

# PROJET RNTL ARCAD\*

## D4.1 Canevas logiciel pour la construction de noyaux de système d'exploitation dynamiquement configurables

Coordonnateur : Daniel Hagimont  
Projet Sardes - Laboratoire LSR-IMAG - INRIA Rhône-Alpes  
655, avenue de l'Europe, Montbonnot  
38334 St Ismier Cedex France

1<sup>er</sup> décembre 2002

### Résumé

Ce document a pour objectif de présenter nos travaux dans le domaine des noyaux de système d'exploitation, où des noyaux "sur mesure" sont développés pour des applications spécifiques (informatique embarquée, temps réel). Nos travaux s'appuient sur le canevas logiciel THINK [REF], initialement développé à France Télécom R&D. Ils visent à apporter flexibilité et configurabilité, à la fois statiquement et dynamiquement, dans des noyaux de système d'exploitation. Notre principale contribution porte sur l'élaboration d'une structure à composant, qui fournit aux noyaux construits des capacités de composition et de reconfiguration dynamique.

## 1 Introduction

L'émergence des systèmes embarqués engendre de nouvelles contraintes de fonctionnement tant au niveau de la taille mémoire des systèmes, des capacités de calcul, qu'au niveau de la mobilité physique. Le défi pour ces systèmes contraints porte sur le maintien de performances acceptables, en éliminant les fonctionnalités haut-niveau superflues, et en personnalisant le plus possible les caractéristiques du système pour qu'il soit adapté au support matériel. Il faudrait donc disposer d'une technologie logicielle susceptible de se déployer sur une vaste gamme de machines.

---

\*Le projet ARCAD (Architecture Extensible pour Composants Adaptables) a été labellisé en décembre 2000. Les différents partenaires sont France Télécom R&D (équipe ASR - Thierry.Coupage@francetelecom.com), INRIA (projet Oasis - Denis.Caromel@inria.fr, projet Sardes - Daniel.Hagimont@inria.fr), Ecole des Mines de Nantes (équipe OCM - Thomas.Ledoux@emn.fr), le laboratoire I3S commun à l'Université de Nice - Sophie Antipolis et CNRS (projet Rainbow - Anne-Marie.Pinna@unice.fr). Le coordonnateur du projet est Michel.Riveill@unice.fr

De plus, une adaptation de l'infrastructure selon les applications qu'elle doit héberger est essentielle. Il s'agit d'adapter les fonctions du système aux conditions d'exécution des applications, permettant de viser des domaines applicatifs très spécifiques, comme le temps-réel ou le multimédia.

Il s'avère donc nécessaire de se doter de mécanismes pour adapter de tels systèmes, aussi bien selon le support matériel sous-jacent, que selon les domaines d'applications visés. L'adaptation se doit d'être dynamique. En effet, la plupart des systèmes ubiquitaires, nomades et embarqués interagissent de manière étroite avec un environnement qui évolue en cours d'exécution. Par exemple, considérons un serveur d'un opérateur télévisuel qui pourvoit en signaux vidéo des clients très diversifiés, allant de récepteurs de télévision classiques à des objets mobiles communicants. La mobilité de ces appareils induit d'importantes variations dans les caractéristiques de la connexion réseau. Le canal de communication établi entre le client et le serveur a besoin d'évoluer pour répondre à ces variations, sans pour autant être arrêté. Reconfigurer et adapter des systèmes en cours d'exécution est une de nos préoccupations majeure.

Nous avons adopté THINK [FSLM02] comme base de développement, que nous avons enrichie d'un canevas logiciel d'introspection et de configuration dynamique. Pour cela, nous avons étendu le modèle de composant initial pour permettre des reconfigurations dynamiques à faible coût. Le programmeur peut charger, décharger des composants dynamiquement, reconfigurer les liaisons, désigner et introspecter des composants pendant leur exécution. En plus du support de reconfiguration, le nouveau modèle de composants est totalement récursif. L'architecture THINK peut donc être uniformément appliquée à tous les niveaux.

Ce document se propose d'exposer le modèle de composition et de reconfiguration de noyaux de système d'exploitation. Il est organisé en trois parties. Dans un premier temps, nous présentons la démarche que nous avons suivie pour l'élaboration du canevas. Ensuite, nous exposons le modèle de composition, tel qu'il a été intégré à THINK. Enfin, nous proposons un canevas pour la reconfiguration de noyaux de système d'exploitation.

## 2 Démarche suivie

Une infrastructure logicielle facilitant la configurabilité dynamique fournit de fait les éléments nécessaires pour modifier dynamiquement la structure interne des systèmes. Construire un système sous forme d'un assemblage de composants est une manière adéquate de rendre cette structure explicite et de l'exhiber afin de faciliter les opérations de reconfiguration.

De nombreux travaux ont adopté cette approche orientée-composants pour construire des noyaux de système d'exploitation évolutifs. SPIN [BSP<sup>+</sup>95] est un noyau extensible qui peut être spécialisé dynamiquement par chargement d'extensions à l'exécution. eCOS [eCO], MMLite [HF98] et OSKit [FBB<sup>+</sup>97] fournissent une librairie de composants permettant la construction de noyaux par assemblage. Cependant, ces systèmes manquent de flexibilité, notamment au niveau des liaisons. Les interactions et les liaisons entre composants dans ces systèmes sont figées et pré-définies.

