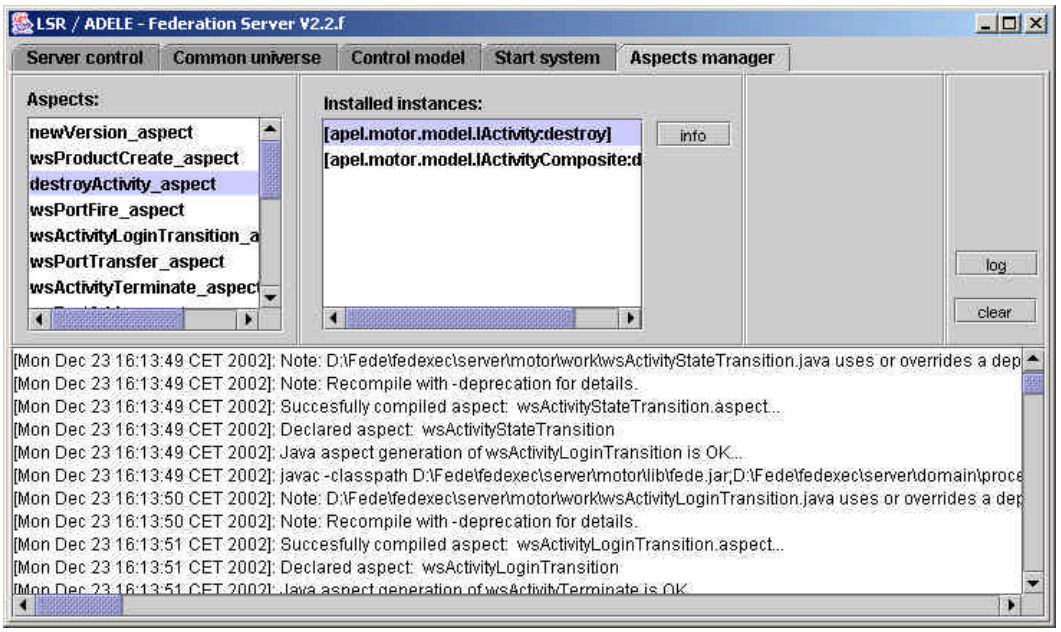
Soit $contrat(q_1, \ldots, q_n)$ la signature d'un contrat. Le trigger est un prédicat avec domaine $C \times Q_1 \times \ldots \times Q_n$, où C est le contexte d'exécution, et $q_i \in Q_i$

Un participant correspond à une instance de composant (calculée au moment d'exécuter le contrat), décrit par son rôle, et par un ensemble de caractéristiques physiques et logiques exigées à cette instance pour participer dans le contrat. Chaque participant a un nom, utilisé dans le corps du contrat pour expliquer les actions qui le concernent. Si le modèle de composants n'arrive pas à trouver une instance avec ces propriétés, le contrat échoue.

Le corps du contrat est une séquence d'appels vers les divers services fournis par les rôles des participants, des invocations vers les objets de l'univers commun, et des consultations au contexte d'exécution.

Le compilateur est contrôlé par le moteur de l'univers commun. Il fait la traduction du langage de contrats à java, appelle le compilateur de java et installe des instances des objets résultants sur les objets de l'univers commun. Au moment de démarrer, le moteur vérifie si la date de modification du fichier du contrat est plus récente que celle du fichier java généré, et décide s'il faut re-déclencher le processus de compilation.

L'interface graphique du moteur de la fédération fournit une façon de visualiser et d'interagir avec les contrats de coordination. Dans cette interface, chaque contrat est appelé un aspect. Dans la figure suivante on peut voir que l'aspect `destroyActivity` est installé sur deux méthodes de l'univers commun : la méthode `destroy` de l'interface `IActivity`, et sur la méthode `destroy` de l'interface `IActivityComposite`.



Un exemple d'un contrat utilisé dans l'application de gestion documentaire, qu'on va présenter dans la section 6 de ce chapitre, est le suivant :

```
contract wsProductOpen( String machineName ) of IProductInstance
{
      when instance.getPort() != null

      components
      {
            workspaceManager: workspaceServerRole at machineName
      }

      body( JAVA )
      {
            IProductInstance  productInstance   = instance;
            IPort port = productInstance.getPort();
            IActivity activity = port.getActivity();
            if ( port.getType()!=IPort.DESKTOP || !activity.isActive() )
                  end;
            workspaceManager.loadProduct( productInstance );
      }
}
```

Ce contrat est associé avec la méthode `open` de toutes les instances de la classe `ProductInstance` qui s'exécutent dans l'univers commun, et demande à un outil avec le rôle `workspaceManagerRole` de charger les fichiers concernés par le produit, sur une machine donnée.

Dans la syntaxe de l'exemple il est possible de distinguer le trigger (introduit avec le mot clé **`when`**), les participant (dans le groupe de **`components`**) et le corps (définit avec le mot clé **`body`**).

3.5. Le contexte d'exécution d'un contrat

Au moment d'exécuter un contrat global, le moteur génère un objet appelé le contexte d'exécution, qui permet aux contrats de se synchroniser et de partager des informations. Le contexte d'exécution comporte :

- L'objet de l'univers commun intercepté.
- Le mandataire de l'objet intercepté.
- La méthode interceptée (un objet de type `Method`).
- Un booléen qui indique si la méthode interceptée a été déjà appelée.
- Le résultat de l'appel de la méthode (`null`, si la méthode n'a pas été appelée).
- L'ordre d'exécution des contrats (`BEFORE`, `AFTER`, `AROUND`).
- Un ensemble de couples variable – valeur, qui permet aux contrats de partager des objets pendant l'exécution du contrat.

Tous les contrats reçoivent le même objet, et disposent des opérations de consultation et de modification des éléments du contexte. Les contrats ont aussi accès à la méthode `proceed( Object[ ] )`, qui permet de faire l'appel de la méthode interceptée, dans le schéma `AROUND`.

Dans notre expérimentation, nous avons utilisé souvent les variables du contexte comme base pour interpréter les changements d'état produits par l'appel d'une méthode.

4. Le monde du contrôle : APEL

4.1. Description

Dans les applications construites pour expérimenter avec les idées des fédérations, nous avons utilisé l'outil APEL [Vil02, DEA98] (développé dans le laboratoire LSR), comme l'implémentation du monde du contrôle, un monde actif chargé de guider l'application globale.

APEL permet la définition d'un procède (i.e. un graphe d'activités) en utilisant un langage graphique. Avec cette syntaxe, le programmeur peut définir, à un haut niveau d'abstraction, la "manière" de résoudre un problème concernant plusieurs mondes. Lors de l'exécution, le moteur de l'outil interprète le langage graphique et génère les actions nécessaires pour déclencher les contrats de coordination.

La première version d'APEL a été développée en 1997. Depuis cette date, six nouvelles versions ont été livrées, la dernière au début de l'année 2003. Dans la dernière version, nous avons fait un effort particulier pour simplifier l'architecture du moteur, et résoudre ainsi quelques problèmes de performance présentes dans les versions antérieures.

APEL est un outil crée pour travailler dans le contexte d'une fédération. Pour ce faire, il travaille avec des abstractions des concepts, tels que donnée ou responsable, et laisse aux autres mondes la tâche de concrétiser sa réelle signification. En particulier, en utilisant la composition de mondes de la fédération, APEL peut être étendu pour s'adapter à plusieurs contextes différents.

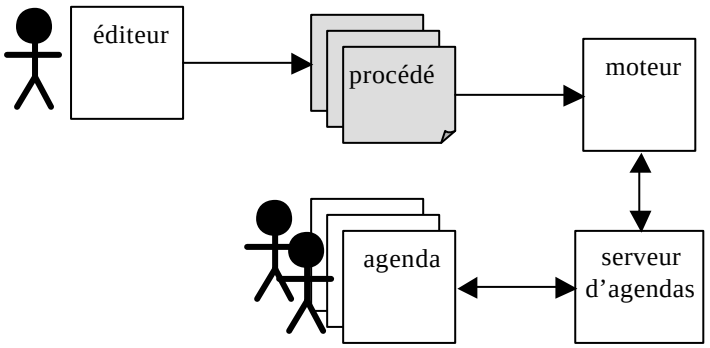
Dans cette section nous présenterons l'outil APEL, son architecture, sa syntaxe graphique et son meta-modèle, ce dernier utilisé plus tard comme base de la composition avec les autres mondes.

4.2. Architecture

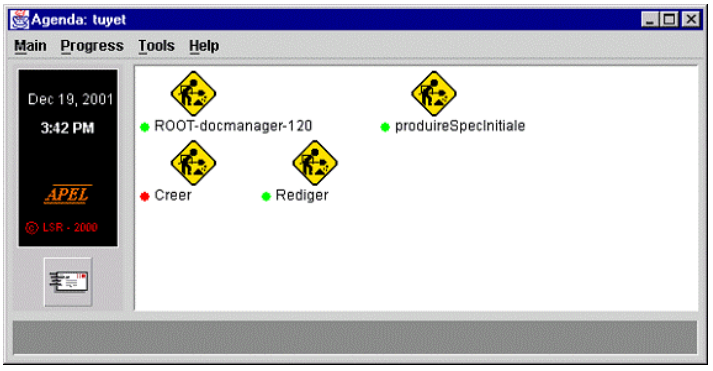
APEL est composé de quatre outils différents :

- Un éditeur, qui permet de définir de manière graphique la structure et les éléments d'un procédé. Chaque procédé créé avec l'éditeur est stocké sous forme de fichier XML.
- Un moteur, chargé d'interpréter les procèdes créés par l'éditeur.
- Un agenda, qui est un outil qui s'exécute sur le poste d'un client et qui lui permet de recevoir et d'exécuter des activités d'un procédé.
- Un serveur d'agendas, qui synchronise le travail entre le groupe d'agendas et le moteur d'APEL.

La figure suivante illustre l'architecture d'APEL :



Lorsqu'une activité est assignée à une personne, sur son agenda elle va recevoir une icône à partir de laquelle elle a accès à la tâche, avec les instructions nécessaires.

