

## Composants et composition dans l'architecture des systèmes répartis

Thierry Coupaye, Romain Lenglet, Mikaël Beauvois, Pascal Déchamboux

France Télécom R&D (DTL/ASR)  
Chemin du Vieux Chêne, BP 98, 38243 Meylan

---

### Résumé

Cet article défend l'idée que l'architecture des systèmes répartis nécessite des modèles de composants plus généraux que les modèles industriels, et des frameworks permettant de configurer les composants de manière programmatique – et ce de manière statique et dynamique. Ces frameworks doivent au minimum offrir des capacités de *composition structurelle* mais peuvent également offrir d'autres types de composition, définis dans cet article : *opératoire*, *fonctionnelle*, *comportementale*, et *contractuelle*.

**Mots-clés** : architecture logicielle, systèmes répartis, composants, composition, configuration.

---

### 1. Architectures logicielles adaptables

La convergence de l'informatique et des télécommunications – qui se traduit par une explosion du nombre d'objets en communication, possiblement mobile, dans un contexte largement réparti et hétérogène – conduit au concept de *réseau d'information* défini comme un ensemble de ressources, considérés comme des objets de programmation, nécessaires à la fourniture de services. Cette vision d'un cyberspace ubiquitaire nécessite une infrastructure logicielle appropriée, permettant de gérer les différentes ressources matérielles ou logicielles disponibles, et permettant aux applications d'accéder à ces ressources et de programmer leur utilisation.

La grande diversité des contraintes techniques exercées sur ce réseau d'information, la difficulté d'identifier ces contraintes a priori, et l'évolution dynamique de ces contraintes, imposent que les infrastructures logicielles sous-jacentes offrent une grande capacité d'adaptation. L'*adaptabilité* d'un système peut être définie par deux concepts : la *modularité*, i.e., la distinction d'unités constructives dans le système, et la *composabilité*, i.e., la capacité à faire interagir ces unités constructives. Cette adaptation doit pouvoir être à la fois être statique (compilation pour la configuration au déploiement) et dynamique (reconfiguration à l'exécution), pour offrir un compromis entre performances et dynamique suivant le contexte d'exécution et de déploiement.

Pour maîtriser la complexité d'un tel système, il est nécessaire de pouvoir identifier clairement les objets qui le composent et leurs interactions, et donc de disposer d'abstractions appropriées à sa modularité (objets, composants, interfaces, etc.). Il existe de nombreux travaux dans des contextes académiques et industriels, autour de l'architecture logicielle et des composants. Ces travaux sont discutés dans la section 2. La section 3 présente les éléments d'un modèle et d'un framework de composants minimaux, qui généralisent ces travaux.

La section 4 introduit les différentes possibilités en ce qui concerne la composabilité d'un système. Nous distinguons globalement cinq types de composition : *structurelle*, *opératoire*, *fonctionnelle*, *comportementale* et *contractuelle*. Nous montrons les relations entre ces types de composition, et comment le framework de composants que nous proposons peut être étendu pour supporter ces types de composition. La dernière section conclut et présente les perspectives de ce travail.

### 2. Architectures logicielles et modèles de composants

#### Architecture et ADLs

L'architecture logicielle [9] fait l'objet de nombreux travaux depuis une dizaine d'années, en particulier dans le domaine du génie logiciel et des systèmes répartis. Il n'existe pas véritablement pour l'instant de

consensus quant à sa définition, son utilisation ou les outils qui devraient lui être associés. L'*architecture* d'un système peut être définie comme une spécification abstraite de ce système en termes des *modules* – ou *composants* – qui le constituent et des interactions entre ces composants. La notion d'architecture véhicule principalement une distinction nette entre la structure conceptuelle d'un système et ses possibles réalisations techniques (implantations) sur des cibles particulières (langages et systèmes d'exploitation).