Notre objectif est d'apporter modularité et flexibilité dans les noyaux de système d'exploitation en offrant une infrastructure à base de composants pour la

configuration dynamique. Nous avons adopté un canevas logiciel baptisé THINK, comme point de départ pour le développement de notre infrastructure de noyaux de système d'exploitation adaptables et introspectables. Plusieurs raisons ont motivé ce choix.

D'abord, dans l'architecture THINK, les ressources matérielles et logicielles sont présentées sous forme de composants. Les composants peuvent communiquer avec d'autres composants au moyen de liaisons flexibles. Les liaisons peuvent elles-même être constituées d'un assemblage de composants et permettent des interactions entre un ou plusieurs composants. Les diverses formes qu'elles peuvent prendre comportent : une simple association entre un nom et une référence ou une structure plus complexe, comme un canal de communication distribué comportant une pile de protocoles de communication. Ce modèle général de liaisons flexibles facilite la construction de systèmes d'exploitation.

Ensuite, la philosophie de THINK est conforme à celle des exo-noyau [EK95] : ne pas imposer de services système comme l'ordonnancement, la gestion mémoire mais plutôt se limiter à ne fournir que les interfaces des ressources matérielles. Les seules abstractions proposées par défaut sont celles qui réifient les fonctionnalités matérielles de la machine hôte en objets logiciels. La plupart des services système utilisés habituellement dans les systèmes d'exploitation sont disponibles dans une librairie appelée Kortex, mais leur emploi reste entièrement optionnel. Par conséquent, des noyaux spécifiques peuvent être facilement construits par assemblage de composants, depuis les briques de base matérielles jusqu'à des briques plus évoluées.

Nous avons enrichi THINK d'un canevas logiciel à composants, inspiré du modèle Fractal [BCS02]. Ce canevas autorise la définition de composants munis de capacité d'auto-description et de reconfiguration dynamique, ainsi qu'un support d'exécution. L'objet de ce document est de présenter le modèle de composition et de reconfiguration intégré dans l'architecture THINK, se référer à [FSLM02] pour une présentation plus détaillée de THINK.

### 3 Modèle de composition

Le modèle de composition et de reconfiguration que nous proposons d'intégrer à THINK est une implémentation du modèle Fractal simplifié. Ce dernier est une tentative d'unification des différents modèles définis au sein des produits du consortium ObjectWeb. Une première implémentation de ce modèle, nommée Julia, a été développée en Java. Cette implémentation se destine à des systèmes plutôt haut-niveau et s'intègre mal dans un système d'exploitation. Notre objectif est donc de fournir une implémentation en C pour l'intégrer à THINK.

Le modèle de configuration que nous avons introduit dans THINK s'organise autour d'un ensemble réduit de concepts. La plupart d'entre eux sont directement inspirés de Fractal et chacun d'eux est appliqué de manière systématique à tous les niveaux du système. Ces concepts forment le cœur du modèle de composition et de reconfiguration.

## 3.1 Composant

Dans notre modèle, un composant est une **unité de configuration logicielle**, manifestée pendant l'exécution du système. Les fonctionnalités qu'un composant fournit sont utilisables et réutilisables dans de nouvelles compositions. Pour distinguer un composant de ses homologues, une identité est associée à chaque composant. Cette identité permet de désigner un composant de manière unique au sein d'une configuration donnée.

### 3.1.1 Granularité d'un composant

Le modèle nous permet de manipuler des composants de n'importe quelle taille. Les composants peuvent en effet être de **granularité arbitraire**. Un élément complexe comme un gestionnaire réseau ou un ordonnanceur, et un simple booléen sont des exemples de composants THINK. Dans cette architecture, le concept de composant ne se limite pas aux parties logicielles d'un système, il permet aussi de modéliser les ressources matérielles comme le processeur, les périphériques, etc.

### 3.1.2 Contrôleur

Un composant est constitué d'un **contrôleur** et d'un **contenu** (voir Figure 1). Le contenu est composé d'un nombre fini de composants qui ont tous

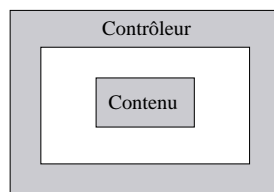


FIG. 1 – Composition d'un composant

connaissance de leur contrôleur. Le contrôleur, quant à lui, est chargé de l'administration du contenu. La notion de contrôleur s'apparente entièrement à celle de conteneur d'infrastructures industrielles (de type EJB).

Le comportement du contrôleur n'est pas défini a priori. Son rôle peut être multiple.

D'abord, toutes les interactions avec le contenu peuvent être interceptées par le contrôleur. Ce dernier peut exporter les interfaces fonctionnelles du contenu et effectuer certains traitements avant et après l'appel à destination du (ou des) composant(s) encapsulé(s). Le contrôleur peut donc être vu comme un objet d'interposition.