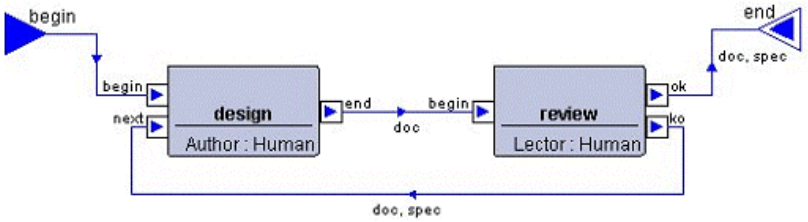


4.3. Syntaxe graphique

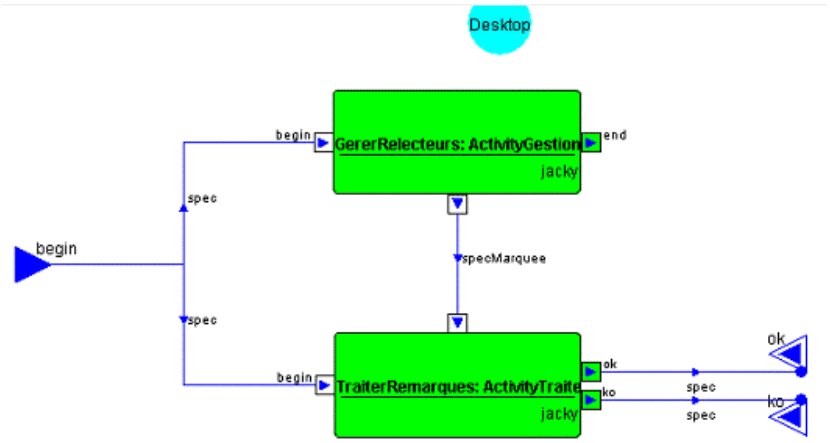
APEL se base sur un petit ensemble de concepts et utilise une syntaxe graphique très intuitive, ce qui permet à l'utilisateur de définir un procédé rapidement. Le tableau suivant, montre un résumé des concepts et la syntaxe graphique associée. Dans [Vil02] il y a une spécification formelle complète du fonctionnement du moteur (i.e. de la sémantique du langage graphique) :

Activité : Une activité est un pas dans un procédé qui contribue à la réalisation d'un objectif. Une activité, du point de vue fonctionnel, peut être considérée comme une transformation d'un ensemble de données fournis en entrée vers un ensemble de données de sortie, qui représentent le résultat de la tâche. Les activités peuvent être simples ou composites, et elles ont une cardinalité dans le cas des activités multiples.	
Ports synchrones d'entrée et de sortie : Les ports, en général, définissent les possibles points d'entrée et de sortie des activités. Chaque port est caractérisé par l'ensemble de donnés attendus. Chaque donnée, de sa part, a un type. Une activité peut commencer, si dans un des ports d'entrée il y a toutes les données attendues. Une activité peut terminer, si dans un des ports de sortie il y a toutes les données attendues.	
Ports asynchrones d'entrée et de sortie : Les ports asynchrones sont une source additionnelle de données pour les activités. La différence est qu'ils ne sont pas utilisés pour définir l'ordre d'exécution des activités, mais pour permettre à deux activités qui s'exécutent simultanément d'échanger des données. Les ports asynchrones ont aussi un ensemble de données attendues.	
Flot de données : Les flots de données montrent comment les données circulent entre les ports des activités. Les flots de données peuvent être considérés comme les arcs du graphe défini par le procédé.	
Poste de travail (*desktop*) : Chaque activité a un poste de travail, où le responsable de la tâche va trouver les données nécessaires pour pouvoir la faire. Les flots de données doivent connecter les ports d'entrés de l'activité avec le respectif poste de travail, pour faire passer les données.	

La figure suivante montre un extrait d'un procédé avec deux activités (`design` et `review`), et un ensemble de flots de données pour les connecter et expliquer la manière de faire passer les données d'un port à un autre. L'activité `design`, par exemple, a deux ports synchrones d'entrée (`begin` et `next`). Chaque flot de données montre les données (`doc` et `spec` dans l'exemple) qu'il fait circuler entre les ports.



Pour illustrer les ports asynchrones, nous montrons dans la figure suivante une partie du procédé de l'application de gestion de documents qui sera présentée dans la section 5 de ce chapitre. Sur cette figure, on peut voir deux activités qui s'exécutent en parallèle, et que pendant leur exécution la première activité (`GererRelecteurs`) va envoyer une donnée (`specMarquée`) à la deuxième activité (`TraiterRemarques`).

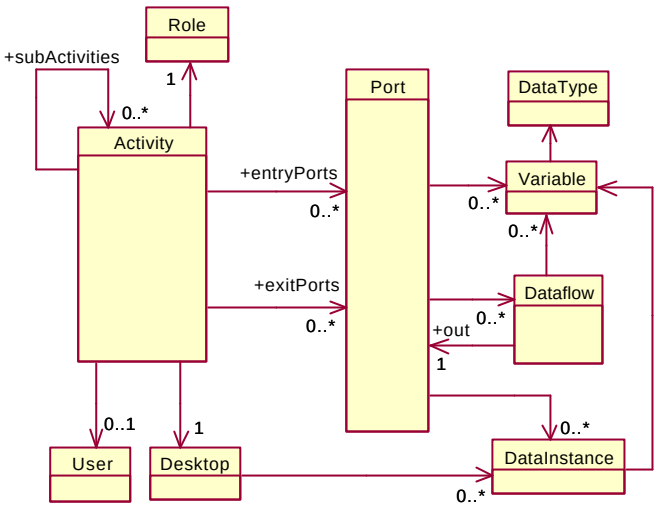


APEL supporte d'autres concepts, qui ne seront pas présentés dans ce document.

4.4. Meta-modèle du monde

Dans cette partie nous montrons une partie du meta-modèle du monde géré par APEL. En simplifiant, on peut dire que un procédé est une hiérarchie d'activités interconnectées par un ensemble de flots de données. Chaque activité a un responsable, d'un type défini par son rôle, et chaque donnée, attendue dans un port, a un nom et un type.

Par simplicité nous avons enlevé du meta-modèle les ports asynchrones, et les concepts qui ne seront pas utilisés dans les exemples des sections suivantes comme des éléments de composition.



4.5. Caractéristiques du moteur

A propos de l'implémentation actuelle du moteur, il y a quelques caractéristiques qu'il faut souligner :

- L'implémentation a été faite pour permettre l'installation de *proxies* dynamiques entre les instances. Ceci nous a permis d'automatiser les notifications d'activité vers le monde de la coordination.
- Le moteur a une spécification sémantique complète. Dans l'implémentation, il y a un contrôle permanent des pré-conditions et des invariantes des classes. De cette façon, nous avons réussi à stabiliser rapidement le comportement de l'outil.
- Le moteur est capable de se récupérer en cas de panne, et de guider le monde de la coordination pour que l'application globale récupère aussi son état. Cette fonctionnalité a été développée en se basant sur les mécanismes de récupération du moteur de la fédération.

5. Une application de gestion documentaire

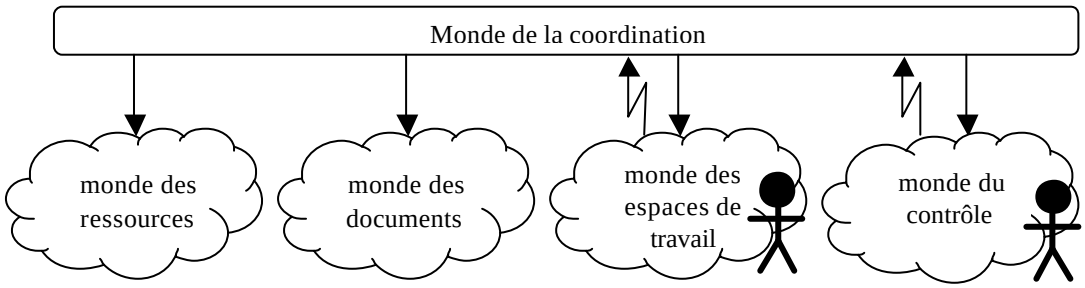
5.1. Objectifs et cahier de charges

La principale application utilisée pour évaluer notre approche est un gestionnaire générique pour la création coopérative de tout type de document (simple ou structuré). Ici nous n'allons présenter que les aspects les plus importants de l'application, comme un moyen d'illustrer les idées exposées tout au long de ce document.

L'application fait partie du projet Centr'Actoll [Actoll], et les besoins fonctionnels ont été définis en considérant les politiques de production de documents de la société Actoll. Du point de vue non fonctionnel, les principales contraintes sont la facilité d'évolution (les procédés peuvent changer à tout moment) et que l'application puisse s'adapter aux divers outils utilisés par les clients.

L'application est en exploitation actuellement pour contrôler le processus de création de documents dans la société Actoll.

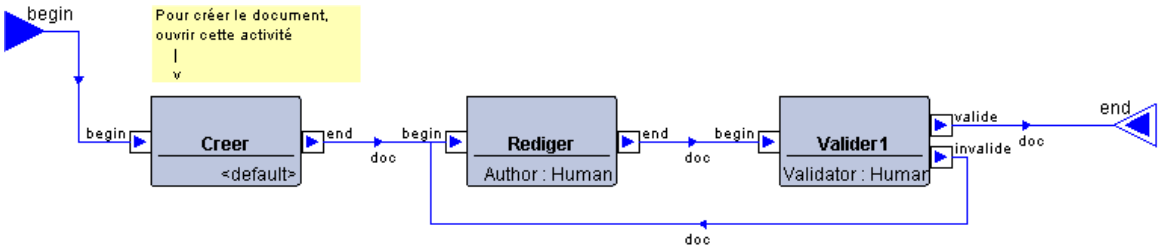
La figure suivante résume la vision abstraite de l'application : (1) le monde du contrôle (implémenté avec l'outil APEL), qui guide l'application pour arriver à une solution, (2) le monde des ressources, qui administre un ensemble de ressources disponibles (en particulier des personnes), (3) le monde des documents, qui stocke et gère un ensemble de documents et leurs différentes versions, et (4) le monde des espaces de travail, qui permet à une personne de travailler sur un ensemble de produits. Il y a deux mondes actifs dans cette fédération : le monde du contrôle et le monde des espaces de travail. Le monde du contrôle est en contact avec les utilisateurs à travers les agendas. Le monde des espaces de travail, à travers la gestion de l'ensemble d'éléments (documents dans notre cas) sur le poste de travail de l'utilisateur.



Dans la suite de cette section nous montrerons chaque monde, son meta-modèle, et les dépendances conceptuelles entre les concepts.

5.2. Le modèle de procède

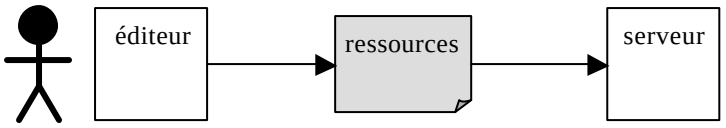
Le procédé défini pour représenter le processus de création d'un document comporte 12 activités, séparées en deux étapes : une étape initiale de création du document, et une étape de relecture et validation. La première étape est décrite avec le procédé suivant :



La deuxième étape a neuf activités structurées à deux niveaux.

5.3. Le monde des ressources

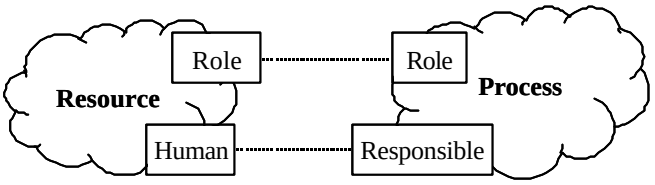
Le monde des ressources est implémenté avec deux outils : un éditeur de ressources et un serveur. Avec l'éditeur on peut décrire les ressources d'une organisation. Le serveur, de sa part, fournit un accès au modèle de ressources défini par l'éditeur.