Les *langages de description d'architectures* (ADLs en anglais) constituent l'un des aboutissements des travaux sur l'architecture logicielle. Suivant [16], nous définissons un ADL comme (1) une notation textuelle ou graphique formelle ou semi-formelle qui permet de spécifier des architectures logicielles, (2) accompagnée d'outils spécifiques. En ce qui concerne les notations offertes, on constate une très grande diversité entre les ADLs. Néanmoins, les trois concepts de *composant*, *connecteur* et *configuration* sont généralement acceptés comme essentiels<sup>1</sup>. Un composant correspond à la définition intuitive que l'on peut en avoir (!) et qui en fait un *module* ou encore un *objet*, bref "une unité de calcul ou une donnée" [16]. Un connecteur est une entité architecturale qui modélise les interactions entre composants et qui régit ces interactions. Composants et connecteurs sont généralement accessibles uniquement au travers d'*interfaces* (que l'on appelle parfois *ports* pour les composants et *rôles* pour les connecteurs). Les interfaces des composants expriment les *services fournis* (interfaces à la Java) et les *services requis* par ces composants. Les composants peuvent généralement exposer des *interfaces multiples*, i.e., fournir plusieurs services. Ils peuvent être typés ou non. La notation peut permettre ou non de spécifier leur *sémantique* ou de leur associer des *contraintes*, voire des *propriétés non fonctionnelles*. Enfin, une configuration est un *agencement*, une *topologie*, bref un graphe de composants et de connecteurs qui décrit une structure architecturale. Parmi les propriétés associées aux configurations dans les différents ADLs, la composition/décomposition *hiérarchique* est généralement considérée comme primordiale. Cette propriété permet de spécifier un composant comme une combinaison de (sous-)composants qui sont à leur tour décomposables, etc. Les outils associés aux différents ADLs<sup>2</sup> reflètent deux courants animant cette communauté. On trouve d'une part des ADLs dont la vocation est essentiellement la formalisation, la vérification et la validation d'architectures (UniCon, Wright, Rapide, SADL) ; et d'autre part, des ADLs dont la vocation est la génération ou configuration de systèmes (souvent des squelettes de systèmes plutôt que des systèmes complets) et parfois la prise en compte de la dynamique des configurations construites (Weaves, MetaH,Olan). Certains ADLs comme Darwin ou C2 combinent les deux approches.

Les deux courants mentionnés ci-dessus sont également intéressants. Toutefois, la génération de code offerte par la plupart des ADLs semble montrer ses limites en ce qui concerne la construction de systèmes répartis de grande taille. Notamment, de tels systèmes nécessitent des possibilités de configuration statique qu'offrent certains ADLs, mais également de (re)configuration dynamique que gèrent mal les ADLs pour l'instant. En effet, l'approche "génération de code" permet de créer une liaison entre une architecture et sa réalisation mais cette liaison est unidirectionnelle (une modification de l'architecture d'un système entraîne la génération de la nouvelle implantation de ce système) alors qu'elle devrait être, selon nous, bidirectionnelle dans une approche intégrée pour la construction de systèmes répartis (une modification de l'implantation devrait être rendue visible dans l'architecture). Ceci montre le besoin de "construire un pont" entre architecture et implantation et, pour ce faire, de représenter et manipuler des architectures de manière programmatique au travers d'un modèle et d'un *framework* de composants.

## Modèles de composants industriels

San Francisco d'IBM (SF) [1], les Enterprise Java Beans (EJB) de Sun [7], le modèle de composants CORBA (CCM) [3] et COM de Microsoft [17] sont les quatre modèles de composants et *frameworks* industriels les plus répandus. Ils sont bâtis sur les mêmes principes généraux, qui sont exposés ci-après, et ils convergent rapidement : intégration SF-EJB, EJB-CCM, prise en compte des communications asynchrones par CCM puis EJB puis COM, etc.

SF, EJB, CCM et COM sont des modèles de *composants applicatifs ou métiers*. Les composants représentent des processus (transferts bancaires, gestion d'ordres) ou des entités (comptes bancaires, employés,

---

<sup>1</sup>bien que les concepts de *connecteur* ou *configuration* puissent être absents de certains ADLs. Certains considèrent également que le fait de manipuler explicitement des connecteurs est un trait particulier des ADLs les distinguant notamment des MILs (Module Interconnection Languages) [19] ... Signalons également ACME ("l'XML des ADLs") qui représente un format pivot permettant l'échange de descriptions architecturales entre ADLs et l'intégration des outils associés, et dont l'un des mérites est d'explicitier les points communs et différences entre ADLs.

<sup>2</sup>Les références bibliographiques des ADLs cités peuvent être trouvées notamment dans [16][8].

clients, etc.) utilisés couramment dans différents domaines d'activité d'une entreprise (finance, paie, gestion de stocks, de commandes, etc.). Les composants sont déployés dans des structures d'accueil : les *conteneurs*<sup>3</sup>. Les conteneurs ont deux rôles principaux : (1) ils gèrent le cycle de vie (création, accès, activation, passivation, destruction, etc.) des composants qu'ils contiennent et (2) leur fournissent automatiquement et de manière transparente un certain nombre de services techniques<sup>4</sup> (persistance, comportement transactionnel, concurrence, sécurité, etc.) par interception des interactions avec ces composants. Le paramétrage de ces services techniques est effectué de manière déclarative, au moment du déploiement, au travers de *descripteurs de déploiement*<sup>5</sup>. Les différents modèles décrivent principalement des *modèles de programmation* qui spécifient en réalité des *contrats* entre les composants et les conteneurs. Il existe un certain nombre de *types de composants* prédéfinis dans chaque modèle (*sessions, entités, procédés, messages, etc.*). Une *granularité tacite* est associée aux types de composants (les entités sont de "petits composants", tandis que les procédés sont de "plus gros composants"). Les contrats dépendent de ces types des composants. Ils spécifient notamment quels services techniques sont associés à chaque type de composants.