Le rôle du contrôleur est également d'ajouter du comportement de contrôle au contenu (*cf.* Figure 3), comme la suspension, la reprise des activités, etc. Pour cela, le contrôleur peut définir certaines interfaces de contrôle. Par exemple, une interface pour le cycle de vie du contenu peut être spécifiée par le contrôleur.

Ensuite, la sémantique de composition n'est pas imposée dans le modèle de composants, elle est à la charge des contrôleurs. Cette approche autorise la coexistence de plusieurs sémantiques de composition au sein d'un même sys-

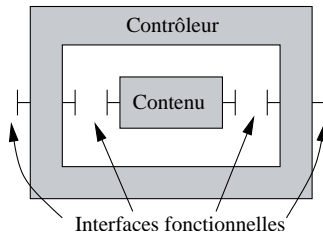


FIG. 2 – Interfaces fonctionnelles du contenu exportées par le contrôleur

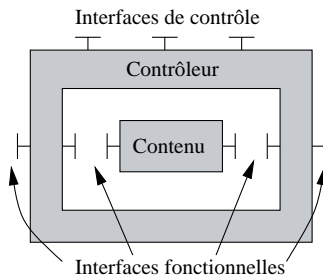


FIG. 3 – Interfaces d’un contrôleur

tème : la composition fonctionnelle (également appelée séparation d’aspects), la composition de type “composition filter” [Ber94], etc.

Enfin, les méta-informations du contenu peuvent être maintenues par le contrôleur. Elles comportent les interfaces serveur, les dépendances entre le contenu et les composants extérieurs (interfaces clientes), les diverses accointances (les composants fournisseurs connectés), etc. La structure du contenu peut être également explicitée.

### 3.1.3 État d’un composant

Un composant interagit d’une manière bien définie avec son environnement au travers de points d’accès identifiés, les interfaces. Cette encapsulation stricte restreint l’accès au composant et permet d’y adjoindre une protection supplémentaire. Elle permet en outre de cacher certains aspects de l’implémentation du composant, tout en permettant une manipulation explicite de son état. Cette séparation claire entre interface et implémentation est essentielle pour la maintenance et l’évolution des applications.

Seules les interactions au niveau des interfaces d’un composant permettent l’accès et la modification de son état. Ainsi, une interaction entre deux composants ne peut affecter directement un composant extérieur. Tout changement d’état d’un composant est donc soit le fait d’une interaction entre lui et d’autres composants, soit la conséquence d’actions internes du composant.

L’état d’un composant à l’exécution est caractérisé par son état d’instance et son état de contrôle. La condition à l’exécution d’un composant, à un instant donné, fait partie intégrante de son état d’instance. Cet état est en effet déterminé par les connaissances que le composant possède sur son environnement. Nous le définissons comme étant constitué de la valeur des variables d’instance et de la valeur des variables locales aux méthodes. L’état de contrôle d’un com-

posant est, quant à lui, réduit aux états constituant son cycle de vie. Deux états sont possibles : l'état "arrêté", et l'état "démarré". Quand un composant est arrêté, il ne peut ni émettre ni recevoir d'appels de méthodes sur ses interfaces fonctionnelles. Les interfaces de contrôle de son contrôleur sont toutefois accessibles. Un composant est démarré quand il peut être initiateur ou destinataire d'interactions avec d'autres composants.

#### 3.1.4 Type d'un composant

Le système vérifie dynamiquement la validité d'une configuration donnée grâce à un **système de type**. Dans une architecture THINK, les interfaces et les composants sont typés. Le type d'un composant comprend au minimum l'union des types de ses interfaces. Le système de type est organisé sous forme d'un treillis, ordonné par une relation de sous-typage.

La possibilité d'exporter dynamiquement des interfaces suppose une reconfiguration possible du type d'un composant. Cette modification de type peut affecter des niveaux différents. Spécifiquement, nous envisageons deux classes de reconfiguration distinctes :

**Au niveau des instances** La modification porte alors sur le type d'une instance particulière. Nous devons par exemple disposer d'un mécanisme autorisant l'ajout d'un sous-composant à un composant englobant et de rendre visible les interfaces serveur par le nouveau composant à l'extérieur de son composite. Le composant englobant doit pour cela exporter les interfaces de son sous-composant, ce qui induit une évolution de son type.

**Au niveau du type** La modification porte alors sur toutes les instances de ce type. Pour des composants de type "serveur", répliqués et distribués sur plusieurs sites, il est par exemple possible de leur adjoindre une interface de persistance afin de gérer la tolérance aux fautes du système. L'interface est alors ajoutée au type, faisant par là même évoluer le type de toutes les instances.

Pour l'instant, nous ne permettons que le premier niveau de reconfiguration de type.

#### 3.1.5 Fabrique de composants

Les composants et leurs contrôleurs associés sont créés à partir de **fabriques**. Une fabrique est un composant et peut, à ce titre, être relativement complexe. Toutefois, la forme la plus simple de fabrique est un constructeur (par exemple, une méthode `new()`), qui à chaque appel, instancie un nouveau composant.