Le sous-ensemble du meta-modèle dont nous sommes intéressés, pour pouvoir le composer avec les autres mondes est présenté dans la figure qui suit :

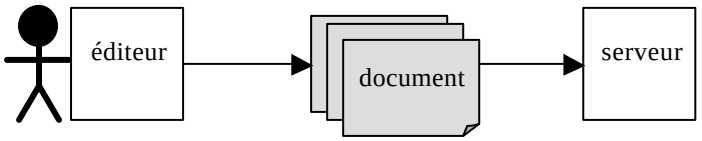
Une personne joue un ou plusieurs rôles, et chaque rôle est joué par zéro ou plusieurs personnes.

Dans une première vision de la composition, on peut dire que les responsables des activités des procédés seront choisis entre toutes les personnes qui jouent ce rôle dans le monde des ressources. Ceci nous amène à définir, dans l'application, une dépendance entre le concept de rôle dans les deux mondes, et entre les concepts de "Responsable" et de "Humain", tel qu'il est suggéré dans la figure :



5.4. Le monde des documents

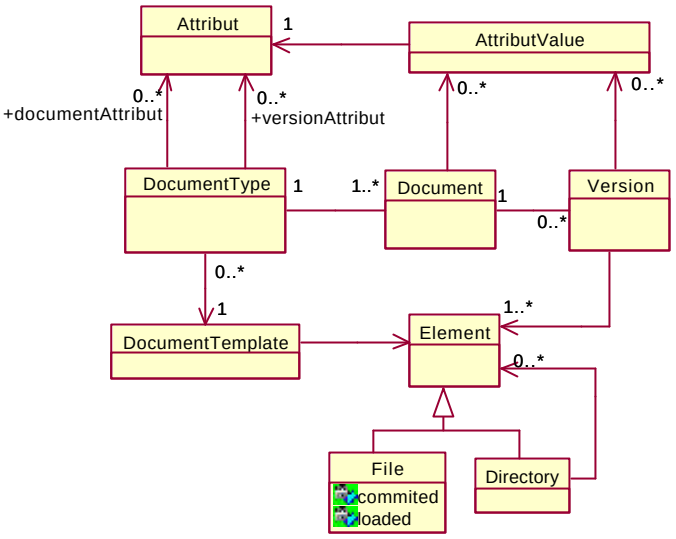
Le monde des documents a une architecture semblable a celle du monde des ressources : un éditeur et un serveur.



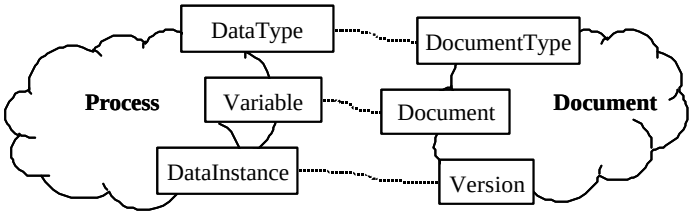
Dans le monde des documents, chaque document a un type, et il est composé par un ensemble de descripteurs (exprimés en termes de couples attribut – valeur), et par une structure initiale (*template*), décrite comme une hiérarchie de fichiers. Chacun de ces documents peut avoir à la fois plusieurs versions, chacune d'elles ayant son propre jeu de descripteurs additionnels et une hiérarchie de fichiers représentant son contenu.

Du point de vue de l'implémentation, le monde des document a été construit avec l'outil CVS (un client et un serveur) et une base de données de descripteurs des documents (avec leurs diverses versions).

Le meta-modèle du monde peut se résumer avec le diagramme de classes montré dans la figure suivante :

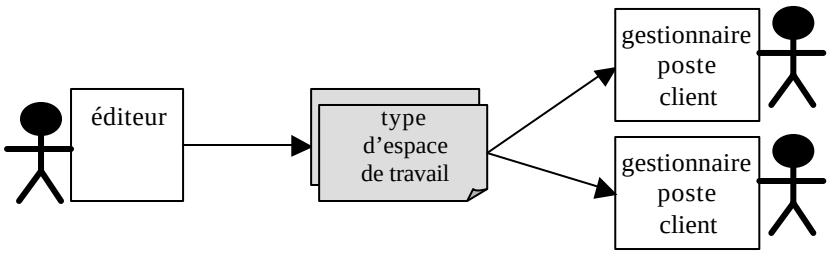


Il y a plusieurs relations conceptuelles entre le monde des procédés et le monde des documents. En particulier, parce que nous voulons que les données qui circulent entre les activités (à travers les flots de données) soient des documents (des versions des documents pour en être précis). Ceci nous génère les relations suivantes :



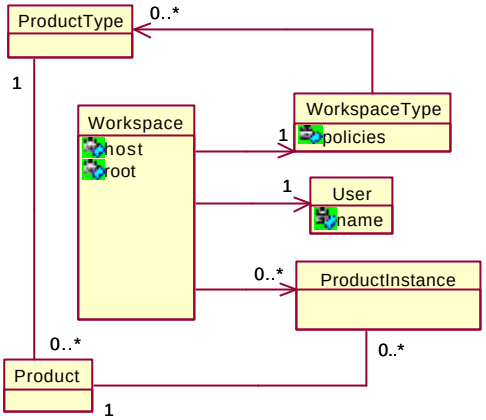
5.5. Le monde des espaces de travail

Ce monde est implémenté aussi avec deux outils : un éditeur de types despaces de travail, et un interpréteur qui va s'exécuter sur la machine de chaque client pour gérer ses espaces de travail.

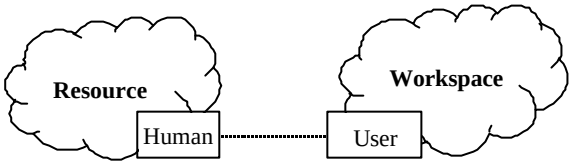


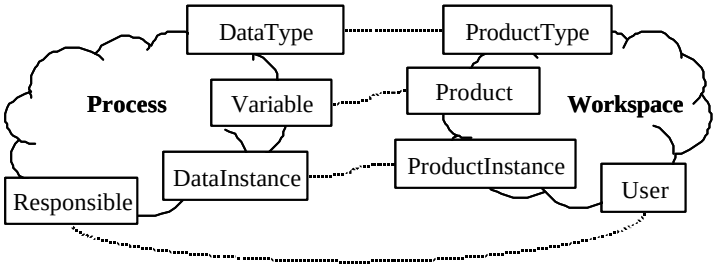
Le monde des espaces de travail associe, sous un type, un ensemble de politiques concernant la gestion des produits qu'il contient et un ensemble de types de produits à gérer. Dans l'implémentation actuelle, les politiques font référence, par exemple, au droit de l'utilisateur à créer ou à détruire les instances de produit présentes dans un espace de travail.

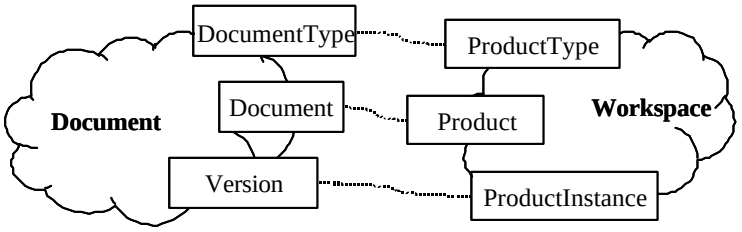
Un espace de travail est donc définit par un type, par un utilisateur (le propriétaire de l'espace de travail) et par un ensemble d'instances de produit.



Le monde des espaces de travail a des dépendances conceptuelles avec les autres trois mondes :

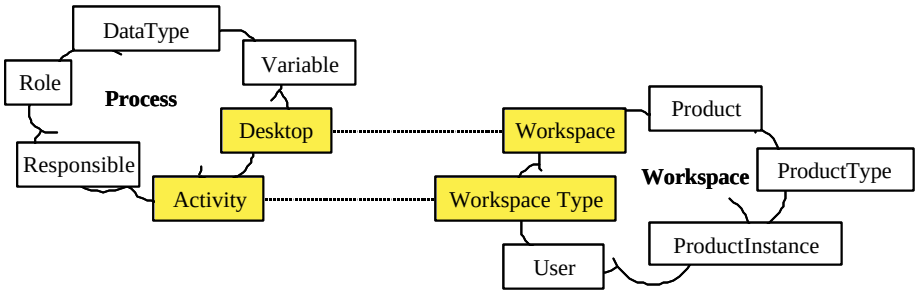






Pour cette raison, il faut associer des contrats de coordination dans la fédération avec chaque couple de concepts dépendants (et parfois entre trois concepts, comme dans le cas du triplé responsable – utilisateur – humain).

Le monde des espaces de travail a aussi deux associations de compositions, introduites pour représenter des relations additionnelles définies dans l'application :



Nous voulons créer une association entre chaque activité d'un procédé et un type d'espace de travail, et une association entre le *Desktop* d'une activité et un espace de travail physique lors de l'étape d'exécution.

Un contrat de coordination, par exemple, chargé de gérer la dépendance entre un espace de travail et le *Desktop*, établit que si une instance de donnée est supprimée du Desktop d'une activité, dans l'espace de travail du responsable de l'activité, il faut effacer la hiérarchie de fichiers correspondants.

Dans cette application, il a 18 contrats de coordination, chargés de guider les outils pour garder une cohérence globale et pour atteindre les objectifs fixés pas le monde du contrôle.

5.6. Résultats

Avec cette application nous avons réussi à construire une fédération solide et performante, en utilisant les principes de composition par coordination et les outils (éditeurs, interpréteurs, compilateurs, etc.) qui font partie du *framework* de support de la fédération.

Dû à la constante évolution des besoins fonctionnels et non-fonctionnels de l'application nous avons pu constater les avantages de l'approche.

6. Autres applications

Trois applications additionnelles ont été développées dans le laboratoire (dans le contexte de différents projets), en utilisant la fédération comme architecture. Dans tous les cas, les résultats ont été très positifs et encourageants.

Une brève description de chacune de ces applications est la suivante :

- Une application dans le domaine de l'ingénierie concurrente. Développée par Sergio Garcia [Gar02]. L'objectif est d'enrichir la gestion des espaces de travail avec une vision de travail en équipe. Ceci permet de gérer le concept de document partagé, et de définir les politiques de synchronisation et de fusion entre les diverses copies d'un même document sur plusieurs espaces de travail.

- Un outil pour automatiser le déploiement à grande échelle d'applications. Développée par Vincent Lestideau [Les02]. L'objectif est de construire une fédération pour faire collaborer un ensemble d'outils (installateurs, outils de diagnostique, outils de transport de fichiers, etc.), pilotés par des procédés APEL. Cet outil permet le déploiement distant d'applications à partir d'un modèle d'application et d'un ensemble de stratégies et de politiques configurables. Dans cet outil, les mondes sont indépendants, et il n'y a aucune composition : toute la coordination se fait par des contrats de coordination déclenchés par le moteur d'APEL.