Il existe, bien sûr, beaucoup de différences entre les différents modèles, que nous ne pouvons toutes décrire ici. Ces différences concernent les points abordés ci-dessus : types de composants considérés (sessions sans états dans COM - sessions, entités, messages dans EJB - entités, dépendants, commandes dans SF - sessions, entités, processus, services dans CCM), services techniques associés à chacun de ces types (nommage, cycle de vie, transactions, sécurité dans COM - cycle de vie, nommage, persistance, transactions, concurrence, sécurité dans EJB - idem + notification d'événements dans CCM - idem + requêtes dans SF), modèles de programmation associés. Parmi les différences notables, signalons que SF et COM, à la différence de EJB et CCM, ne sont pas des spécifications mais des produits (SF par exemple propose plus de 1100 composants directement utilisables). Signalons également que CCM et COM sont les seuls modèles à supporter des composants offrant des *interfaces multiples* (dont une par défaut, QueryInterface de IUnknown, dans COM qui permet de naviguer entre les multiples interfaces des composants).

Si l'on compare les modèles de composants industriels aux modèles issus du domaine de l'architecture logicielle et des ADLs, force est de constater l'existence d'un fossé certain en défaveur des modèles industriels. La notion de connecteur est absente des modèles industriels. La notion de configuration est également simplement inexistante (SF, EJB, COM) ou embryonnaire : il existe bien un concept d'*agencement* dans CCM mais il n'est pas formellement défini et un agencement, i.e., une composition de composants, n'est pas un composant ! Ceci est sans doute l'une des plus grosse lacune des modèles industriels. Ces modèles sont "plats" du point de vue de la composition : il n'y a pas de composition hiérarchique. Un composant ne peut pas être lui-même composé de sous-composants.

SF, EJB, CCM et COM sont des modèles de composants métiers utilisables industriellement par des programmeurs d'applications. Ils sont bien adaptés à ce pour quoi ils sont prévus : la programmation, le déploiement et la réutilisation des aspects applicatifs des systèmes répartis. Néanmoins, nous arguons que les manques de ces modèles vis-à-vis de la problématique générale de la composition exposés ci-dessus ; et surtout l'architecture fermée et sous-spécifiée des infrastructures afférentes (serveurs et conteneurs) n'en font pas de bons outils pour composer des composants autres que des composants métiers tels que les composants d'infrastructures (services techniques).

### 3. Un modèle de composants général

Nous avons montré jusqu'ici que l'apport potentiel des travaux sur les architectures logicielles pour la construction de systèmes répartis est conditionné par la capacité à représenter et manipuler ces concepts de manière programmatique au travers de *frameworks* (Section 2). Nous avons ensuite montré que les *frameworks* de composants industriels existants, s'ils représentent certainement une voie viable pour la modélisation et la programmation de composants métiers (applicatifs), ne sont pas adaptés à la construction et à la composition de composants plus généraux comme les composants d'infrastructure, i.e., des composants réalisant des services techniques transversaux par rapport aux composants métiers. Dans la

<sup>3</sup>On peut considérer l'infrastructure transactionnelle MTS (Microsoft Transaction Server) comme le conteneur des composants COM.

<sup>4</sup>On parle également d'*aspects non fonctionnels*.

<sup>5</sup>Pour COM, le registre de Windows est utilisé comme descripteur de déploiement.

suite, nous introduisons les caractéristiques centrales, selon nous, pour la construction d'un modèle de composants général, ainsi que l'ébauche d'un *framework* basé sur ce modèle.