Une fabrique peut être source de plusieurs instances de composants, chaque instance étant d'un même type. Tous les composants instanciés partagent en effet le type de leurs interfaces. Ils ne diffèrent que par leur état, c'est-à-dire la valeur de leurs données et leur état de contrôle. Une fabrique de composants définit donc le type initial de toutes les instances pouvant être créées par la fabrique. Une modification de la fabrique mènerait à une modification du type des instances créées et des futures instances. Cette évolution n'est pour le moment pas permise dans Think.

La Figure 4 présente deux instances d'un composant. Les interfaces fonctionnelles et les interfaces des contrôle des deux composants sont de même type.

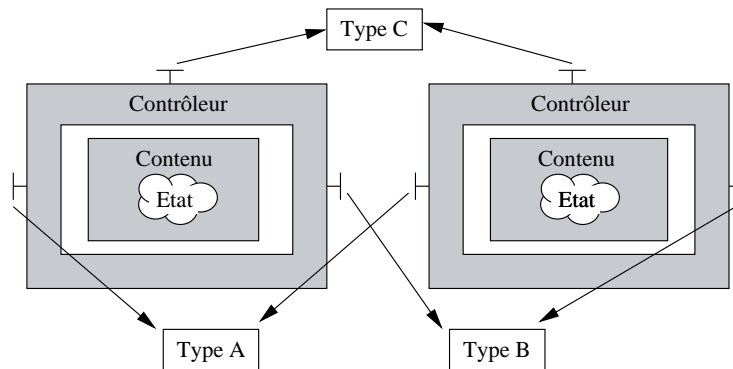


FIG. 4 – Deux instances de composant d’un même type : les interfaces sont de même type mais les états sont différents

La création d’une instance de composant via une fabrique peut être de l’initiative de n’importe quel acteur de l’architecture THINK. Lors du déploiement initial, tous les composants englobant ont la possibilité d’instancier leurs sous-composants. De surcroît, il est possible d’instancier paresseusement un composant, quand le besoin s’en fait ressentir. Un futur client, intéressé par les services du composant, peut très bien l’instancier. Il en est de même pour toute entité du système.

Enfin, c’est au cœur des fabriques que l’initialisation des composants a lieu. Une fois instancié, les différentes liaisons du composant sont établies et le composant est ajouté dans la configuration courante du système.

## 3.2 Interfaces

Toutes les connexions entre composants sont effectuées au travers de points d’accès particuliers, appelés **interfaces**. L’interface d’un composant est définie comme l’ensemble des services serveur et requis du composant, ces services étant déclarés comme un ensemble de signatures de méthodes.

La définition d’une architecture de système informatique repose essentiellement sur la notion d’interface. La séparation stricte entre la spécification de l’interface et son implémentation offre aux composants conteneurs toute leur puissance. Trois principales fonctionnalités peuvent être relevées :

- Interposition : contrôle complet du contenu par le conteneur ;
- Observation, introspection : le conteneur peut détailler l’état du contenu, observer son état ;
- Superposition intrusive : le conteneur peut modifier le contenu.

### 3.2.1 Types d’interface

Deux types d’interfaces existent : les **clientes** et les **serveurs**. Les interfaces clientes sont des ports de sortie pour les composants. Ils permettent l’émission de requêtes vers l’extérieur. Les interfaces serveurs, quant à elles, permettent de réceptionner ces requêtes et de retourner un résultat si nécessaire. Elles sont les ports d’entrée des composants.

THINK suppose que les interfaces sont typées. Le type d'une interface doit comprendre au moins l'union des signatures de ses méthodes. Une signature de méthode est elle-même constituée du nom de la méthode, du type des arguments, et des types retour s'ils existent.

### 3.2.2 Visibilité

La visibilité des interfaces d'un composant dépend de son niveau d'imbrication. L'encapsulation de composants par un composant englobant restreint la visibilité des interfaces des sous-composants au contenu du composant englobant, c'est-à-dire aux composants partageant le même niveau d'imbrication.

La visibilité des interfaces à l'extérieur du contenu est déterminée par le contrôleur du composite englobant. En particulier, le contrôleur peut cacher une interface d'un composant du contenu. Il peut également la rendre visible à son environnement en altérant son nom et son type, ou en surchargeant ses fonctionnalités. Ainsi, les interfaces visibles par l'environnement extérieur sont celles du contenu rendues visibles par le contrôleur ainsi que celles interfaces du contrôleur.

Par exemple, dans la Figure 5, le composant englobant n'exporte pas l'interface serveur par son sous-composant. La visibilité de l'interface est réduite au contenu.

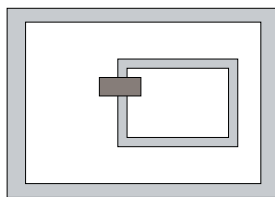


FIG. 5 – Visibilité d'un interface, réduite aux composants du même niveau d'imbrication

Par contre, si le contrôleur du composite englobant décide d'exporter l'interface à l'extérieur, il peut la rendre visible directement, ou bien la redéfinir (cf. Figure 6).

