# Conception de noyaux de systèmes embarqués reconfigurables

Olivier Charra

UNIVERSITÉ JOSEPH FOURIER

# THÈSE

pour obtenir le grade de

Docteur de l'Université Joseph Fourier — Grenoble 1

École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique
spécialité Informatique, Systèmes et Communication

présentée et soutenue publiquement par

Olivier CHARRA

le 10 mai 2004

# CONCEPTION DE NOYAUX DE SYSTÈMES EMBARQUÉS RECONFIGURABLES

Thèse préparée au sein du laboratoire LSR–IMAG, projet SARDES

sous la direction du Pr. Sacha KRAKOWIAK

## JURY

| | | |
|---|---|---|
| Président : | Jacques | MOSSIERE |
| Rapporteurs : | Michel | RIVEILL |
| | Isabelle | PUAUT |
| Examinateur : | Jean-Bernard | STEFANI |

# Table des matières

# Chapitre 1

# Introduction

La perspective de l'émergence d'un environnement global du traitement de l'information dans lequel la plupart des objets physiques qui nous entourent seront équipés de processeurs, dotés de capacités de communication et interconnectés par le biais de réseaux divers, nous oblige à repenser les systèmes informatiques. Aux systèmes traditionnels, lourds, monolithiques et peu évolutifs, nous devons préférer les systèmes légers, flexibles et reconfigurables.

Cette thèse présente une architecture permettant la conception et le développement de noyaux de systèmes d'exploitations flexibles et reconfigurables à destination du monde de l'embarqué.

## 1.1 Contexte et objectifs de la thèse

Cette thèse s'inscrit dans un contexte nouveau et particulier, né de la rencontre de deux domaines de recherche auparavant disjoints, celui des architectures logicielles permettant l'adaptation, et celui des systèmes d'exploitation.

Les recherches en matière d'adaptation et de reconfiguration dynamiques se sont en effet longtemps cantonnés aux domaines des intergiciels et des applications. De même, la recherche en systèmes d'exploitation visait plus à en améliorer les performances brutes, mais restait trop souvent conservatrice des structures traditionnelles. Récemment, l'émergence de l'informatique embarquée et mobile est venue bouleverser la donne. La recherche en informatique a dû faire face à de nouvelles problématiques, liées notamment à l'hétérogénéité des plates-formes matérielles et aux contraintes nouvelles de leur utilisation. Les systèmes d'exploitation traditionnels se sont rapidement avérés trop lourds et rigides pour convenir à ce type d'appareils.

Les chercheurs ont alors compris les intérêts potentiels d'un rapprochement entre les intergiciels et les systèmes d'exploitation. Plusieurs travaux se sont

ainsi intéressés à la définition de systèmes d'exploitations reconfigurables et ont essayé d'exporter certains des concepts et des idées des intergiciels au cœur même du système. L'architecture de développement de systèmes d'exploitation THINK se place parmi eux.

THINK [1], développé par Jean-Philippe Fassino à France Télécom R&D, est un environnement de développement de noyaux de systèmes d'exploitation à base de composants permettant l'instanciation de noyaux ou de systèmes d'exploitation de toutes sortes. Sa philosophie originale, qui consiste à limiter au maximum les abstractions et les concepts imposés par défaut au développeur d'un système, le rend très flexible.

Si l'architecture THINK semble répondre de manière intéressante aux nouveaux besoins en matière de systèmes, elle prend encore une nouvelle dimension lorsqu'on songe à lui adjoindre des mécanismes de reconfiguration dynamique. Nous nous intéressons dans ce document à l'intégration d'un canevas de reconfiguration dynamique au sein de l'architecture THINK. Nous étudions également l'utilisation de Think comme base de développement de noyaux de systèmes d'exploitation à destinations d'architectures embarquées fortement contraintes.

## 1.2 Plan de la thèse

Le chapitre 2 se consacre à faire le point sur les systèmes d'exploitation actuels. Après un rapide historique permettant de comprendre l'évolution de l'architecture des systèmes d'exploitation, les principaux modèles de structuration sont présentés.