### 3.1. Caractéristiques visées

#### Généralité

Un modèle de composants général ne fait aucune hypothèse particulière sur la nature, le type ou la granularité des composants qu'il permet de modéliser. A la différence des modèles de composants industriels qui permettent de manipuler uniquement des composants métiers, notre modèle doit permettre de manipuler des composants de toute nature et en particulier des composants réalisant des services techniques. De la même manière, à la différence des modèles de composants industriels qui proposent un certain nombre de types de composants prédéfinis qui permettent de modéliser par exemple des processus ou entités liés à l'activité d'une organisation, notre modèle doit permettre de manipuler des composants qui n'ont pas de type particulier, ou bien de définir à loisir de nouveaux types par dérivation des types existants. Enfin, alors qu'une granularité implicite est associée à chaque type de composants dans les modèles industriels, notre modèle doit gérer des composants dont la granularité peut être très variable, allant d'une table de hachage à un conteneur EJB par exemple.

#### Composition structurelle imbriquée

Notre modèle de composants doit permettre une *composition structurelle imbriquée*. Par composition structurelle, nous entendons une composition basée principalement sur les *dépendances* entre composants (interfaces fournies et requises), c'est-à-dire une composition basée sur la structure des composants et leurs connexions – et non sur leur comportement ou des propriétés qui leur seraient associées (ce point est abordé plus longuement dans la Section 4). La relation entre un "sous-composant" et un composant "de niveau supérieur" est une relation client-fournisseur : un "sous-composant" fournit un service à un ou plusieurs composants<sup>6</sup>. Par ailleurs, du fait du partage de composants (décrit ci-après), les structures représentant des composants ne sont pas des arbres, mais des *graphes orientés connexes enracinés* (i.e. ayant une racine unique). Nous parlons donc de composition imbriquée plutôt que de composition hiérarchique.

#### Partage entre composants

Une caractéristique plus originale de notre modèle de composants est qu'il doit permettre le partage de composants : un composant peut être un "sous-composant" (fournisseur) de plusieurs composants. Un serveur de noms est un exemple de composant fortement partagé. Le partage de composants est intéressant du point de vue de la réutilisation, des performances de la phase de configuration des composants, et des performances en général des systèmes construits à base de composants car il évite certaines redondances. On peut imaginer par exemple un composant cache qui serait partagé entre un adaptateur d'objets (OA, dans CORBA) et un support de persistance.

#### Configuration statique et dynamique

Le modèle de composants et le *framework* associé doivent offrir un continuum pour la configuration statique et dynamique des composants : composants et configurations étant des objets manipulables par programme, il est possible de configurer statiquement et dynamiquement les composants lors de leur déploiement (et à plus long terme de les reconfigurer dynamiquement, i.e., sans arrêter un système en cours d'exécution).

### 3.2. Éléments du framework

Nous introduisons dans cette section les principaux éléments d'un framework de composition en cours de définition. Nous essayons d'avoir une approche minimaliste, basée sur quelques principes structurant tels que (1) utilisation du minimum de liens statiques (dans le code source), mais (2) réification des liens à l'exécution dans des espaces de désignation, et (3) entre interfaces uniquement.

---

<sup>6</sup>Cette notion de relation correspond à la notion de lien (binding) dans RM-ODP [12].

## Composants

Un composant est une entité contrôlée par le *framework* à l'exécution, c'est-à-dire une instance d'un type de composants. C'est est une composition de deux objets : l'objet (métier ou technique) que l'on désire percevoir comme un composant, et un objet de contrôle. Ces deux objets réalisent les deux rôles de *contrôleur* et *contrôlé*<sup>7</sup>. L'objet de contrôle, piloté par le framework, gère le cycle de vie et les connexions du composant avec d'autres composants. L'objet de contrôle est en particulier chargé de la configuration en trois phases du composant : (1) la création – ou l'accès au travers d'un service de noms ou un service de persistance si le composant existe déjà, (2) la connexion et (3) l'initialisation du composant. Pour ce faire, il a accès à des (méta)informations sur le composant : interfaces fournies, dépendances entre ce composant et les autres composants qui lui sont nécessaires (interfaces requises), et paramètres d'initialisation. Ces informations sont accessibles par l'intermédiaire d'un certain nombre de *descripteurs* décrits ci-après et également de par la nature même des composants qui sont des contextes de nommage de la forme  $\{(\text{nom}, \text{descripteur d'interface})\}$  qui expriment les *accointances* de ce composant, c'est-à-dire les composants fournisseurs connectés à ce composant. L'objet de contrôle doit être vu comme l'objet d'interposition dans les EJB (EJBObject). Toutes les interactions avec le composant sont interceptées par cet objet pour gérer le cycle de vie du composant. Comme dans les EJB, cet objet doit être généré par le framework statiquement (compilation, réflexion structurelle) ou dynamiquement (réflexion comportementale).