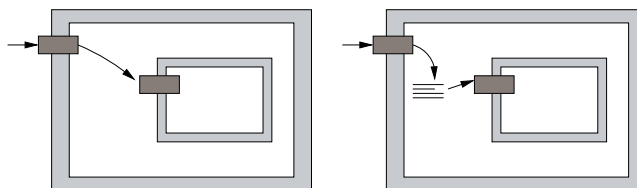


FIG. 6 – Exportation directe ou surcharge d'une interface serveur

Quant aux interfaces clientes, le principe est le même. Le composant englobant peut requérir directement le service auprès de l'environnement, ou peut exporter l'interface cliente en la surchargeant (cf. Figure 7). Dans ce dernier cas, il possède également une interface serveur, qui implémente le service requis par le sous-composants, en faisant appel à un service extérieur.

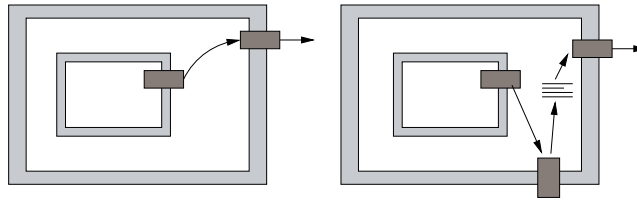


FIG. 7 – Exportation directe ou surcharge d’une interface cliente

### 3.2.3 Nommage

Une interface dans THINK est identifiée par un nom. Un nom est associé à une (et une seule) interface d’un composant, permettant de la désigner de manière non ambiguë par rapport aux autres interfaces.

Les noms sont valides au sein de contextes de nommages. Un contexte de nommage définit des conventions de nommage et offre un moyen de créer des noms. Au sein d’un seul et même système, plusieurs contextes de nommage peuvent co-exister, mais aucune supposition n’est effectuée sur leur organisation. Les contextes de nommage peuvent être imbriqués et peuvent se chevaucher, permettant à un autre d’être valide dans différents contextes. Un contrôleur constitue un contexte de nommage primitif.

Le nommage permet la désignation de ressources variées dans le système. Cette notion est proche de celle des liaisons, permettant l’accès effectif à ces ressources.

## 3.3 Liaison

Les interactions entre composants nécessitent l’établissement de liaisons entre leurs interfaces. Une liaison est un canal de communication entre deux ou plusieurs composants. Typiquement, une liaison encapsule l’ensemble des ressources utilisées pour la communication.

### 3.3.1 Fabriques de liaisons

Une liaison est un composant instancié par une usine à liaisons, appelée aussi fabrique. Les usines à liaisons sont des composants ayant un rôle particulier. Elles permettent, comme leur nom l’indique, de créer et gérer des liaisons entre composants.

Dans le canevas logiciel THINK, une fabrique de liaisons implémente une sémantique de communication particulière pour toutes les liaisons qu’elle instancie. Puisque plusieurs fabriques de liaisons peuvent coexister dans un système THINK donné, il est possible d’interagir avec un même composant selon plusieurs sémantiques de communication (interaction locale, distribuée, avec contrôle d’accès, avec cache...).

### 3.3.2 Formes de liaison

Il est à noter que les liaisons peuvent être créées implicitement (liaisons langage) ou peuvent résulter d’une invocation explicite une fabrique de liaisons.

Les liaisons niveau langage s'apparentent à une association entre un symbole du langage et une adresse mémoire.

Créer une liaison explicitement résulte en la création d'un composant de liaison, c'est-à-dire un composant qui réifie la liaison. Ce dernier peut être administré et contrôlé par d'autres composants. Une liaison explicite, en tant que composant, peut prendre des formes complexes. Elle peut être distribuée, composée de composants ordinaires et d'autres liaisons. Des exemples de liaison incluent des liaisons RPC, des liaisons transactionnelles entre clients et serveurs répliqués).

### 3.4 Configuration

Une configuration THINK est un assemblage de composants logiciels permettant la construction d'applications et de noyaux de système d'exploitation spécialisés. Cet ensemble de composants est structuré en une **composition hiérarchique**. Une composition hiérarchique nous paraît indispensable afin d'envisager le développement de systèmes de grande taille et faciliter la reconfiguration dynamique. Elle permet en plus de gérer la complexité de l'application en raffinant peu à peu son architecture. Une configuration THINK autorise également le partage ou **recouvrement de composants**. Cette caractéristique est nécessaire à la gestion de ressources. Enfin, la composition est **ouverte**, permettant l'observation et la maintenance du système.

#### 3.4.1 Récursivité du modèle

Un composant pouvant être contenu dans un autre composant, le modèle de composition est considéré comme récursif (cf. Figure 8). La récursivité n'a pas de limite : elle peut atteindre n'importe quel niveau depuis les composants matériels. Un composant peut donc être construit en couche, menant à une distribution plus naturelle des différentes fonctionnalités.