Le chapitre 3 établit les perspectives de recherche en matière de systèmes d'exploitation. Les nouvelles techniques de structuration de systèmes sont expliquées et sont illustrés par quelques exemples de systèmes d'exploitation de recherche les mettant en œuvre.

La présentation de l'architecture THINK fait l'objet du chapitre 4. On y aborde notamment la philosophie à la base du modèle, les concepts fondateurs de l'architecture et leur instanciation. Ce chapitre explique également en quoi et pourquoi l'architecture de THINK est perfectible et peut être améliorée.

Le chapitre 5 présente l'intégration d'un modèle de reconfiguration dynamique au sein même de Think. Une première implémentation de fonctions de reconfiguration avancées est également proposée.

L'utilisation de l'architecture présentée au chapitre 5 comme base de développement de noyaux de systèmes à destination de l'embarqué est étudiée dans

1. L'acronyme de THINK signifie " THINK Is Not a Kernel"

le chapitre 6. On y présente notamment une projection du modèle sur le domaine des architectures embarquées fortement contraintes, et le développement d'une bibliothèque de composants spécifiquement conçus pour l'embarqué.

Enfin, le chapitre 7 propose une évaluation des travaux présentés dans les chapitres précédents sous la forme du développement d'un noyau de système d'exploitation pour un environnement matériel fortement contraint : le Lego RCX.

# Chapitre 2

# Architecture des systèmes d'exploitation

## 2.1 Historique des systèmes d'exploitation

Les systèmes d'exploitation, au cours de leur courte histoire, ont connu divers bouleversements, la plupart du temps liés aux progrès technologiques accomplis dans le domaine de la miniaturisation électronique. Cette section présente un rapide historique des systèmes d'exploitation à travers les différentes étapes de l'évolution des matériels.

### 2.1.1 Les ordinateurs centraux

La notion de système d'exploitation est apparue pour la première fois en 1955, quand des chercheurs du centre de recherche de General Motors créent le premier programme de gestion de traitements différés pour IBM 701. Ce système était conçu pour faciliter la gestion des exécutions de programmes écrits sur cartes perforées.

Après l'invention du circuit intégré en 1958, IBM introduit en 1964 la famille d'ordinateurs System/360, avec pour objectif de proposer un modèle de programmation uniforme, quelque-soit la machine considérée (du "petit" ordinateur commercial au gros calculateur scientifique). Les ordinateurs de cette série disposaient tous d'un seul et même jeu d'instructions, et les logiciels développés pour l'un étaient théoriquement compatibles avec les autres, le système d'exploitation gérant lui-même toutes les opérations de bas niveau et faisant office d'interface entre la machine et les programmes. Toutefois, le développement et le maintien d'un seul et même système d'exploitation pour toute une famille d'ordinateurs s'avéra très difficile. Le système était constitué de millions de lignes d'assembleur et comportait un nombre très important de bogues. Mais malgré

ses inconvénients, ce système fut le premier à introduire la notion de multiprogrammation : pour la première fois, un ordinateur fut capable d'exécuter un programme pendant qu'un autre était en attente d'entrée / sortie.

La notion de multiprogrammation céda rapidement la place à la notion de temps partagé. Un système à temps partagé permet à plusieurs utilisateurs d'utiliser simultanément un seul et même ordinateur, chaque utilisateur se voyant attribuer un certain pourcentage des ressources en fonction de ses propres besoins et de la charge globale de l'ordinateur. Le rôle du système d'exploitation est alors de multiplexer les différentes tâches du système et de donner aux utilisateurs l'impression que chaque tâche s'exécute comme si elle était seule. Le premier système d'exploitation à temps partagé de l'histoire, CTSS, fut développé au MIT en 1962 au dessus d'une machine IBM 7094. Puis le MIT, Bell labs et General Electrics s'associèrent afin de concevoir et développer un système à temps partagé géant, suffisamment puissant pour répondre à l'ensemble des besoins de puissance de calcul de la ville de Boston. Ce projet donna naissance au système d'exploitation MULTICS (pour MULTiplexed Information and Computing Service). Si MULTICS ne fut jamais le système géant imaginé, les idées et les concepts autour desquels il est bâti ont eu une grande influence sur les systèmes d'exploitation développés ensuite.