- Une application pour faire inter-opérer les systèmes d'information de deux sociétés de gestion d'autoroutes. Cette application n'utilise pas APEL, car les deux mondes (i.e. les deux systèmes d'information) sont actifs. Au moment d'écrire ce document, nous sommes dans la phase de définition de l'architecture.

7. Ma contribution

Dans cette section, je fais une description de ma participation concrète dans l'expérimentation et la validation des propositions faites tout au long de la thèse :

- Développement du moteur de la fédération. Ceci inclut (1) l'implémentation du *framework* du modèle de composants, (2) l'implémentation du système de synchronisation et de communications, (3) l'implémentation de l'univers commun et (4) l'implémentation du compilateur du langage de contrats.

- Développement de la nouvelle version du moteur d'APEL. Ceci inclut la spécification formelle de la sémantique du moteur, et son implémentation.

- Participation dans le développement de l'application de gestion documentaire. En particulier l'implémentation de quelques outils du monde des documents, et la participation dans la définition de l'application et de son architecture de coordination.

- Implémentation des tests de compatibilité entre la fédération et les différentes technologies existantes (CORBA, WEB Services, gestionnaires de bases de données, EJB, etc).

8. Conclusions et discussion

Par rapport aux trois objectifs fixés au début du chapitre, nous pouvons affirmer que :

- Nous avons démontré de la viabilité technique des fédérations, pour une utilisation industrielle (objectif 1). Nous avons réussi à trouver une solution aux problèmes liés à la mise en œuvre des idées de coordination de composants.

- Nous avons développé des applications pour étudier les aspects méthodologiques de la construction de fédérations de composants (objectif 2). Nous considérons qu'il y a encore beaucoup de travail à faire dans cet axe, car il y a des aspects de la problématique jusqu'à maintenant ignorés. Nous espérons qu'avec la dernière application, mentionnée à la fin de la section 6 (sur l'inter-opération des systèmes

d'information de deux sociétés de gestion d'autoroutes), nous pourrons compléter l'expérimentation. Pour l'instant, nous pouvons dire que les résultats sont très positifs, et qu'ils nous ont permis de comprendre l'intérêt pratique de nos recherches.

- Pour le troisième objectif, qui concerne la définition de formalismes et outils pour aider à gérer la complexité conceptuelle d'une fédération, nous sommes un peu plus loin. Avec le développement de l'ensemble d'éditeurs, nous avons facilité la tâche de construire une fédération. Cependant, nous considérons qu'il faut développer un formalisme graphique pour visualiser à la fois toutes les facettes et dimensions d'une fédération. Parfois, la vision partielle fournie par les éditeurs n'est pas suffisante.

Chapitre VIII
Conclusions et perspectives

1. Synthèse des résultats

Parmi les résultats les plus importants de notre travail, nous pouvons citer les suivants :

- Définition d'un cadre conceptuel pour définir et étudier les modèles de composants. Nous avons identifié et caractérisé trois types de composants : les composants fonctionnels (remontée du niveau d'abstraction et réification du concept de module), composants de composition (remontée du niveau d'abstraction des objets) et composants de coordination (éléments qui permettent à un groupe de composants de divers mondes de travailler ensemble). Ce cadre conceptuel nous a permis d'étudier les aspects fondamentaux et structuraux des composants, et de déterminer leurs limitations.

- Comparaison entre l'approche des composants et l'approche de la programmation orientée aspect (POA), comme deux solutions possibles aux problèmes inhérents à la programmation orientée objet, notamment en ce qui concerne la gestion des caractéristiques non-fonctionnels. Les composants se basent sur l'idée de remonter le niveau d'abstraction et considèrent une application comme un graphe de boîtes noires, dont on ne connaît individuellement que la fonctionnalité. Dans le modèle d'exécution et le modèle non-fonctionnel on peut introduire ainsi les solutions non-fonctionnelles, de telle façon qu'on évite le mélange dans les méthodes de l'application des choses fonctionnelles et des choses non-fonctionnelles. Dans la POA, par contre, l'idée est d'exprimer les solutions non-fonctionnelles en terme d'additions à la partie fonctionnelle, et de les tisser, pour obtenir de cette façon une solution complète. Nous l'appelons "intégration" ce type de composition de niveau boîte blanche. Il faut remarquer que les modèles de composants commerciaux utilisent surtout une composition de niveau boîte grise, où le composant fonctionnel est conscient de la présence du modèle d'exécution, et il est obligé de lui fournir quelques services ou d'en utiliser les fonctionnalités.

- Caractérisation du problème de coordination entre mondes. Nous avons montré que le modèle d'objets a des limitations pour exprimer les relations entre deux mondes distincts, et que le problème de coordination n'était qu'un cas particulier du problème de composition. Nous avons présenté le modèle de conteneur comme une solution rudimentaire au problème de coordination, notamment, à cause des hypothèses que le modèle fait sur les mondes à coordonner, et parce que le modèle suppose que le problème de coordination peut se réduire à une coordination "individuelle" (un composants avec plusieurs mondes), sans interférences possibles avec les autres "coordinations" (deux composants qui se coordonnent avec les mêmes mondes ne génèrent pas de conflits). Dans le cas général, les mondes peuvent avoir des

dépendances et être actifs, ce qui contredit l'hypothèse de base des modèles à conteneurs.

- Proposition de guider l'architecture des applications par un mécanisme de coordination, plutôt que par un simple mécanisme de connexion de fonctionnalités. Si on accepte que, dans une application, il y a plusieurs mondes concernés, il est imaginable d'utiliser les idées des objets pour chaque monde participant, et de construire l'architecture globale de l'application comme un groupe de modèles qui se coordonnent pour travailler ensemble. Dans ce cas, le monde de la coordination émerge et il est chargé de garantir que les mondes travaillent de manière cohérente. L'approche que nous avons suivie est d'étudier ce monde, ses éléments et ses relations, et de proposer sa structure comme l'architecture de base des applications.

- Proposition d'une fédération comme une architecture logicielle flexible qui correspond à une implémentation possible du monde de la coordination. Nous avons défini les contrats de coordination et l'univers commun, comme les structures de base de ce monde. Nous avons montré qu'une fédération peut être utilisée pour gérer aussi le problème d'intégration d'applications, cas dans lequel il faut matérialiser une partie du meta-modèle de l'application, pour l'utiliser comme base de la coordination.

- Validation des résultats par expérimentation pratique. Nous avons construit une plate-forme de support pour une fédération de composants, et nous avons implémenté une application d'évaluation sur cette plate-forme. Bien qu'il faille aller beaucoup plus loin dans les tests, nous considérons que les résultats sont très satisfaisants, et que nous avons pu démontrer la viabilité de l'approche.

2. Principales conclusions

Le travail développé nous a apportés beaucoup de leçons, dont les principales nous les résumons en trois grandes points :

- La première leçon provient du développement du travail, plutôt que des résultats obtenus. Il est frappant de constater la difficulté actuelle pour faire de la recherche dans le domaine de l'informatique. Ce qui au début n'était qu'un problème de terminologie (i.e. l'incompréhension de quelques termes, quelques discussions sur les définitions, quelques petits malentendus) est devenu avec les années un vrai cauchemar : trop de syntaxe, trop de technologie, trop de techniques ad-hoc, et très peu de vision conceptuelle. Les termes "objet", "composant", "aspect", "conteneur" ou même "architecture logicielle" sont utilisés avec des significations différentes. L'effort que nous avons réalisé pour constituer un cadre conceptuel cohérent de travail a été considérable.

- La deuxième leçon du travail est l'énorme potentiel derrière les idées de composant. Il faut d'abord abandonner l'idée qu'un composant est simplement une boîte noire, avec une fonctionnalité bien définie, et qui peut être utilisée comme unité de déploiement. Nous pensons que cette définition ne représente qu'un cas très particulier. Considérer un modèle de composants comme un triplé (modèle logique, modèle d'exécution et modèle non-fonctionnel) permet d'aller plus loin. Par contre, nous considérons que

l'approche proposée par la POA peut présenter des problèmes, bien que l'idée d'étendre la sémantique d'une application par tissage soit un mécanisme très puissant de composition. Le problème de la POA est de proposer une composition de niveau boîte blanche, qui peut engendrer des problèmes d'évolution. Nous considérons que la mécanique de tissage utilisée dans un cadre conceptuel solide et contrôlé peut constituer la base d'une plate-forme de composition. Un exemple possible est l'utilisation des techniques de tissage pour étendre le modèle d'exécution d'un modèle de composants, à partir de la description des extensions.

- La dernière leçon concerne directement le sujet de cette thèse. Nous considérons que la coordination est un mécanisme puissant de composition, qui peut apporter une architecture des applications meilleure que celle générée par la simple connexion de fonctionnalités. Nous avons proposé une coordination basée sur la structure des mondes, bien qu'il soit possible imaginer d'autres formes de coordination. En particulier, nous avons montré que les fédérations sont une alternative réelle pour la construction d'applications.

3. Perspectives et travail futur

Tout au long du document nous avons fait référence à une grande quantité de points ouverts, qui méritent une étude approfondie. Nous considérons les axes suivants de travail comme les plus importantes :

- Exploration des aspects méthodologiques et des formalismes graphiques pour exprimer une fédération. Bien que la coordination semble une approche viable pour guider l'architecture des applications, il est vrai que le niveau de complexité augmente, et qu'il faut pouvoir s'appuyer sur des formalismes pour exprimer et valider une conception, de même que sur des guides méthodologiques, qui expliquent, par étapes, la construction d'une telle architecture.

- Evaluation des possibilités des fédérations sur deux axes : (1) comme architecture de base pour l'intégration d'applications (i.e. coordination par matérialisation du meta-modèle des applications), et (2) comme architecture pour des applications graphiques interactives, avec la vision MVC+C proposée dans cette thèse, et comparer les résultats avec ceux obtenus avec une architecture MVC, où la coordination est assumée entièrement par les mondes C et V.

- Approfondir l'étude des modèles de composants. En particulier il est important d'étudier les modèles de nouvelle génération comme Fractal [CBS02, BCS02] et Avalon. Il est aussi important d'essayer de remonter le niveau d'abstraction du modèle d'exécution des modèles de composants, de telle façon qu'il soit possible de construire des modèles de composants sur mesure pour des domaines non-fonctionnels concrets.

Chapitre IX
Bibliographie

[AAP99] J. M. Andreoli, D. Arregui, F. Pacull, M. Riviere, J. Y. Vion-Dury, J. Willamowski "CLF/Mekano: A Framework for Building Virtual-Enterprise Applications". Proceedings of the 3rd International Enterprise Distributed Object Computing Conference (EDOC'99), Germany, September 1999.