## Descripteurs

Les descripteurs sont également des objets (à l'exécution). Il en existe différents types :

- un *descripteur de composant* fournit des informations de typage. Il décrit (1) les services fournis par le composant (interfaces fournies) et (2) les dépendances de ce composant vis-à-vis d'autres composants (interfaces requises) exprimées par un contexte de nommage de la forme  $\{(\text{nom}, \text{descripteur de connexion})\}$ .
- un *descripteur de connexion* décrit une liaison entre un composant (client) et des interfaces de composants (fournisseurs) auxquelles il peut être connecté. Le descripteur spécifie (1) le type de l'interface du composant fournisseur, représenté par un descripteur d'interface, (2) la cardinalité de la connexion<sup>8</sup>, et (3) si cette connexion est obligatoire ou non. Les connexions non obligatoires permettent de spécifier des composants optionnels.
- un *descripteur d'interface* fournit principalement le nom de cette interface et des informations permettant de tester la compatibilité entre interfaces<sup>9</sup>.

## Configurations

Une configuration est une usine à composants. C'est un contexte de nommage de la forme  $\{(\text{nom}, \text{configuration})\}$ , qui nomme les accointances des composants créés, et donc qui représente un assemblage de composants. Les configurations sont imbriquées puisque de la forme récursive  $\text{configuration}=\{(\text{nom}, \text{configuration})\}$ . Plusieurs configurations peuvent être connectées à une même configuration, permettant ainsi le partage de composants. Une configuration a deux rôles : elle est capable (1) de vérifier la conformité de l'assemblage qu'elle représente, en termes de compatibilité dans les connexions exprimées par les descripteurs, et (2) d'instancier cet assemblage s'il est conforme. Une configuration est donc un schéma d'instanciation.

## Framework

Le framework est un ensemble d'APIs qui permet (1) de créer des composants, configurations et descripteurs, et (2) d'instancier des configurations, c'est-à-dire de configurer les composants de cette configuration. Le framework est utilisé aussi bien à l'activation d'un système avec sa configuration initiale, qu'en cours d'exécution pour modifier cette configuration. L'instanciation d'une configuration se déroule en plusieurs étapes successives associées au cycle de vie du composant correspondant : *création*, *connexion*,

---

<sup>7</sup>Un composant est donc un *domaine* au sens de RM-ODP [11][13]. Voir également RM-ODP pour la définition des concepts d'*entité*, d'*objet* et de *composition*.

<sup>8</sup>Il peut exister des connexions multivaluées entre un service de persistance et des sources de données ; ou encore entre un conteneur EJB et les beans qu'il exécute, etc.

<sup>9</sup>Le test concerne la compatibilité structurelle dans le framework minimal, mais il peut être étendu pour tester la compatibilité fonctionnelle, comportementale, etc.

*initialisation* (paramétrage), *activation* (démarrage), *désactivation* (arrêt), *déconnexion*, *destruction*.

#### 4. Composition

Nous avons introduit un modèle de composants et un framework de configuration de composants minimal, dans le sens où il offre essentiellement des capacités de composition structurelle basée sur les dépendances entre composants (cf. Section 3.1). Néanmoins, dans son sens le plus général, la composition est définie comme “l’action, manière de former un tout en assemblant plusieurs parties, plusieurs éléments” (dictionnaire Le Robert). Nous introduisons dans cette section quatre autres types de composition qui diffèrent par la nature des éléments assemblés, et par la nature du “tout” obtenu. Ces types de composition correspondent à des niveaux d’abstraction croissants<sup>10</sup> puisqu’ils manipulent des types d’entités de plus en plus abstraits : opérations, fonctions, comportements, contrats. La composition structurelle est considérée comme fournissant le plus bas niveau d’abstraction principalement car elle est nécessaire aux autres types de composition. A terme, le framework devrait être capable d’intégrer différents types de composition afin de satisfaire différents besoins vis-à-vis de la composition.

##### Composition opératoire

Nous définissons la *composition opératoire* comme l’amalgame, la fusion (inlining), de *parties de code* (source ou compilé) d’un système. L’objectif ici n’est plus seulement architectural, mais concerne l’optimisation en terme d’empreinte mémoire (primaire et secondaire) et/ou de performance (du fait de la réduction du nombre d’objets en mémoire applicative). Cette approche repose sur une algèbre basée sur les opérateurs mixins [6] : fusion, surcharge, renommage, etc., qui permettent de manipuler des hiérarchies de classes d’objets. Une des difficultés majeures dans l’intégration de ce type de composition dans un framework général est de préserver la vision architecturale d’un système (composition structurelle) alors même que les implantations de certains composants peuvent être agglomérées par composition opératoire.