### 2.1.2 Les mini-ordinateurs et les micro-ordinateurs

Les années 1970 virent l'émergence et le développement de mini-ordinateurs, moins performants mais également beaucoup moins chers que leurs aînés. Un chercheur de Bell Labs, Ken Thompson, rapidement rejoint par Brian Kernighan et Dennis Ritchie, décida de développer à partir de MULTICS un système d'exploitation pour mini-ordinateur. Le système fut ironiquement appelé UNIX (UNiplexed Information and Computing Service). Développé à l'origine sur un DEC PDP-7, il fut rapidement porté sur des PDP-11/20, PDP-11/45 et PDP-11/70. En parallèle de son développement, Thompson et Ritchie imaginèrent le langage C et développèrent un compilateur pour ce langage. UNIX fut ensuite totalement réécrit en C. Rapidement adoptés par la plupart des centres de recherches et des universités, UNIX et le langage C connurent le succès que l'on sait et devinrent des références pour nombre d'utilisateurs de par le monde.

L'évolution de l'informatique et des systèmes d'exploitation va connaître ensuite une mini-révolution grâce au microprocesseur. En 1971, un ingénieur d'Intel conçoit le premier microprocesseur du monde, qu'il nomme Intel 4004. Cadencé à 108KHz et gérant des mots de 4 bits, il intègre 2300 transistors gravés avec une précision de 10 microns (en comparaison, les processeurs actuels

approchent les 3GHz et comportent plusieurs dizaines de millions de transistors gravés avec une précision de 0,13 microns). Ce microprocesseur sera rapidement suivi par le 8008 (en 1972), le 8080 (en 1974) et les 8086/8088 (en 1978), ces derniers inaugurant la longue série des "x86". Un autre processeur connaîtra un grand succès : le 6800, créé en 1976 par Motorola.

L'arrivée des microprocesseurs sur le marché permet, à partir de 1980, l'émergence d'un nouveau type d'ordinateurs : les micro-ordinateurs (ou ordinateurs personnels). Beaucoup plus petits et beaucoup moins chers que leurs prédécesseurs, les micro-ordinateurs mettent la puissance de l'informatique à la portée de la plupart des entreprises et d'un nombre non négligeable de particuliers. Divers types d'ordinateurs vont voir le jour mais, rapidement, certaines architectures vont s'imposer : c'est le cas de l'IBM-PC et ses compatibles, basés sur des processeurs Intel, et des Apple, construits autour de processeurs Motorola. Côté systèmes d'exploitation, deux acteurs se partagent la plus grosse partie du marché : Microsoft avec son système MS-DOS, et UNIX.

### 2.1.3 Développement des réseaux

A partir du milieu des années 1980, les réseaux informatiques se développent fortement. L'utilisation sur une grande échelle des protocoles TCP/IP débute en 1983 avec la naissance d'Internet. La France est connectée à Internet à partir du 28 juillet 1988 grâce à l'INRIA de Sophia Antipolis. De nombreux micro-ordinateurs à travers le monde sont maintenant reliés entre eux et peuvent s'échanger des données. En 1993, le NCSA (National Center for Super computing Application) lance Mosaic, le premier navigateur Web (interprétant le langage HTTP 1.0). La croissance du Web est exponentielle : début 1995, le trafic HTTP dépasse le trafic FTP sur Internet.

Les systèmes d'exploitation prennent en compte ce changement en intégrant les protocoles de communication et en fournissant les outils permettant aux utilisateurs de se connecter à des machines distantes. En 1991 naît un système d'exploitation alternatif libre, très orienté réseau, qui fera rapidement parler de lui : Linux. Son inventeur, Linus Torvald, un étudiant finlandais en licence d'informatique de 21 ans, parle de lui ainsi dans le tout premier message qu'il adresse à la communauté informatique : "Je développe un système d'exploitation (libre, ouvert) alternatif à UNIX-Minix sur 486. C'est seulement un passe-temps et cela n'a pas la prétention de devenir un projet professionnel". Cela peut faire sourire quand l'on songe que son système (largement enrichi par la communauté par la suite) est utilisé aujourd'hui par plus de 12 millions de personnes de par le monde.