[ABS94] M. Aksit, J. Bosch, W. van der Sterren, L. Bergmans. "Real-Time Specification Inheritance Anomalies and Real-Time Filters". In Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP'94), Lecture Notes in Computer Science, Vol. 821, Springer-Verlag, Italie, juillet 1994.

[ACN02] J. Aldrich, C. Chambers, D. Notkin. "ArchJava: Connecting Software Architecture to Implementation". Proceedings of the 24th International Conference on Software Engineering, Orlando FL, USA, mai 2002. IEEE Computer Society Press.

[Actoll] Centr'Actoll Project. Information available at: http://www-adele.imag.fr/Les.Groupes/centractoll/index.html

[AF02] L. Andrade, J. L. Fiadeiro. "An Architectural Approach to Auto-Adaptive Systems". 22nd International Conference on Distributed Computing Systems Workshops (ICDCSW '02), Austria, July 2002.

[AF99] L. F. Andrade, J. L. Fiadeiro. "Interconnecting Objects via Contracts". UML'99 –The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings, volume 1723 of LNCS, pages 566–583. Springer, 1999.

[AFM98] J. M. Andreoli, C. Fernstrom, J. L. Meunier. "A Coordination System Approach to Software Workflow Process Evolution". Proceedings of the 13th IEEE International Conference on Automated Software Engineering (ASE'98), Honolulu, U.S.A., October, 1998

[AK02a] D. H. Akehurst, S. Kent. "A Relational Approach to Defining Transformations in a Metamodel". In Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, UML 2002 - The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany, September/October 2002, Proceedings, volume 2460 of LNCS, pages 243–258. Springer, 2002.

[AK02b] C. Atkinson, T. Kühne. "The Role of Meta-modeling in MDA". In Jean Bezivin and Robert France, editors, Workshop in Software Model Engineering, Germany, October 2002.

[Ami99] M. Amiour. "Vers une Fédération de Composants Interopérables pour les Environnements Centrés Procédés Logiciels". Thèse de doctorat de l'Université Joseph Fourier, Grenoble, France, juin 1999.

[ANT] The Apache Software Foundation. "The Apache Ant Project". Disponible à : http://ant.apache.org/

[APR00] D. Arregui, F. Pacull, M. Riviere. "Heterogeneous Component Coordination: the CLF Approach". Proceedings of the Fourth International Enterprise Distributed Object Computing Conference (EDOC'00), Japan, September 2000.

[Arc01] T. Archer. "Inside C#". Microsoft Press. 2001.

[AT98] M. Aksit, B. Tekinerdogan. "Aspect-Oriented Programming Using Composition Filters", in Object-Oriented Technology, Workshop Reader, European Conference on Object-Oriented Programming (ECOOP'98), Springer-Verlag, Juillet 1998.

[AWB93] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, A. Yonezawa. "Abstracting Object Interactions Using Composition-Filters". Object-Based Distributed Processing, Springer-Verlag, pp. 152-184, 1993.

[BA01] L. Bergmans, M. Aksit. "Composing Crosscutting Concerns Using Composition Filters". Communications of the ACM, Vol. 44, No. 10, pp. 51-57, October 2001.

[BA99] B. Boehm, C. Abts. "COTS Integration: Plug and Play?". IEEE Computer, pp. 135-138, January 1999.

[BBB00] F. Bachman, L. Bass, C. Buhman, S. Comella-Dorda. "Volume II: Technical Concepts of Component-Based Software Engineering". Technical Report CMU/SEI-2000-TR-008, Carnegie Mellon University, Mai 2000.

[BC01] G. Bieber, J. Carpenter, "Introduction to Service-Oriented Programming", September 2001, Online whitepaper at http://www.openwings.org/download.html

[BC90] G. Bracha, W. Cook. "Mixin-Based Inheritance". Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'90), Canada, octobre 1990.

[BCK98] L. Bass, P. Clements, R. Kazman. "Software Architecture in Practice", Addison Wesley, 1998.

[BCS02] E. Bruneton, T. Coupaye, J. B. Stefani. "Recursive and Dynamic Software Composition with Sharing". 7th International Workshop on Component-Oriented Programming (WCOP '02), European Conference on Object Oriented Programming (ECOOP '02), Malaga, Spain, June 2002.

[BEAN97] Sun Microsystems. "Java Beans 1.01 Specification". 1997. Disponible à : http://java.sun.com/beans

[BEAN98] Sun Microsystems. "Extensible Runtime Containment and Server Protocol for JavaBeans Version 1.0", 1998. Disponible à : http://java.sun.com/products/javabeans/glasgow/beancontext.pdf

[BeanBox] Sun Microsystems. "The Beanbox". Disponible à : http://java.sun.com/products/javabeans/software/beanbox.html

[Bez01] J. Bézivin "From Object Composition to Model Transformation with the MDA". TOOLS USA'2001, Volume IEEE TOOLS-39, Santa Barbara, USA, August 2001. Disponible à: http://www.sciences.univ-nantes.fr/info/lrsg/Recherche/mda

[Bez01] J. Bézivin, "From Object Composition to Model Transformation with the MDA". Proceedings of TOOLS USA, Volume IEEE TOOLS-39, Santa Barbara, USA, August 2001

[BG02] J. P. Barros, L. Gomes. "Activities as Behavior Aspects". The 2nd International Workshop on Aspect-Oriented Modeling with UML, Germany October 2002.

[BH02] J. Baker, W. Hsieh. "Runtime Aspect Weaving Through Metaprogramming". In Proceedings of the 1st Aspect-Oriented Software Development, ACM Press, Pays-Bas, Avril 2002.

[BJP99] A. Beugnard, J.M. Jezequel, N. Plouzeau, D. Watkins. "Making Components Contract Aware". IEEE Software, juin 1999.

[BMR96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. "Pattern-Oriented Software Architecture A System of Patterns", volume 1. Wiley, 1996.

[Boo94] G. Booch. "Object-Oriented Analysis and Design with Applications". Object Technology Series. Addison-Wesley, second edition, 1994.

[Bos00] J. Bosch. "Design and Use of Software Architectures. Adopting and Evolving a Product Line Approach", Pearson Education (Addison-Wesley & ACM Press), Mai 2000.

[Box98] D. Box. "Essential COM". Addison-Wesley, 1998.

[BP02] M. Belaunde, M. Peltier. "From EDOC Components to CCM Components: a Precise Mapping Specification". Fundamental Approaches to Software Engineering, Proceedings of the 5th International Conference FASE 2002, Grenoble, France, April 2002.

[Bro01] M. Broy. “Specification and Modeling: An Academic Perspective”. Proceedings of the 23th International Conference of Software Engineering, ICSE ’01, Pages: 673 – 675, Toronto, Canada, 2001.

[Bro96] A.W. Brown. (ed.), "Component-Based Software Engineering". IEEE Computer Society Press, Los Alamitos, CA, 1996.

[BS02] M. Barnett, W. Schulte. "Contracts, Components, and their Runtime Verification on the .NET Platform". Microsoft Research, Technical Report MSR-TR-2002-38, April 2002.

[BW98] A. W. Brown, K. Wallnau. "The Current State of CBSE". IEEE Software, pp. 37-46. September/October 1998.

[CAS] ExoLab Group. "The Castor Project". Disponible à : http://castor.exolab.org/

[CBS02] T. Coupaye, E. Bruneton, J.B. Stefani. "The Fractal Composition Framework". Rapport Technique, The ObjectWeb Group, juillet 2002.

[CCD00a] C. Cugola, P.Y. Cunin, S. Dami, J. Estublier, A. Fuggetta, F. Pacull, M. Rivière, H. Verjus. "Support for Software Federations: the PIE Platform". Proceedings of the 7th European Workshop on Software Process Technology, EWSPT’2000, Austria, février 2000.

[CCD00b] C. Cugola, P.Y. Cunin, S. Dami, J. Estublier, A. Fuggetta, F. Pacull, M. Rivière, H. Verjus. "Customizing the Behavior of Middleware: the PIE Approach". Proceedings of the Workshop on Reflective Middleware (RM2000). New York, USA, avril 2000.

[CCM] OMG, "CORBA Components – Version 3.0". Juin 2002. Disponible à : http://www.omg.org/technology/documents/formal/components.htm

[CEF01] M. Chaudron, E. Eskenazi, A. Fioukov, D. Hammer. “A Framework for Formal Component-Based Software Architecture ”. Workshop Specification and Verification of Component- Based System at OOPSLA 2001, Tampa Bay, Florida, USA October 2001.

[Cer02] H. Cervantes. "Beanome: A Component Model for the OSGi Framework", Workshop Software Infrastructures for Component-Based Applications on Consumer Devices,

Lausanne, Suisse, septembre 2002.

[CES97] K. Czarnecki, U.W. Eisenecker, P. Steyaert, "Beyond Objects: Generative Programming". 11th European Conference on Object-Oriented Programming (ECOOP '97), Workshop on Aspect-Oriented Programming. 1997.

[CFD02] H. Cervantes, J. M. Favre, F. Duclos. "Describing Hierarchical Compositions of Java Beans with the Beanome Language". Software Composition Workshop of the 5th European Joint Conference on Theory and Practice of Software (ETAPS'02), Grenoble France, 2002.

[CHJ02] I. Crnkovic, B. Hnich, T. Jonsson, Z. Kiziltan. "Specification, Implementation, and Deployment of Components". Communications of the ACM, Vol. 45, No. 10, octobre 2002.

[CKK00] P. Clements, R. Kazman, M. Klein. "Evaluating Software Architectures: Methods and Case Studies", Addison Wesley, 2000.

[CKO92] B. Curtis, M. I. Kellner, J. Over. "Process Modeling", Communications of the ACM, Volume 35, No. 9, September 1992.

[COM] Microsoft. "COM Specification".
Disponible à: http://www.microsoft.com/com/resources/comdocs.asp

[CS02] P. J. Clemente, F. Sánchez, M. A. Pérez. "Modeling with UML Component-Based and Aspect Oriented Programming Systems". 7th International Workshop on Component-Oriented Programming (WCOP'02), European Conference on Object Oriented Programming (ECOOP '02), Malaga, Spain, June 2002.

[CW01] S. Clarke, R. Walker. "Composition Patterns: An Approach to Designing Reusable Aspects". Proceedings of the 23th International Conference of Software Engineering, ICSE '01, Pages: 5 – 14, Toronto, Canada, 2001

[CW02] T. Clark, J. Warmer, editors. "Object Modeling with the OCL: The Rationale Behind the Object Constraint Language". Lecture Notes in Computer Science (LNCS), Vol. 2263, Springer, 2002.

[DDM00] T. D'Hondt, K. De Volder, K. Mens, R. Wuyts. "Co-evolution of Object-Oriented Software Design and Implementation". Proceedings of the International Symposium on Software Architectures and Component Technology 2000, 2000.