##### Composition fonctionnelle

Les techniques de composition fonctionnelle, appelées aussi techniques de séparation de problèmes (separation of concerns), ont pour objectif de séparer et d’assembler des *vues fonctionnelles*. Ces vues sont combinées pour obtenir une seule vue globale du système, et le comportement global du système est une combinaison du comportement de chaque vue. Ces vues sont indépendantes et spécifiées séparément, et peuvent être utilisées dans plusieurs assemblages différents. Les vues restent séparées après être assemblées.

Les techniques existantes les plus représentatives sont la programmation par aspects (AOP) [14], la programmation par sujets (SOP) [18] et les filtres de composition [2]. Ce qui différencie ces techniques est la nature des vues assemblées, et les capacités d’expression pour construire les assemblages, notamment les points du système où ces vues sont mises en relation.

Dans la programmation par aspects (AOP), les vues sont appelées “aspects”. L’objectif est d’éviter les cas où des aspects d’un système ne sont pas encapsulables dans des modules distincts (par exemple par composition structurelle), parce que le code de ces aspects est “dispersé” dans tout le système, ou que des parties du système sont un “mélange” du code de plusieurs de ces aspects. Des exemples de tels aspects sont la gestion de la concurrence d’accès à des objets, la gestion des transactions, etc. L’AOP permet de résoudre ces problèmes, en permettant de construire un système en deux étapes : (1) définition des aspects indépendamment les uns des autres, et (2) composition des aspects pour former le système global. Les points dans le système où peuvent être combinés les aspects sont appelés “points de jonction”, et sont spécifiques à chaque technique d’AOP. Donc comme les aspects sont assemblés après avoir été définis, ils sont peu couplés entre eux, et réutilisables dans d’autres assemblages.

La programmation par sujets (SOP) a pour objectif de pouvoir définir des classes d’objets de manière décentralisée. Pour cela chaque vue d’un système, appelée “sujet”, définit un ensemble de classes ou de fragments de classes. Chaque sujet définit donc séparément une partie du comportement et de l’état d’un ensemble d’objets du système à l’exécution, cette définition étant réalisée avant l’assemblage des sujets. Contrairement aux aspects dans l’AOP, les sujets peuvent définir un fragment de chaque classe d’objet du système, et donc être dispersés dans tout le système.

---

<sup>10</sup>Ces niveaux ne sont cependant sans doute pas complètement disjoints...

Les filtres de composition permettent d'étendre des objets d'un système qui interagissent par envois de messages, par interception de ces messages. Ainsi, chaque message reçu ou émis par un objet est traité par une liste de filtres, qui réalisent des traitements, et donc étendent le comportement de l'objet. L'objet original reste inchangé, et un ensemble de filtres permet donc de définir une vue particulière de cet objet. Les filtres sont réutilisables pour adapter de nombreux objets différents, et sont donc séparés.

Assembler des "vues" comme les aspects ou les sujets peut nécessiter de modifier ces vues, par exemple par composition opératoire pour insérer le code d'une vue dans une autre. Mais cette modification est automatique lors de l'assemblage, et n'impacte pas la définition des vues. Nous considérons donc que la composition fonctionnelle est à un niveau d'abstraction supérieur à la composition opératoire.

Ce qui caractérise la composition fonctionnelle est une définition séparée des vues d'un système, avant leur assemblage. Mais ces vues peuvent être difficiles à séparer (vues non-orthogonales). Par exemple, une vue qui gère la persistance des objets et une vue qui gère les transactions sont fortement interdépendantes. De plus, il n'existe pas de démarche ou de théorie générale pour séparer les vues d'un système, et cela doit être réalisé manuellement. De plus, le comportement des vues n'est pas spécifié de manière formelle, mais est directement implanté dans le code de ces vues.

Dans le contexte de la construction de systèmes répartis, nous utiliserons par exemple la composition fonctionnelle pour mettre en relation différents services techniques, dans des objets de contrôle du framework que nous proposons (cf. Section 3.1). Ainsi, nous générerons statiquement ou dynamiquement des objets de contrôle, qui seront une composition fonctionnelle de plusieurs "vues de contrôle".

### Composition comportementale

Une autre approche pour assembler un système à base de composants consiste à s'appuyer sur un modèle synchrone réactif [10] [4]. En effet, un composant, outre les interfaces qu'il propose (cf. composition structurelle) et donc les fonctionnalités qu'il exhibe (cf. composition fonctionnelle), affiche un *comportement* propre. Ce comportement peut être représenté, par exemple, par des automates. Le problème de la composition de composants (en particulier techniques) peut donc être envisagé comme un problème de composition d'automates. Le langage synchrone Esterel [5] possède, à priori, des propriétés intéressantes pour une gestion fine de la composition comportementale : déterminisme, réactivité, cohérence temporelle des signaux et réactions. Par ailleurs, des techniques de validation du comportement global d'une composition de composants s'appuyant sur la vérification formelle (model-checking symbolique [15]) et la simulation pourront être appliquées pour certifier que la composition résultante est fiable.