### 2.1.4 Informatique embarquée

Au cours des années 1990, le marché de l'informatique s'ouvre à de multiples domaines. La baisse du coût de production des processeurs permet de faire bénéficier la plupart des appareils domestiques d'une nouvelle puissance de calcul. Certains lave-vaisselles ou fours se voient ainsi dotés d'un processeur et de quelques kilo-bits de mémoire et offrent des fonctionnalités avancées aux utilisateurs. Le phénomène est flagrant dans le domaine de l'automobile, ou de plus en plus de tâches sont déléguées à des "cerveaux" électroniques (l'injection, le contrôle de trajectoire, les organes de sécurité, etc.). Une Renault Mégane construite en 2002 embarque ainsi plus de puissance de calcul que les premiers avions développés par Airbus en 1970.

L'informatique embarquée (car c'est comme cela qu'on qualifiera cette nouvelle tendance) a des conséquences sur les systèmes d'exploitation. Traditionnellement, ceux-ci étaient prévus pour contenter les besoins d'un maximum d'utilisateurs, quite à faire certains compromis sur les performances. Dans le contexte de l'embarqué, il faut au contraire privilégier les performances, parfois au détriment des fonctions et de la portabilité.

## 2.2 Fonctions d'un système d'exploitation

Quelle que soit la machine cible, les fonctions d'un système d'exploitation restent identiques. Un système doit d'une part fournir aux applications une interface abstraite de la machine physique au-dessus de laquelle il est implanté, et d'autre part gérer l'attribution des ressources physiques de la machine aux diverses applications.

### 2.2.1 Abstraction de la machine physique

Les ordinateurs actuels comportent un processeur, une certaine quantité de mémoire, une horloge et plusieurs périphériques d'entrée / sortie. Toutes ces entités sont tour à tour mises en œuvre lors de l'exécution d'une application, quelle que soit sa complexité (même le plus simple des "Hello World" fait déjà appel aux périphériques d'entrée / sortie). La mise en œuvre se fait à travers des appels à des instructions de très bas niveau, fournies par le processeur ou les contrôleurs des périphériques. Un appel prend la forme d'une séquence particulière de bits correspondant au code et aux différents paramètres de l'instruction.

Programmer une application à ce niveau d'abstraction constitue une tâche très ardue. Par exemple, la manipulation des périphériques d'entrée / sortie exige du développeur une connaissance extrêmement précise des spécifications

du matériel et de son fonctionnement. L'idée selon laquelle il était nécessaire de soulager le développeur d'applications d'un certain nombre de détails d'implémentation s'est rapidement imposée comme une évidence. Une solution fut trouvée : on intercala entre les applications et la machine physique une couche logicielle responsable de la gestion de bas niveau du système. Cette couche logicielle prit le nom de système d'exploitation.

Un système d'exploitation réalise une machine virtuelle, interface entre la machine physique et les applications. Il offre aux développeurs des primitives de manipulation des composants matériels du système à travers une interface standardisée et facile d'utilisation. Il se charge en outre d'opérations de bas niveau telles que la gestion des interruptions ou l'ordonnancement des processus.

### 2.2.2 Gestion des ressources

Une machine offre un certain nombre de ressources aux applications (processeur, mémoire, périphériques d'entrée / sortie, etc.). Les ressources sont partagées par toutes les applications. La gestion de la concurrence des accès aux ressources matérielles est une des charges dédiées au système d'exploitation. A cette fin, le système offre une interface d'accès standardisée à chacune des ressources matérielles. Tous les accès à une ressource s'effectuant au travers de cette interface sont multiplexés de façon cohérente.

Un système d'exploitation est ainsi capable d'allouer des ressources à une application. Il connaît, pour chaque application, les ressources qui lui sont attribuées et sait évaluer leur usage de façon quantitative. Enfin, il offre des mécanismes permettant de résoudre d'éventuelles requêtes d'allocation conflictuelles.