[DEA98] S. Dami, J. Estublier, M. Amiour. "APEL : A Graphical Yet Executable Formalism for Process Modeling". ASE Journal, vol. 5, Kluwer Academic Publishers, janvier 1998.

[DEM01] F. Duclos, J. Estublier, P. Morat. "Environnement pour la Gestion d'Aspects Non-fonctionnels dans un Modèle à Composants". ICSSEA, Paris, décembre 2001.

[DEM02] F. Duclos, J. Estublier, P. Morat. "Describing and Using non Functional Aspects in Component Based Applications". Proceedings of the 1st Aspect-Oriented Software Development (AOSD '02), ACM Press, Enshede, Pays-Bas, avril 2002.

[DES02] F. Duclos, J. Estublier R. Sanlaville. "Une Machine à Objets Extensibles pour la Séparation des Préoccupations". Journées Systèmes à Composants Adaptables et Extensibles, octobre 2002, Grenoble, France

[DFF00] D. Deveaux, R. Fleurquin, P. Frison, J.-M. Jézéquel, Y. Le Traon. "Composants Objets Fiables : Une Approche Pragmatique". L'Objet, Vol. 5, No. 3-4, pp. 469-494, April 2000.

[DFS02] R. Douence, P. Fradet, M. Südholt. "Detection and Resolution of Aspect Interactions".

Rapport de Recherche INRIA No. 4435, avril 2002.

[Dil94] A. Diller. "Z: An Introduction to Formal Methods, 2nd Edition". John Wiley & Son, June 1994.

[DPC] Sun MicroSystems. "Dynamic Proxy Classes". Disponible à: http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html

[DR02] B.Demsky, M. Rinard. "Role-Based Exploration of Object-Oriented Programs". Proceedings of the 24th International Conference of Software Engineering, ICSE '02, Pages: 313 – 324, Orlando, Florida, USA, 2002.

[DSo01] D. D'Souza. "Model-Driven Architecture and Integration: Opportunities and Challenges" Version 1.1, Document available at www.kinetiuym.com, February 2001.

[DT03] A. Dogru, M. Tanik. "A Process Model for Component-Oriented Software Engineering". IEEE Software, Vol. 20, No. 2, pp. 34-41, April 2003.

[Duc02] F. Duclos. "Environnement de Gestion de Services Non Fonctionnels dans les Applications à Composants". Thèse en informatique à l'Université Joseph Fourier (Grenoble), octobre 2002.

[DW98] D. D'Souza, A. Wills. "Objects, Components, and Frameworks with UML - The Catalysis Approach". Addison-Wesley Object Technology Series. Addison-Wesley Publishing Company, 1998.

[EAB02] T. Elrad, O. Aldawud, A. Bader. "Aspect-Oriented Modeling: Bridging the Gap between Implementation and Design". Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA, October, 2002.

[EAK0l] I. Elrad, M. Aksit, G. Kiczales, K. Lieberherr, H. Ossher. "A Discussion on Aspect-Oriented Programming: Frequently-Asked Questions". Communications of the ACM, Vol. 44, No. 10, pp. 33-38, octobre 2001.

[EC00] U. Eisenecker, K. Carnecki. "Generative Programming: Methods, Tools, and Applications", Addison-Wesley, 2000.

[ECB98] J. Estublier, P.Y. Cunin, N. Belkhatir. "Architectures for Process Support System Interoperability". Proceedings of the 5th International Conference on Software Process ICSP 5. Chicago, USA, juin 1998.

[Edw01] W. Edwards. "Core Jini – Second Edition". Prentice Hall, 2001.

[EFS02] J. Estublier, J. Favre, R. Sanlaville. "An Industrial Experience with Dassault Systemes' Component Model". Dans "Building Reliable Component-Based Systems", Archtech House, 2002.

[ELV03] J. Estublier, T. Le-Anh, J. Villalobos. "Using Federations for Flexible SCM Systems". Proceedings of the 11th International Workshop on Software Configuration Management (SCM-11), ICSE'2003, LNCS Series, Springer-Verlag, USA, May 2003.

[Eng02] V. Englebert. "The Synchronization of Independent and Specific Models". In Jean Bezivin and Robert France, editors, Workshop in Software Model Engineering, Germany, October 2002.

[Eng97] R. Englander. "JavaBeans Guide du Programmeur". O'Reilly, première édition, 1997.

[EVC01a] J. Estublier, H. Verjus, P.Y. Cunin. "Designing and Building Software Federations". 1st Conference on Component Based Software Engineering. (CBSE), Varsovie, Poland, septembre 2001.

[EVC01b] J. Estublier, H. Verjus, P.Y. Cunin. "Building Software Federation". Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'2001, Las Vegas, USA, juin 2001.

[EVC01c] J. Estublier, H. Verjus, P.Y. Cunin. "Modeling and Managing Software Federations". European Conference on Software Engineering (ESEC), Wien, Austria. septembre 2001.

[FBF99] T. Fraser, L. Badger, M. Feldman. "Hardening COTS with Generic Software Wrappers". Proceedings of the 1999 IEEE Symposium on Security and Privacy, pp. 2-16, May 1999.

[FG03] A. Farias, Y. Guéhéneuc. "On the Coherence of Component Protocols". Workshop on Software Composition (SC'2003), ENTCS Vol. 82, No. 5, Elsevier, Warsaw, Poland, April 2003.

[FHS92] R. Freund, B. Haberstroh, C. Stary. "Applying Graph Grammars for Task-Oriented User Interface Development". In W. Koczkodaj, editor, Proceedings IEEE Conference on Computing and Information ICCI'92, pages 389-392, 1992.

[Fin00] A. Finkelstein, editor. "The Future of Software Engineering". 22nd International Conference on Software Engineering (ICSE'2000), IEEE Computer Society Press / ACM Press, juin 2000.

[Fin98] L. Finch. "So Much OO, So Little Reuse". Dr. Dobb's Journal, disponible à: http://www.ddj.com/articles/1998/9875/, mai 1998.

[Fow99] M. Fowler. "Refactoring: Improving the Design of Existing". Addison Wesley Publisher, juin 1999.

[FS02] A. Farias, M. Südholt. "On the Construction of Components with Explicit Protocols". Technical Report 02/4/INFO. Ecole de Mines de Nantes, Nantes, France. 2nd edition, juin 2002.

[FWK02] P. Fremantle, S. Weerawarana, R. Khalaf. "Enterprise Services". Communications of the ACM, Vol. 45, No. 10, octobre 2002.

[Gar02] S. Garcia. "Ingénierie Concurrente dans le Domaine de Gestion de Documents". Université Joseph Fourier de Grenoble, Rapport de DEA (Systèmes et Communications), septembre 2002.

[Gen97] W. M. Gentleman. "Effective Use of COTS (Commercial-Off-the-Shelf) Software Components in long Lived Systems". Proceedings of International Conference on Software Engineering (ICSE'97), Boston, USA, May 1997.

[GHJ95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. "Design Patterns : Elements of Reusable Object- Oriented Software". Addison-Wesley, 1995.

[GJS00] J. Gosling, B. Joy, G. Steele, G. Bracha. "The Java Language Specification". Addison-Wesley, second edition, 2000.

[GMH99] J. Grundy, W. Mugridge, J. Hosking. "Constructing Component-Based Software Engineering Environments: Issues and Practices". Journal of Information and Software Technology, Special Issue on Constructing Software Engineering Tools, 1999.

[GMW00] D. Garlan, R. Monroe, D. Wile. "ACME: Architectural Description of Component-Based Systems", Foundations of Component-Based Systems, Cambridge University Press, 2000.

[Gon99] L. Gong. "Inside Java 2 Platform Security: Architecture, API Design, and

Implementation", Addison Wesley, 1999.

[GR85] A. Goldberg, D. Robson. "Smalltalk-80: The Language and its Implementation". Addison-Wesley, 1985.

[Gro01] W, Grosso. "Java RMI", O'Reilly, 2001.

[GS00] D. Garlan, J. Sousa. "Documenting Software Architectures: Recommendations for Industrial Practice". Technical Report CMU-CS-00- 169, Carnegie Mellon University, Pittsburgh, Pennsylvania, octobre 2000.

[GS96] M. Goedicke, B. Sucrow. "Towards a Formal Specification Method for Graphical User Interfaces Using Modularized Graph Grammars". Proceedings of the Eighth International Workshop on Software Specification and Design, pp. 56-65, IEEE Computer Society Press, Germany, March 1996.

[Har87] D. Harel. "Statecharts : A Visual Formulation for Complex Systems". Science of Computer Programming, Vol. 8, No. 3, pp. 231-274, June 1987.

[HC01] G.T. Heinman, W.T. Council. "Component-based Software Engineering: Putting the Pieces Together". Addison-Wesley, 2001.

[HJ] IBM. "HyperJ" Disponible à: http://www.alphaworks.ibm.com/tech/hyperj

[HL95] W. Hürsch, C. Lopes. “Separation of Concerns”. Technical Report. College of Computer Science, Northeastern University, Boston, February 1995

[HO93] W. Harrison, H. Ossher. “Subject-Oriented Programming (A Critique of Pure Objects)”. Proceedings of the OOPSLA ‘93 Conference on Object-Oriented Programming Systems, Languages and Applications, pp. 411-428, ACM 1993.

[Hoa96] C. Hoare. “The Role of Formal Techniques: Past, Current and Future or How Did Software Get so Reliable Without Proof?”. Proceedings of the 18th international conference of Software engineering, ICSE ’96, Pages: 233 – 234, Berlin, Germany, 1996

[IBM] IBM Research. "Multi-Dimensional Separation of Concerns: Software Engineering using Hyperspaces". Disponible à : http://www.research.ibm.com/hyperspace/

[IT02] P. Inverardi, M. Tivoli. “The Role of Architecture in Components Assembly”. 7th International Workshop on Component-Oriented Programming (WCOP ’02), European Conference on Object Oriented Programming (ECOOP ’02), Malaga, Spain, June 2002.

[JBR99] I. Jacobson, G. Booch, J. Rumbaugh. "The Unified Software Development Process". Object Technology series. Addison-Wesley, Reading, Massachusetts, USA, 1999.

[JCC] WebGain. "Java Compiler Compiler (JavaCC) – The Java Parser Generator". Disponible à : http://www.webgain.com/products/java_cc/

[JCJ93] I. Jacobson, M. Christerson, M. Jonsson, P. van Overgaard. "Object-Oriented Software Engineering, A Use Case Driven Approach". Addison-Wesley, Reading, Massachusets, 1993.

[JNI] Sun Microsystems. "Java Native Interface (JNI)". Disponible à : http://java.sun.com/products/jdk/1.2/docs/guide/jni/

[JOnAS] ObjectWeb Open Source Middleware. "JOnAS: Java Open Application Server". Disponible à : http://www.objectweb.org/jonas/

[JS02] R. Jardim-Gonçalves, A. Steiger-Garçao, "Implicit Multilevel Modeling in Flexible Business Environments". Communications of the ACM, Vol. 45, No. 10, octobre 2002.