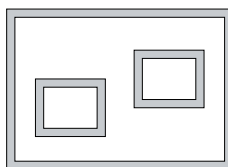


FIG. 8 – Exemple de configuration hiérarchique

La sémantique d'un composant composite est arbitraire. Aucune supposition n'est faite sur sa nature et son rôle. Un composant englobant un nombre fini de composants peut se réduire à une membrane transparente, ou au contraire, peut gérer la répartition, l'accès aux ressources, etc.

Grâce à la récursivité du modèle, le déploiement des applications est grandement facilité. On appelle déploiement l'action d'installer, d'instancier et d'initialiser les divers composants de l'application. Le déploiement d'un composant composite correspond au déploiement de l'ensemble des sous-composants constituant sa configuration.

### 3.4.2 Partage de composants

En plus d'une configuration hiérarchique, il nous semble important de pouvoir autoriser des topologies avec partage, c'est-à-dire autorisant des configurations non strictement arborescentes. Ce besoin de recouvrement est notamment indispensable pour le multiplexage de ressources logicielles et matérielles.

Dans un système THINK, plusieurs composants peuvent partager une partie de leur contenu (cf. Figure 9). En d'autres termes, un composant peut apparaître dans le contenu de plusieurs autres composants.

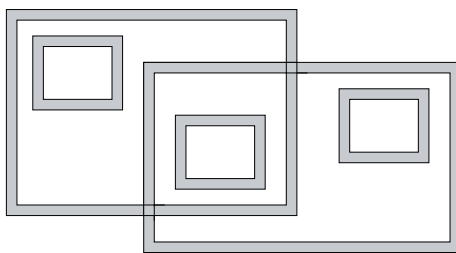


FIG. 9 – Exemple de partage de composant

Un composant partagé parmi deux ou plusieurs composants est soumis au contrôle de leurs contrôleurs respectifs. La sémantique exacte de partage est définie par le contrôleur du composant englobant cette configuration particulière.

### 3.4.3 Maintenance de la représentation à l'exécution

L'administrateur d'un système peut vouloir remplacer dynamiquement un composant en panne par un nouveau, tout en préservant l'intégrité interne du sous-système et en maintenant la continuité du service qu'il fournit. Pour garantir l'intégrité, on a besoin d'identifier le sous-système en tant qu'un assemblage de composants et de manifester à l'exécution les dépendances du sous-système par rapport au nouveau composant. Ces relations de conteneur/contenu et de liaisons doivent être explicites, maintenues et modifiables à l'exécution.

Une configuration THINK possède donc une représentation de sa structure d'exécution, c'est-à-dire de ses composants, de leur état, de leurs interfaces clientes ou serveur et de leurs accointances. Un composant offre une vue de sa représentation interne à travers un niveau arbitraire d'abstractions. La forme de cette représentation peut donc être différente d'un composant à l'autre (par exemple, représentation sous forme d'arbre, de forêt ou de liste).

En outre, la représentation d'un domaine est causalement connectée à sa structure d'exécution. Ainsi, tout changement de la structure d'exécution se répercutera sur sa représentation, la réciproque étant également vraie.

## 4 Modèle de reconfiguration

Un modèle de composants doit permettre la description de changements arbitraires dans une configuration de composants. En particulier, le modèle doit permettre la création dynamique de nouveaux composants et de nouveaux assemblages, comme la migration de composants. Les dépendances de ressources

et de contenu peuvent changer et évoluer avec le temps, soit spontanément, soit en réponse aux interactions avec des composants locaux ou entre les composants et l’environnement extérieur.

Dans THINK, un composant n’est pas seulement une unité de configuration, mais est également une unité de **reconfiguration dynamique**. Notre modèle vise à offrir toutes les briques de base utiles à la reconfiguration dynamique de systèmes répartis ouverts. La reconfiguration n’est pas cablée dans le système mais elle est paramétrable au niveau de chaque conteneur. Ainsi, toutes les opérations de reconfiguration sont sémantiquement liées à leur domaine.

Un noyau THINK n’est pas reconfigurable par défaut. En effet, la mise à disposition de mécanismes de reconfiguration a un coût qui peut ne pas être supporté par tous les environnements (c’est notamment le cas des architectures matérielles embarquées, fortement contraintes en espace mémoire). Toutefois, dans le cas où de tels coûts restent supportables par l’environnement cible, il serait souhaitable de pouvoir disposer de mécanismes de reconfiguration dynamique “à la demande”. De plus, l’architecture doit autoriser la coexistence, dans un même système, de boîtes noires, sans capacité d’introspection ou de reconfiguration, jusqu’à des systèmes boîtes blanches, pleinement réflexives.

Au sein d’une composition donnée, une reconfiguration peut prendre plusieurs formes. On pourra par exemple ajouter ou supprimer un composant d’un composite, remplacer un des composants par un autre de même type ou, tout simplement, changer la configuration interne de la composition. Le modèle de composant de THINK doit donc être suffisamment flexible pour autoriser ces opérations. En particulier, la modification des liaisons entre composants doit être possible à l’exécution. Cette section décrit les opérations de configuration dynamique supportées par une architecture THINK.