## 2.3 Modèles classiques de structuration d'un système

Il existe plusieurs modèles de structuration d'un système d'exploitation. Si tous se basent sur un même schéma organisationnel, ils diffèrent toutefois dans son application.

### 2.3.1 Organisation générale

Un système d'exploitation est généralement organisé en deux parties distinctes : la partie "noyau" et la partie "processus" (voir figure 2.1). Le noyau s'exécute directement au-dessus du matériel. Les processus, eux, s'exécutent au-dessus du noyau.

Le noyau fournit un ensemble très réduit de fonctionnalités. Pour des raisons de sécurité, il s'exécute dans son propre domaine de protection. Il s'exécute éga-

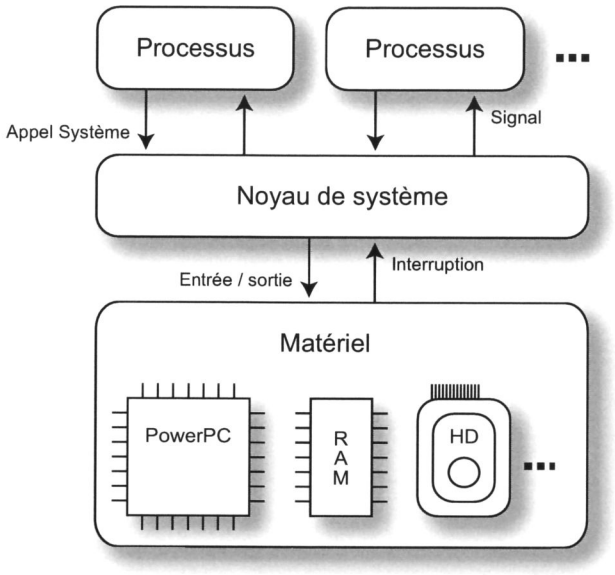


FIG. 2.1 – *Organisation générale d'un système d'exploitation*

lement en mode superviseur : il a donc accès à toutes les ressources du système. Un noyau communique avec le matériel grâce à des entrée / sortie. Le matériel appelle des fonctionnalités du noyau grâce à des interruptions.

La partie "processus" regroupe le reste des services du système d'exploitation et les applications. Les processus s'exécutent en mode utilisateur. Ils accèdent aux fonctionnalités du noyau par des appels système. A l'inverse, le noyau agit sur les applications grâce à des signaux. Seuls les appels système et les signaux ont la possibilité de traverser la frontière séparant les domaines de protection des processus et du noyau.

La séparation entre processus et noyau présente l'intérêt majeur d'augmenter le niveau de sécurité d'un système : un programme s'exécutant en mode utilisateur est isolé. Aucun programme de la partie "processus" ne peut donc perturber l'exécution du noyau. Toutefois, un tel niveau de sécurité se paye : les commutations de domaine de protection engendrent un surcoût important à l'exécution.

### 2.3.2 Domaines de protection unique

Un système à domaine de protection unique, comme son nom l'indique, ne possède qu'un domaine de protection. L'ensemble des applications et du système s'exécute au sein du noyau (qui n'a plus de noyau que le nom, voir figure 2.2). Le système joue un simple rôle d'interface entre les applications

et le matériel. L'absence de protection permet toutefois à une application de manipuler directement les ressources matérielles sans en référer au système.

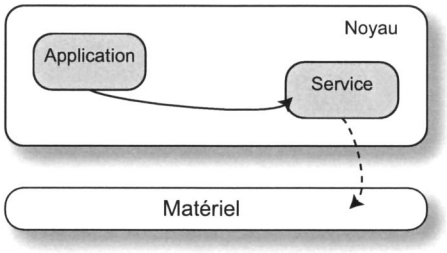


FIG. 2.2 – *Systèmes à domaine de protection unique*

Un système construit selon ce paradigme est très efficace, car aucun appel système ne vient ralentir son exécution. Toutefois, aucun mécanisme de sécurité ne peut préserver le noyau d'une défaillance d'une application ou d'un service du système.

Les systèmes à domaine de protection unique sont à réserver à de systèmes de petite taille, où les applications sont garanties dignes de confiance. C'est pourquoi de nombreux systèmes embarqués utilisent une architecture de ce type.