[KAF02] G.Koutsoukos, L.Andrade, J.L.Fiadeiro, J.Gouveia. “Architectural Concerns and Use of a Model-Driven Development Framework”. OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture, Seattle, USA, 2002

[KHH01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold. “An Overview of AspectJ”. In Proceedings of the 5th European Conference on Object-Oriented Programming, Springer Verlag, 2001.

[KHL97] M. Katchabaw, S. Howard, H. Lutfiyya, A. Marshall, M. Bauer, “Making Distributed Applications Manageable Through Instrumentation”. Proceeding of Second International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE ‘97), pp. 84-94, Boston, Massachusetts, May 1997.

[KK03] M. Katara, S. Katz. "Architectural Views of Aspects". Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03), pp. 1-10, Boston, USA, March 2003.

[KLM97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, J. Irwin. “Aspect-Oriented Programming”. Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP ‘97). Lecture Notes in Computer Science vol.1241, Springer-Verlag, Juin 1997.

[Klo02] C. Kloukinas, “Composition of Software Architecture”. Thèse de Doctorat en Informatique, Université de Rennes I, 2002.

[KM97] J. Kramer, J. Magee, "Exposing the Skeleton in the Coordination Closet", Proceedings of the Second International Conference on Coordination Models and Languages, LNCS, Number 1282, pp. 18-31, September, 1997.

[KRB92] G. Kiczales, J. des Rivieres, D. Bobrow. “The Art of the Metaobject Protocol”. The MIT Press, Cambridge, Massachusetts, 1992.

[Kru95] P. B. Kruchten. "The 4+1 View Model of Software Architecture". *IEEE Software*, 12(6): 42-50, novembre 1995.

[LA02] K. Levi, A. Arsanjani. "A Goal-driven Approach to Enterprise Component Identification and Specification". Communications of the ACM, Vol. 45, No. 10, octobre 2002.

[Lar01] C. Larman. "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process". Prentice Hall, 2nd Edition, 2001.

[LBM01] A. Lédeczi, A. Bakay, M. Maroti, P. Völgyesi, G. Nordstrom. "Composing Domain-Specific Design Environments". IEEE Computer, Vol. 34, No. 11, pp. 44-51, November 2001.

[Les02] V. Lestideau. "Un Environnement de Déploiement Automatique pour les Applications à Base de Composants". International Conference "Software & Systems Engineering and their Applications" ICSSEA'02, France, December 2002.

[LL94] C. Lopes, K. Lieberherr, “Generative Patterns”. 8th European Conference on Object-Oriented Programming (ECOOP’94), Workshop on Patterns, Italie, juillet 1994.

[LLM99] K. Lieberherr, D. Lorenz, M. Mezini. "Programming with Aspectual Components". Rapport technique NU-CCS-99-01, College of Computer Science Northeastern University, Boston MA, mars 1999.

[LOO01] K. Lieberherr, D. Orleans, J. Ovlinger. “Aspect-Oriented Programming with Adaptive

Methods". Communications of the ACM, Vol. 44, No. 10, pp. 39-4 1, octobre 2001.

[LR97] F. Leymann, D. Roller, "Workflow-Based Applications". IBM Systems Journal, Vol. 36, No. 1, 1997.

[LS00] R. Lee, S. Seligman. "JNDI API Tutorial and Reference: Building Directory-Enabled Java Applications", Addison Wesley, 2000.

[LS00] G. T. Leavens, M. Sitaraman (editors). "Foundations of Component-Based Systems" Cambridge University Press, 2000.

[LSH03] J. Lee, K. Siau , S. Hong. "Enterprise Integration with ERP and EAI". Communications of the ACM, Volume 46. Issue 2, February 2003.

[LSX94] K. Lieberherr, I. Silva-Lepe, C. Xiao. "Adaptive Object-Oriented Programming Using Graph-Based Customization". Communications of the ACM, Vol. 37, No. 5, pp. 94-101, mai 1994.

[Lut00] J. C. Lutz. "EAI Architecture Pattern". EAI Journal, pp. 64-73, March 2000.

[LV01] D. Lorenz, J. Vlissides. "Designing Components versus Objects: A Transformational Approach". Proceedings of the 23th International Conference of Software Engineering, ICSE '01, Pages: 253 – 262, Toronto, Canada, 2001

[LVE03] T. Le-Anh, J. Villalobos, J. Estublier. "Multi-Level Composition for Software Federations". ETAPS-2003, Workshop on Software Composition (SC'2003), ENTCS Vol. 82, No. 5, Elsevier, Warsaw, Poland, avril 2003.

[Mey00] B. Meyer. "What to Compose". Software Development Magazine, mars 2000. Disponible à : http://www.sdmagazine.com

[Mey01] B. Meyer. "The Power of a Unifying View". Software Development Magazine, juin 2001.

[Mey96] B. Meyer. "The Reusability Challenge". IEEE Computer, Component and Object Technology, février 1996.

[Mey97] B. Meyer. "Object-Oriented Software Construction". Prentice-Hall, second edition, 1997.

[Mey99a] B. Meyer. "The Significance of Components". Software Development Magazine, novembre 1999. Disponible à : http://www.sdmagazine.com

[Mey99b] B. Meyer. "Rules for Component Builders", Software Development Magazine, mai 1999.

[Mey99c] B. Meyer, "A Really Good Idea". IEEE Computer IEEE, Component and Object Technology, janvier 1999.

[MM01] R. Marvie, P. Merle, "CORBA Component Model: Discussion and Use with Open CCM". Submitted to a Special Issue of the Informatica - An International Journal of Computing and Informatics dedicated to "Component Based Software Development", 2001

[MMC95] P. Mulet, J. Malenfant, P. Cointe. "Towards a Methodology for Explicit Composition of MetaObjects". Proceedings of 10th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95), pp. 316-330, octobre 1995.

[Mon01] R. Monson-Haefel. "Enterprise JavaBeans". O'Reilly & Associates, 3rd edition, octobre 2001.

[MOT97] N. Medvidovic, P. Oreizy, R. N. Taylor. "Reuse of Off-the-Shelf Components in C2-Style Architectures". Proceeding of the Symposium on Software Reusability (SSR'97), Boston, USA, May 1997.

[MS01] E. Makinen, T. Systa. “MAS - An Interactive Synthesizer to Support Behavioral Modeling in UML”. Proceedings of the 23th International Conference of Software Engineering, ICSE ’01, Pages: 15 – 24, Toronto, Canada, 2001

[MSZ01] S.A. McIlraith, T.C. Son, H. Zeng. "Semantic Web Services". IEEE Intelligent Systems (Special Issue on the Semantic Web), Vol. 16, No. 2, pp. 46-53, March/April 2001.

[MT00] N. Medvidovic, R. Taylor. "A Classification and Comparison Framework for Software Architecture Description Languages". IEEE Transaction on Software Engineering, Vol. 26, No. 1, Janvier 2000.

[MT02] S. Matougui, B. Traverson. "DRACON : une démarche outillée de réutilisation et d'assemblage de composants reposant sur le concept de contrat". Génie Logiciel, Intégration & Interopérabilité, No. 62, pp. 24-31, septembre 2002.

[MTK97] J. Magee, A. Tseng, J. Kramer. “Composing Distributed Objects in CORBA”. Proceedings of 3rd International Symposium of Autonomous Distributed Systems (ISADS’97), Berlin, Germany, avril 97.

[MTS] Microsoft Corporation, "Microsoft Transaction Server (MTS)", Microsoft Component Services. Server Operating System: A Technology Overview, Disponible à: http://www.microsoft.com/com/wpaper/compsvcs.asp

[ND95] O. Nierstrasz, L. Dami. “Component-Oriented Software Technology”, in Object-Oriented Software Composition, Eds. Nierstrasz 0. and Tsichritzis D., Prentice Hall, Englewood Cliffs NJ USA, 1995

[NetBeans] NetBeans: An Open Source Modular IDE. Disponible à : http://www.netbeans.org

[NR69] P. Naur, B. Randell, (editors). "Software Engineering: Report", NATO Scientific Affairs Division, janvier 1969.

[NR98] R. Natarajan, D. Rosenblum. "Merging Component Models and Architectural Styles". Proceedings of the 3rd International Software Architecture Workshop, Florida, Novembre 1998.

[OBS] Excelon Corporation – Progress Software. "ObjectStore". Disponible à : http://www.exln.com/products/objectstore/

[OH98] R. Orfali, D. Harkey. "Client/Server Programming with Java and CORBA, 2nd Edition", John Wiley & Sons, March 1998.

[OL01] D. Orleans, K. Lieberherr. “DJ: Dynamic Adaptive Programming in Java”. Proceedings of Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns, Springer Verlag, septembre 2001.

[OMG95] Object Management Group, “CORBA Services: Common Object Services Specification”, Object Management Group, Revision Edition March 31, 1995

[Omm02] R. Ommering. “Building Product Populations with Software Components”. Proceedings of the 24th International Conference of Software Engineering, ICSE ’02, Pages: 255 – 265, Orlando, Florida, USA, 2002.

[OMT98] P. Oreizy, N. Medvidovic, R. Taylor, D. Rosenblum. "Software Architecture and Component Technologies: Bridging the Gap", Workshop on Compositional Software Architectures, Monterey, Janvier 1998.

[ORBacus] IONA Technologies. "ORBacus for C++ et Java". 2001. Disponible à : http://www2.iona.com/support/docs/orbacus/4.0.5/OB-4.0.5.pdf

[OSGi] "OSGi Service Gateway Specification", Release 1.0, Mai 2000. Disponible à : http://www.OSGI.org

[OT00] H. Ossher, P. Tarr. “Multi-Dimensional Separation of Concerns and The Hyperspace Approach”. Proceedings of the Symposium on Software Architectures and Component Technology. The State of the Art in Software Development. Kluwer, 2000.

[PA98] G. A. Papadopoulos, F. Arbab. "Coordination Models and Languages". Centrum voor Wiskunde en Informatica, Report SEN-R9834, December 1998.

[Par72] D. L. Parnas. “On the Criteria for Decomposing Systems into Modules”. Communication of the ACM, 15(12) pp: 1053-1058, décembre 1972.

[Per00] E., Persson. "Business Object Components", ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'2000), Minneapolis, USA, 2000.

[PKG99] J. Payton, R. Keshav, R. F. Gamble. "System Development Using the Integrating Component Architecture Process". Proceedings of the First Workshop on Ensuring Successful COTS Development, Los Angeles, USA, 1999.

[PM01] E. Pitt, K. McNiff. "Java RMI: The Remote Method Invocation Guide", Addison Wesley, 2001.

[Pol01] J. T. Pollock, “The Big Issue: Interoperability vs. Integration”, EAI Journal, pp. 48-52, October 2001

[Poo01] J. Poole. “Model-Driven Architecture: Vision, Standards and Emerging Technologies”. ECOOP 2001, Workshop on Meta-modeling and Adaptive Object Models, April 2001.