## 4.1 Granularité de la reconfiguration

La granularité de la reconfiguration peut être variable. Une reconfiguration peut n’affecter qu’un seul composant du système, ou peut englober un ensemble de composants complexes. Par exemple, une restructuration du système peut amener à changer dynamiquement toutes les liaisons existantes entre composants.

## 4.2 Les initiateurs

N’importe quel élément d’une architecture THINK peut être l’instigateur d’une reconfiguration. Les demandes de modification dynamique doivent être souscrites auprès du contrôleur associé au composant à reconfigurer. Le contenu de ce composant peut être constitué d’un certain nombre d’autres composants. C’est alors au contrôleur du composant englobant d’appeler les opérations de reconfiguration des contrôleurs de ses sous-composants.

Un composant impliqué dans une reconfiguration peut être l’initiateur : un composant peut donc s’auto-configurer. Pour cela, il n’a qu’à émettre une invocation auprès de son contrôleur, qui se chargera d’effectuer les transactions adéquates. Ces opérations de reconfiguration peuvent affecter le contenu (par exemple, lors de la suppression d’un sous-composant) ou le contrôleur lui-même (comme l’ajout d’une interface de contrôle).

### 4.3 Liaisons dynamiques

Les liaisons initiales entre composants sont établies au moment du déploiement du noyau. Le contrôleur du composant englobant le système instancie les sous-composants et met en place toutes les accointances du sous-système.

Pendant l'exécution, des composants peuvent être liés dynamiquement à une configuration THINK. Pour ce qui est de l'adaptation d'une liaison existante entre une ou plusieurs interfaces, la modification se base sur la suppression de l'ancienne liaison et la création d'une nouvelle. Enfin, une simple suppression de liaison peut être gérée pour une liaison explicite comme la suppression d'un composant. Pour une liaison langage, la suppression consiste en une invalidation du symbole utilisé.

Une vérification de type est accomplie à chaque établissement ou modification de liaisons afin de garantir la conformité du lien. Les interfaces reliées doivent avoir une relation de sous-typage, à savoir que toutes les requêtes de l'interface cliente doivent être supportées par l'interface serveur.

### 4.4 Changement de type

L'ajout dynamique d'une interface fonctionnelle à un composant donné induit un changement de type de ce composant. Effectivement, le type d'un composant est défini comme l'union des types de ses interfaces fonctionnelles. Cependant, l'ajout d'une interface au contrôleur de ce même composant n'affectera pas son type. Bien entendu, le type de l'interface n'a pas d'importance : l'interface ajoutée peut aussi bien être une interface cliente qu'une interface serveur.

### 4.5 Changement d'état d'exécution

Traditionnellement, l'application à adapter doit être arrêtée. Cette approche n'est pas souhaitable pour les systèmes critiques qui ne doivent pas être interrompus ou pour les services hautement disponibles, comme les banques, Internet ou les services de télécommunication. Dans ces circonstances, l'adaptation doit être effectuée pendant l'exécution du système.

Cependant, pour conserver la cohérence du système, certaines parties de la configuration doivent suspendre leur exécution. Adapter une application à base de composants signifie, dans la plupart des cas, adapter un ou plusieurs de ses composants, et adapter un composant à l'exécution signifie le déconnecter du système et connecter une nouvelle version du composant et/ou de la configuration. Pour déconnecter un composant ou un ensemble de composant, il est souhaitable, et ceci afin de conserver une configuration cohérente, d'arrêter leur exécution. Une fois que la reconfiguration est accomplie, les composants arrêtés peuvent être redémarrés.

**Arrêt** En particulier, un composant arrêté ne peut recevoir aucune invocation sur ses interfaces fonctionnelles. Toutefois, son contrôleur peut encore répondre à des requêtes des composants de la configuration. Évidemment, un composant suspendu ne peut invoquer des services d'autres composants.

**Démarrage** Un composant qui a dû être arrêté, peut être activé dynamiquement. Démarrer un composant signifie le rendre accessible aux autres compo-

sants du système. Ainsi, un composant démarré, peut aussi bien émettre une invocation d'opération qu'en recevoir.

## 4.6 Ajout de composant

L'architecture THINK offre la possibilité d'ajouter du contenu à un composant. Ajouter un composant à un système dynamique induit une mise à jour structurelle d'une configuration existante. Les systèmes développés avec THINK, ayant la possibilité d'ajouter des fonctionnalités non anticipées lors de leur conception, peuvent être considérés comme extensibles.

Plusieurs types d'ajout sont possibles : l'ajout d'une fabrique, permettant ensuite de créer de multiples instances, et l'ajout d'une instance à partir d'une fabrique.

**Ajout d'une fabrique** Le canevas logiciel THINK permet de charger dynamiquement en mémoire une fabrique de composants. Ainsi, il devient possible d'introduire dans une configuration de nouveaux types de composants à l'exécution.

**Ajout d'une instance** Dès lors qu'une fabrique de composants est chargée en mémoire, il est possible d'instancier des composants. La fabrique détermine le type des instances qu'elle pourra créer.