### 2.3.3 Noyaux monolithiques

Les systèmes à noyaux monolithique, à la différence des systèmes à domaine de protection unique, n'exécutent pas application et système dans le même domaine de protection (voir figure 2.3). Seul le système d'exploitation s'exécute en mode superviseur, l'ensemble des applications s'exécutant en mode utilisateur.

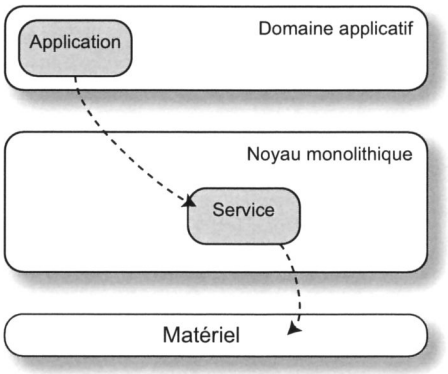


FIG. 2.3 – *Systèmes à noyau monolithique*

Dans un système à noyau monolithique, l'ensemble du système est contenu

dans le noyau, fonctionnalités de bas niveau comme services. Le noyau peut être vu comme une boîte noire, exportant les services du système sous forme d'interfaces. L'organisation interne du noyau n'est pas connue des applications.

Un noyau monolithique est efficace : tous les services du système s'exécutent en mode superviseur et manipulent directement les ressources du système. Le système est protégé des défaillances des applications. Toutefois, de par sa conception mono-bloc, un tel noyau est très peu évolutif. Sa faible structuration engendre en outre de fortes dépendances entre les différents services du système, ce qui rend les opérations de maintenance délicates.

Les noyaux UNIX AIX, Solaris et la plupart des noyaux Linux sont monolithiques, même s'ils se sont récemment vu architecturés de manière plus modulaire.

### 2.3.4 Micro-noyaux

Les systèmes à micro-noyaux représentent une tendance architecturale plus récente. Leur principe est simple : afin d'offrir un niveau de flexibilité important et de garantir une sécurité maximale au noyau du système, les services du système sont extraits du noyau et exportés au sein de leur propre domaine de protection, différent de celui des applications (voir figure 2.4). Chaque appel de service passe par le micro-noyau, qui redirige l'appel vers le service adéquat.

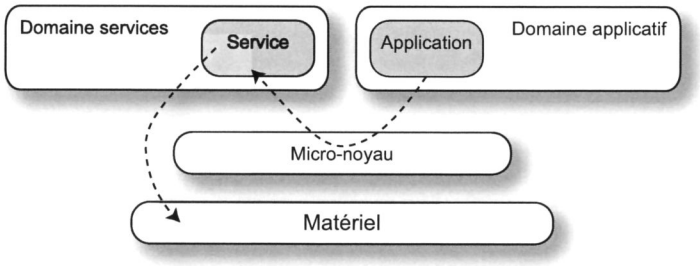


FIG. 2.4 – *Systèmes à micro-noyaux*

La modification du système et sa maintenance sont grandement facilitées par l'indépendance entre le noyau et les services, l'architecture garantissant qu'une modification d'un service ne perturbera pas l'exécution du noyau. Le noyau est de plus extrêmement sûr : aucune défaillance d'un service ne pourra l'affecter. En contrepartie, l'appel à un service système coûte deux commutations de domaines de protection, ce qui rend la performance du noyau très dépendante de celle du mécanisme de communication inter-domaines.

De nombreux systèmes récents sont basés sur des micro-noyaux. C'est no-

tamment le cas des systèmes Chorus [RAA+92] [RAA+88], Mach [ABB+86], L4 [?] et QNX [?].

### 2.3.5 Exo-noyaux

Une des plus récentes tendances dans le domaine de l'architecture de systèmes est constitué par les exo-noyaux. Un système à exo-noyau ne comporte plus au sein du noyau que les mécanismes permettant le multiplexage et la protection des ressources matérielles. L'ensemble des autres mécanismes système, y compris les mécanismes de contrôle des ressources, est transféré au niveau des processus applicatifs (voir figure 2.5).