[PV02] F. Plasil, S. Visnovsky. "Behavior Protocols for Software Components". IEEE Transactions on Software Engineering, Vol. 28, No. 11, novembre 2002.

[Ran96] B. Randell. "The 1968/69 NATO Software Engineering Reports", History of Software Engineering, Dagstuhl - Seminar 9635, août 1996, disponible à: http://www.cs.ncl.ac.uk/old/people/brian.randell/home.formal/NATO/NATOReports/

[Ree00] G. Reese. "Database Programming with JDBC and Java", second edition, O'Reilly, 2000.

[Rit99] A. Rito Silva. “Separation and Composition of Overlapping and Interacting Concerns”. 14th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA ‘99), First Workshop on Multi-Dimensional Separation of Concerns in Object-Oriented Systems, USA, novembre 99.

[RJB99] J. Rumbaugh, I. Jacobson, G. Booch. "The Unified Modeling Language Reference Manual". Object Technology Series. Addison Wesley, 1999.

[RK01] D. Rayside, K. Kontogiannis. “On the Syllogistic of Object-Oriented Programming”. Proceedings of the 23th International Conference of Software Engineering, ICSE ’01, Pages: 113 – 122, Toronto, Canada, 2001

[RM97] W. Ruh, T. Mowbray, "Inside CORBA: Distributed Object Standards and Applications", Addison Wesley, 1997.

[RMI] Sun Microsystems. "Java Remote Method Invocation (RMI)". Disponible à : http://java.sun.com/products/jdk/rmil

[Rom99] E. Roman. "Mastering Enterprise JavaBeans and the Java2 Platform, Enterprise Edition", Wiley Computer Publishing, 1999.

[RS77] D. Ross, K. Schoman. "Structured Analysis for Requirements Definition". IEEE Transactions on Software Engineering, Special Collection on Requirement Analysis, Vol. 3, No. 1, pp. 6-15, janvier 1977.

[San02] R. Sanlaville. “Architecture Logicielle: une Expérimentation Industrielle avec Dassault Systèmes”. Thèse de Doctorat en Informatique, Université de Grenoble, mai 2002.

[SEI] Software Engineering Institute. "How Do You Define Software Architecture?". Disponible à: http://www.sei.cmu.edu/architecture/definitions.html.

[Sel01] B. Selic. “Specification and Modeling: An Industrial Perspective”. Proceedings of the 23th International Conference of Software Engineering, ICSE ’01, Pages: 676 – 677, Toronto, Canada, 2001.

[SF02] H. Smith, P. Fingar "Business Process Management: The Third Wave". Meghan-Kiffer Press, ISBN: 0929652339, December 2002.

[SG01] B. Spitznagel, D. Garlan. “A Compositional Approach for Constructing Connectors“. The Working IEEE/IFIP Conference on Software Architecture (WICSA ‘01), pp 148-157, IEEE Computer Society, The Netherlands, Août 2001.

[SG96] M. Shaw, D. Garlan. "Software Architecture. Perspectives on an Emerging Discipline". Prentice-Hall, 1996.

[Sie96] J. Siegel. ""CORBA Fundamentals and Programming", John Wiley & Sons, 1996.

[SLH01] M. Sitaraman, T.Long, E. Harner. “A Formal Approach to Component-Based Software Engineering: Education and Evaluation”. Proceedings of the 23th International Conference of Software Engineering, ICSE ’01, Pages: 601 – 608, Toronto, Canada, 2001

[SLM96] P. Steyaert, C. Lucas, K. Mens, T. D'Hondt. "Reuse Contracts: Managing the Evolution of Reusable Assets". OOPSLA '96 Conference on Object-Oriented Programming Systems, Languages and Applications. ACM SIGPLAN Notices, Vol. 31, No. 10, octobre 1996.

[Smi84] B. C. Smith. “Reflection and Semantics in Lisp”. Proceedings of the 11th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pages 23-35, ACM Press, 1984.

[Sno89] Snodgrass R., “The Interface Description Language : Definition and Use”. Computer Science Press, New York, USA, USA, 1989.

[SOAP] The Apache Software Foundation. "The Apache XML Project". Disponible à : http://ws.apache.org/soap/

[Sre02] V. Sreedhar. “Mixin’Up Components”. Proceedings of the 24th International Conference of Software Engineering, ICSE’02, Pages: 198 – 206, Orlando, Florida, USA, 2002.

[Sri00] S. Shrivastava, L. Bellissard, F. Herrmann M., De Palma N., Wheater S., “A Workflow and Agent based Platform for Service Provisioning”, C3DS ESPRIT Long Term Research Project no 24962, http://www.newcastle.research.ec.org/c3ds/index.html, 2000

[SSJ02] I. Singh, B. Stearns, M. Johnson. "Designing Enterprise Applications with the J2EE Platform". Addison-Wesley, 2nd edition, June 2002.

[SSR00] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann. "Pattern-Oriented Software Architecture Patterns for Concurrent and Networked Objects", volume 2, Wiley & Sons, 2000.

[Sta02] M. Stal. "Web Services: Beyond Component-Based Computing". Communications of the ACM, Vol. 45, No. 10, octobre 2002.

[Ste90] G. Steele. "Common Lisp the Language, Second Edition". Digital Press, 1990.

[Str97] B. Stroustrup. "The C++ Programming Language". Addison-Wesley, 3rd edition, 1997.

[Sun01] Sun Microsystems. "J2EE Patterns Catalog". Mars 2001. Disponible à : http://developer.java.sun.com/developer/restricted/patterns/DecoratingFilter.html

[Sun-IIOP] Sun Microsystems. "RMI-IIOP Programmer's Guide". Disponible à : http://java.sun.com/j2se/1.4.1/docs/guide/rmi-iiop/rmi_iiop_pg.html

[SV02] J. Sutherland, W. van den Heuvel. "Enterprise Application Integration and Complex Adaptive Systems". Communications of the ACM, Vol. 45, No. 10, octobre 2002.

[SVJ03] D. Suvée, W. Vanderperren, V. Jonckers. "JAsCo: an Aspect-Oriented Approach Tailored for Component Based Software Development". Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03), pp. 21-29, Boston, USA, March 2003.

[SWING] Sun Microsystems. "Introducing Swing Architecture". Disponible à : http://java.sun.com/products/jfc/tsc/articles/getting_started

[Szp00a] C. Szperski. "Point, Counterpoint". Software Development Magazine, février 2000, Disponible à : http://www.sdmagazine.com

[Szp00b] C. Szperski. "Components and Architecture". Software Development Magazine, octobre 2000, Disponible à : http://www.sdmagazine.com

[Szp02] C. Szyperski, "Component Software: Beyond Object-Oriented Programming". Second edition, ACM Press, Component Software Series, Addison-Wesley, 2002.

[Szp99] C. Szperski. "Components and Object Together". Software Development Magazine, mai 1999.

[TCK00] M. Tatsubori, S. Chiba, M.-O. Killijian, K. Itano, "OpenJava: A Class-based Macro System for Java" Reflection and Software Engineering, LNCS 1826, Germany: Springer-Verlag, pp. 119-135, 2000.

[Tou90] D. S. Touretzky. "Common LISP: A Gentle Introduction to Symbolic Computation". Benjamin/Cummings, 1990.

[TOW99] P. Tarr, H. Ossher, W. Harrison, S.M. Sutton. “N Degrees of Separation: Multi-Dimensional Separation of Concerns”. Proceedings of the 2lst International Conference on Software Engineering (ICSE’99), IEEE, mai 1999.

[TT92] K. Takashio, M. Tokoro. “DROL: An Object-Oriented Programming Language for Distributed Real-Time Systems”. Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference (OOPSLA ‘92), ACM Press, Vancouver, Canada, octobre 1992.

[UML97] Object Modeling Group. "Unified Modeling Language version 1.0". Janvier, 1997.

[VE00] J. Villalobos, J. Estublier. "Spécification Formelle et Prototype du Modèle OM du LSR". Rapport Technique Équipe Adèle. juin 2000

[Ver01] H. Verjus. "Conception et construction de fédérations de progiciels". Thèse de doctorat LLP-ESIA, Université de Savoie, Annecy, septembre 2001.

[VH03] M. Veit, S. Herrmann. "Model-View-Controller and Object Teams: a Perfect Match of Paradigms". Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03), pp. 140-149, Boston, USA, March 2003.

[Vil02] J. Villalobos. "APEL: Spécification formelle du moteur". Rapport Technique Équipe Adèle. mars 2002

[Vil86] J. Villalobos. "Automate pour Application Interactive". Rapport de D.E.A., Institut National Polytechnique de Grenoble, France, 1986.

[Voa98] J. M. Voas. "The Challenges of Using COTS Software in Component-Based Development". IEEE Computer, Vol. 31, No. 6, pp. 44-45, June 1998.

[VV01] S. Van den Enden, E. Van Hoeymissen et al. "A Case Study in Application Integration". Proceedings of the OOPSLA Business Object and Component Workshop, 15th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, USA, 2001.

[VW01] W. Vanderperren, B. Wydaeghe. "Towards a New Component Composition Process". Proceedings of ECBS 2001 (Computer-Based Systems), Washington, USA, avril 2001.

[WC00] K. Walrath, M. Campione. "The JFC Swing Tutorial: A Guide to Constructing GUIs", Addison Wesley, 2000.

[WC02] S. Weerawarana, F. Curbera. "Business Processes: Understanding BPEL4WS". IBM TJ Watson Research Center, August 2002. Disponible à : http://www-106.ibm.com/developerworks/webservices/library/ws-bpelcol1

[WD00] R. Weltman, T. Dahbura. "LDAP Programming with Java", Addison-Wesley, 2000.

[WFC99] S. White, M. Fisher, R. Cattell, G. Hamilton. "JDBC API Tutorial and Reference: Universal Data Access for the Java 2 Platform", Addison Wesley, 1999.

[WK99a] J. Warmer, A. Kleppe. "OCL : The Constraint Language of the UML". Journal of Object-Oriented Programming, Vol. 12, No. 2, mai 1999.

[WK99b] J. Warmer, A. Kleppe. "The Object Constraint Language Precise Modeling with UML". Addison-Wesley Object Technology Series, 1999.

[WV01] B. Wydaeghe, W. Vandeperren. "Visual Component Composition Using Composition Patterns". Proceedings of Tools 2001, Santa Barbara, USA, July 2001.

[XML00] World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0, second edition, octobre 2000.
Disponible à http://www.w3.org/TR/REC-xml

[YC79] E. Yourdon, L. Constantine. "Structured Design Fundamentals of a Discipline of Computer Program and Systems Design". Yourdon Press, New York, 1979.

[ZJ98] P. Zave, M. Jackson. “A Component-Based Approach to Telecommunications Software”. IEEE Software, Vol. 15, No. 5, pp. 70-87, septembre 1998.