Les problèmes soulevés par la modélisation du comportement des composants concernent la généralité (est-on capable de trouver un formalisme suffisamment général qui permette de représenter le comportement de tous les composants?) et l'utilisabilité (quel formalisme offrir à l'utilisateur final?) de l'approche. Une modélisation sous forme d'automates est caractérisée par des états (initiaux et finals) et des transitions. Une transition est de la forme  $\langle \text{événement} \rangle \langle \text{condition} \rangle \langle \text{action} \rangle$ . De nombreux types d'événements peuvent être intéressants pour la composition comportementale : les interactions avec des composants, les événements internes aux composants, en particulier ceux liés à leur cycle de vie, les événements provenant de l'environnement système (mémoire insuffisante par exemple), les exceptions, etc. Les conditions portent sur l'état interne des composants et/ou leur contexte d'exécution. Les actions peuvent être des exécutions d'opérations, des modifications des variables de contexte, etc.

Nous considérons que la composition comportementale est à un niveau d'abstraction supérieur à la composition fonctionnelle, car elle permet de spécifier formellement le comportement de "vues fonctionnelles" d'un système.

Il est difficile de déterminer le comportement global du système. Notre objectif est de générer, à partir des spécifications comportementales de chaque composant et des propriétés liées à la composition, le comportement global et déterministe de ce dernier. Pour atteindre cet objectif, une algèbre de composition et une grammaire d'expression des propriétés sont en cours de définition. Ces propriétés vont influencer sur la construction du comportement global dans le sens où elles vont soit ajouter du comportement soit modifier le comportement initial dans les différentes spécifications des composants. Un des problèmes soulevés par la composition de comportements pourrait être l'explosion combinatoire résultant de la composition de nombreuses spécifications complexes.

La composition comportementale devra bien sûr être déclinable dans le contexte général du framework de composition exposé dans la Section 4 qui s'inscrit dans un modèle de programmation impératif

classique. Le framework utilisera le concept de comportement de composants et devra être capable de manipuler cette abstraction pour l'assemblage d'un système. Pour cela, les interactions avec les composants seront interceptées et adaptées par des objets de contrôle spécifiques gérant la composition comportementale, ces objets de contrôle étant générés automatiquement lors de la configuration.

### **Composition contractuelle**

Nous appelons ici *contrat* une spécification permettant de relier certaines obligations d'un composant à certaines hypothèses sur son environnement. Un contrat peut contenir uniquement des obligations (e.g. une source de trafic) ou uniquement des hypothèses (e.g. un niveau de qualité de service requis par un utilisateur). L'objectif est de pouvoir spécifier des contraintes portant, par exemple, sur la création (typage..), le comportement ou la qualité de service d'un composant. Nous appelons *composition contractuelle* la vérification que la combinaison de plusieurs *contrats élémentaires* forment un contrat valide plus global, pouvant, éventuellement, être porté par un composant de granularité supérieure. L'objectif est de pouvoir vérifier qu'un assemblage de composants ne possède pas des propriétés contradictoires. Par exemple, trivialement, pour vérifier que plusieurs contraintes de temps de service ne violent pas un temps de réponse maximal exigé globalement. Pour cela, une technique de composition contractuelle peut raisonner sur des comportements spécifiés par composition comportementale. La composition contractuelle est donc à un niveau d'abstraction supérieur à la composition comportementale.

## **5. Conclusion**

La première partie de cet article a souligné des limitations des modèles de composants industriels et issus des travaux en architecture logicielle : manque d'abstractions manipulables et faiblesse à modéliser des composants d'infrastructure. Face à ces limitations, nous avons défini un framework de composants minimal, qui offre des fonctionnalités essentielles de composition structurelle, et qui est extensible pour prendre en compte d'autres types de composition.

Les différents types de composition que nous avons distingués ont des objectifs différents, des niveaux d'abstractions différents, et donc résolvent des problèmes différents. Les techniques de composition peuvent être complexes et coûteuses globalement, même si elles simplifient la résolution de problèmes particuliers. Il est donc souhaitable de ne pas utiliser une seule technique ou un seul type de composition pour construire un système. Nous proposons au contraire de combiner plusieurs techniques, pour construire des systèmes simples et dont les capacités d'adaptation correspondent à nos besoins.