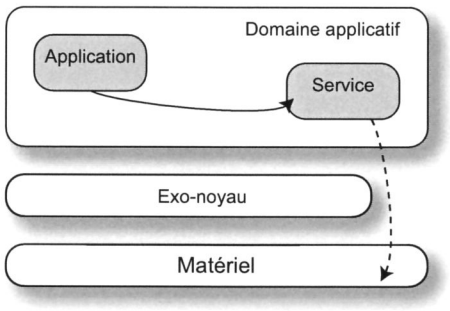


FIG. 2.5 – *Systèmes à exo-noyaux*

Un exo-noyau permet, en théorie, d'obtenir des systèmes flexibles et performants. De plus, les applications contrôlent elle-même les ressources, ce qui leur permet d'appliquer des politiques optimisées, basées sur des sémantiques applicatives. Toutefois, le développement d'un exo-noyau peut s'avérer relativement complexe.

Le premier (et à ce jour l'un des seuls) exo-noyau à avoir été produit est Aegis [EK95] [EKO95] (que l'on appelle communément Exokernel).

## 2.4 Bilan

Le domaine des systèmes d'exploitation constitue l'un des plus vieux thèmes de recherche en informatique. Des plus anciens systèmes à traitement différés des années 50 aux systèmes multi-tâches actuels, les systèmes d'exploitation ont subi de profondes évolutions de leurs structures et des améliorations de leurs fonctions, suivant la vague des évolutions technologiques.

Nous sommes aujourd'hui à la veille d'une nouvelle évolution des systèmes. En effet, le développement de l'informatique embarquée et mobile pose de nou-

velles contraintes aux systèmes d'exploitation. Celles-ci, ainsi que les solutions envisagées par la recherche, font l'objet du chapitre suivant.

# Chapitre 3

# Perspectives de recherche en systèmes d'exploitation

L'informatique traditionnelle est en train de vivre un grand bouleversement avec l'avènement de l'informatique embarquée. La plupart de nos appareils se voient maintenant dotés de capacités de calcul. Très bientôt, ils pourront communiquer entre eux.

Quelles sont les nouvelles contraintes auxquelles devront faire face les systèmes de demain ? Quelles seront les caractéristiques de ces systèmes ? Enfin, quelles solutions sont actuellement proposées par les systèmes d'exploitation de recherche ? C'est à ces trois questions que tente de répondre ce chapitre.

## 3.1 Introduction

Notre époque actuelle est le théâtre d'un nouveau bouleversement technologique. L'arrivée sur le marché de processeurs dotés de capacités de calcul toujours plus importantes pour des coûts de production toujours plus faibles engendre une explosion du nombre des matériels informatiques. L'ordinateur n'est plus seulement un outil de calcul réservé aux spécialistes, mais devient le cœur fonctionnel d'appareils que tout un chacun utilise quotidiennement, sans même se rendre compte de leur nature informatique. A cela s'ajoute le développement récent des réseaux sans fils, qui permet l'établissement de communications entre les différents objets même lorsque ceux-ci ne sont pas physiquement reliés. Le nouveau monde qui se dessine, où l'informatique sera très certainement omniprésente, réclame la conception de supports d'exécution adaptés.

Ce chapitre fait le point sur les systèmes embarqués et leurs contraintes spécifiques en section 3.2. La section 3.3 présente les caractéristiques souhaitées des systèmes de la prochaine génération. Les sections suivantes font ensuite le

point sur les dernières solutions actuellement envisagées par la recherche : les noyaux extensibles, les noyaux réflexifs et les noyaux à composants. Enfin, la section 3.7 dresse un bilan des différentes techniques de recherche présentées.

## 3.2 Les systèmes embarqués

De part leurs contraintes spécifiques de développement, les systèmes embarqués et temps-réel forment une classe particulière ayant des besoins très spécifiques en terme de système.