Une fois qu'une l'instance créée, elle est accueillie par un composant de la configuration, qui l'héberge en tant que sous-composant. Ce composant d'accueil aura la charge d'administrer le composant nouvellement instancié.

## 4.7 Suppression de composant

Similairement à l'ajout, la suppression d'un composant peut s'appliquer à une fabrique ou à une instance de composant. Supprimer un composant revient à libérer la mémoire de toutes les structures de données constituant le composant. Pour effectuer une suppression propre, et ne pas laisser le système dans un état incohérent, le composant doit être arrêté. Différentes politiques peuvent être appliquées. On peut supprimer simplement un composant, sans effectuer la moindre vérification structurelle. Il est toutefois possible de vérifier qu'aucun composant de la configuration ne dépende du composant à supprimer, et si tel n'est pas le cas, enlever les liaisons existantes. Ces politiques pourront être exprimées dans un langage de description.

**Suppression d'une fabrique** Pour le déchargement d'une fabrique, le composant est supprimé de la mémoire. Le contrôleur englobant la fabrique ne le référence plus et donc la fabrique ne peut plus servir à instancier de nouveaux composants. Cependant, toutes les interfaces partagées par les différentes instances sont créées au sein de la fabrique et pourraient donc être supprimées, inhilant par la même toutes les instances courantes.

**Suppression d'une instance** La suppression d'une seule instance d'un composant est possible, laissant libre l'exécution des autres instances.

## 4.8 Remplacement de composant

La relation d'intégrité minimale pour le remplacement d'un composant par un autre est que le nouveau composant doit être conforme à l'ancien. Cette conformité est essentiellement dépendante du type des composants mis en jeu, et donc du type de leurs interfaces. Avant tout remplacement, un composant doit être arrêté.

## 4.9 Contraintes d'intégrité

Enfin, la reconfiguration d'un système ne doit pas le corrompre. Autrement dit, certaines modifications du système ne doivent pas être autorisées si elles sont susceptibles d'être nuisibles. Cela suppose que l'on se dote d'un mécanisme de contraintes d'intégrité permettant de dynamiquement vérifier la cohérence de reconfigurations, et d'outils de sécurisation pour empêcher l'exécution de composants malveillants ou corrompus.

Une contrainte est une propriété ou une assertion sur le système ou une de ses parties, dont la violation rend le système incohérent. Pour assurer la cohérence de ses configurations, un système THINK permet la définition d'un certain nombre de contraintes d'intégrité. Des contraintes de cohérence structurelle, des contraintes de dépendance entre composants, des contraintes de sécurité et des invariants du système peuvent être considérés. Elles sont exprimées en utilisant des notations du langage de description d'architecture, que nous avons étendu. Ce langage permet de définir des types de contraintes que l'on souhaite maintenir à l'exécution.

Une première vérification statique est faite sur la complétude de la configuration, en établissant le graphe de contraintes initial. Ces contraintes sont maintenues pendant l'exécution, et sont vérifiées dynamiquement. Rien de systématique n'est actuellement prévu dans le système, mais nous pourrions imaginer un système pour effectuer des mises à jour de la configuration en cas de changements.

## 5 Conclusion

Nous avons présenté dans ce document un canevas logiciel pour la construction de noyaux de systèmes d'exploitation ouverts et adaptables. Nos travaux combinant les approches exo-noyau et canevas à base de composants, nous avons sélectionné THINK comme infrastructure de développement. Nous avons enrichi THINK par un canevas à composants, muni de capacités de reconfiguration dynamique et permettant des compositions récursives.

Les premières expériences montrent que ces nouvelles capacités n'engendrent qu'un faible accroissement de l'occupation mémoire [SCS02], et que le surcoût en temps d'exécution ne s'applique qu'aux composants qui les utilisent.

## Références

- [BCS02] E. Bruneton, T. Coupaye, and J.-B. Stefani. Recursive and dynamic software composition with sharing. In *Proceedings of the 7th*

*ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, Malaga (Spain), June 2002.

- [Ber94] L. Bergmans. The composition filters object model. In *RICOT symposium "Enabling Objects for Industry"*, June 1994.
- [BSP<sup>+</sup>95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain Resort, Colorado, December 1995.
- [eCO] eCOS. <http://sources.redhat.com/ecos/>.
- [EK95] Dawson R. Engler and M. Frans Kaashoek. Exterminate all operating system abstractions. In *Workshop on Hot Topics in Operating Systems*, May 1995.
- [FBB<sup>+</sup>97] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The flux OSKit : A substrate for kernel and language research. In *Symposium on Operating Systems Principles*, pages 38–51, 1997.
- [FSLM02] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. Think : A software framework for component-based operating system kernels. In *Usenix Annual Technical Conference*, Monterey (USA), June 2002.
- [HF98] J. Helander and A. Forin. MMLite : A highly componentized system architecture. In *Eight ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.
- [SCS02] A. Senart, O. Charra, and J.-B. Stefani. Developing dynamically reconfigurable operating system kernels with the think component architecture. In *Workshop on Engineering Context-aware Object-Oriented Systems and Environments*, Seattle, USA, November 2002. in association with OOPSLA'2002.