Le framework que nous avons proposé sera étendu pour prendre en compte, en particulier, la composition fonctionnelle et la composition comportementale. Nous allons expérimenter la mise en oeuvre de ce framework, et l'utilisation conjointe de plusieurs types de composition, dans la construction d'un conteneur EJB adaptable. Il n'existe actuellement pas de démarche permettant de déterminer de manière systématique quel type de composition (et quelle technique particulière) doit être utilisé pour résoudre le plus simplement possible un problème donné dans la construction d'un système. Cette expérimentation nous permettra donc de déterminer les points d'adaptation stratégiques dans ce contexte, et des cas d'utilisation privilégiés des différentes techniques de composition, notre objectif à long terme étant globalement de faciliter la construction de systèmes répartis, et d'améliorer leur capacités d'adaptation et d'administration.

### **Remerciements**

Cet article présente des idées discutées avec de nombreux collègues, notamment Jean-Bernard Stefani (INRIA) à propos du calcul des domaines [13] qui sous-tend le modèle de composants proposé, Nicolas Rivierre (France Télécom R&D) à propos de la composition contractuelle et Luciano Garcia-Banuelos (IMAG) à propos de la composition opératoire.

### **Bibliographie**

1. D. H. Andrews. IBM's San Francisco Project : Java Frameworks for Application Developers, December 1996.
2. Mehmet Askit, Lodewijk M.J. Bergmans, and Sinan Vural. An Object-Oriented Language-Database Integration Model : The Composition-Filters Approach. In *Proceedings of the 6th European Confe-*

- rence on *Object-Oriented Programming (ECOOP'92)*, volume 615, pages 372–395. Springer-Verlag, June 1992.
3. BEA Systems et al. CORBA Component Model Joint Revised Submission. Object Management Group, OMG Document orbos/99-07-01 ed., July 1999.
  4. G. Berry and A. Benveniste. The Synchronous Approach to Reactive and Real-Time Systems , Proc. of the IEEE, vol. 79, n° 9, September 1991.
  5. Gerard Berry and Georges Gonthier. The Esterel Synchronous Programming Language : Design, Semantics, Implementation. *Science of Computer Programming*, 19(2) :87–152, 1992.
  6. G. Bracha and G. Lindstrom. Modularity Meets Inheritance. *Proc. of the IEEE Int. Conf. on Computer Languages*, 1992.
  7. Linda G. DeMichiel, L. Ümit Yalçinalp, and Sanjeev Krishnan. Enterprise Java Beans Specification Version 2.0 Proposed Final Draft 2. Sun Microsystems Inc., April 24, 2001.
  8. D. Garlan, R. Monroe, and D. Wile. ACME : An Architecture Description Interchange Language. In Proceedings of CASCON'97, pages 169– 183, Toronto, Ontario, November 1997.
  9. David Garlan and Mary Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1, pages 1–40. World Scientific Publishing Company, 1993.
  10. Nicolas Halbwachs. Synchronous programming of reactive systems. In *Computer Aided Verification*, pages 1–16, 1998.
  11. ISO. ITU/ISO Reference Model of Open Distributed Processing – Part 2 : Foundations, International Standard ISO/IEC 10746–2, ITU–T Recommendation X.902, 1995, 1995.
  12. ISO. ITU/ISO Reference Model of Open Distributed Processing – Part 3 : Architecture, International Standard ISO/IEC 10746–3, ITU–T Recommendation X.903, 1995, 1995.
  13. JB.Stefani, F.Germain, and E.Najm. Elements of Objects-Based Model for Distributed and Mobile Computation. In *Proceedings of the IFIP TC6/WG6.1 Fourth International Conference on Formal Methods for Open Object-Based Distributed Systems, Stanford, California, USA*, September 2000.
  14. Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97), Jyväskylä, Finland*, volume 1241, pages 220–242. Springer-Verlag, 1997.
  15. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.
  16. N. Medvidovic. A Classification and Comparison Framework for Software Architecture Description Languages. Technical Report UCI-ICS-9702, Department of Information and Computer Science, University of California, Irvine, February 1997.
  17. Microsoft Corporation. The Component Object Model Specification, March 1995.
  18. Harold Ossher, William Harrison, Franck Budinsky, and Ian Simmonds. Subject-Oriented Programming : Supporting Decentralized Development of Objects. In *Proceedings of the 7th IBM Conference on Object-Oriented Technology*, 1994.
  19. R. Prieto-Diaz and J. Neighbors. Module Interconnection Languages. *The Journal of Systems and Software*, 6(4) :307–334, November 1986.