### 3.2.1 Caractéristiques principales

Une des propriétés les plus contraignantes des systèmes embarqué est leur non-homogénéité. En effet, il existe sur le marché une extraordinaire variété d'appareils. Ces appareils diffèrent principalement par leur puissance (les processeurs sont souvent très différents d'une machine à l'autre) et la variété de leurs périphériques d'entrée / sortie (certains possèdent un clavier mais d'autres de simples boutons, certains disposent d'un écran mais d'autres de simples diodes, certains sont connectés à un réseau mais d'autres sont totalement isolés, etc.).

Un système embarqué est intrinsèquement limité par sa capacité de calcul et sa mémoire. En effet, afin de limiter la consommation électrique de ces systèmes, les processeurs utilisés sont généralement relativement peu performants et la quantité de mémoire embarquée reste limitée. De plus, certains systèmes ne possèdent qu'une quantité très limitée de mémoire persistante (voire, pour quelques-uns, aucune).

La connexion au réseau d'un système embarqué, quand elle existe, est souvent de qualité incertaine (cela est d'autant plus vrai dans le cas d'un réseau sans-fil). De plus, la plupart des périphériques ne sont pas connectés en permanence.

Un système embarqué se caractérise également par son alimentation limitée en énergie. Une grande partie des appareils sont alimentés par une batterie d'une durée de vie limitée à quelques heures.

Enfin, certains systèmes possèdent des contraintes en matière de temps. En effet, les logiciels critiques (tels que ceux que l'on trouve dans les automobiles, les avions, les téléphones, etc.) exigent l'exécution de parties de leur code dans un temps borné. Ce type de système est appelé système temps-réel. Un système est donc dit "temps-réel" si sa validité ne dépend pas seulement du résultat de son exécution mais également du temps de production de ce résultat. Parmi les systèmes temps-réels, on distingue ceux pour qui une contrainte de temps doit

être vérifiée en moyenne sur l'ensemble des exécutions, et ceux pour qui une contrainte de temps doit être vérifiée en toute situation.

### 3.2.2 Principes et techniques de construction traditionnels

Actuellement, lorsqu'un utilisateur souhaite disposer d'un système d'exploitation pour un matériel embarqué, plusieurs alternatives s'offrent à lui.

La première solution est de récupérer un système d'exploitation libre de droit (par exemple Linux) et de l'adapter à l'architecture du matériel qu'il désire faire fonctionner. Cette opération reste délicate : supprimer les fonctionnalités inutiles d'un système existant peut s'avérer très difficile. De plus, une telle démarche résulte souvent en un système lourd, peu efficace, difficile à maintenir et parfois instable.

Si l'architecture matérielle que l'utilisateur possède est suffisamment standard, un système d'exploitation existe peut-être dans le commerce. Cette solution a l'avantage de la facilité, toutefois elle comporte ses propres inconvénients. Le premier est financier; un système commercial pour une architecture peu répandue risque d'être cher. Le deuxième est technique; un système commercial inclut souvent un ensemble de fonctionnalités dont l'utilisateur moyen n'a aucun besoin, mais qui ont pour effet de diminuer les performances et d'augmenter l'occupation mémoire à l'exécution.

La dernière solution, envisageable dans toute situation, est de développer son propre système. Cela reste la meilleure solution pour obtenir un système ayant exactement les caractéristiques souhaitées. Toutefois, cela aboutit en général à des systèmes monolithiques, peu évolutifs, difficilement réutilisables et incompatibles entre eux. Ce processus reste en outre réservé aux développeurs expérimentés, ayant une connaissance complète de l'architecture matérielle cible et disposant de suffisamment de temps pour concevoir, implémenter, tester et mettre au point le système.

### 3.2.3 Besoins en termes de systèmes d'exploitation

Si de nombreux systèmes d'exploitation pour l'embarqué existent, ceux-ci sont souvent conçus pour une gamme d'architectures matérielles données et ne peuvent que difficilement s'adapter à des types de matériel différents. De plus, ils souffrent généralement d'un "embonpoint" dû à l'ajout de fonctionnalités censées leur permettre de couvrir une gamme toujours plus large d'appareils, mais inutilisées dans la majorité des cas. Enfin, très peu s'avèrent capable de prendre en compte de nouvelles fonctionnalités, si ce n'est au prix d'une mise à jour complète du système par écrasement de la mémoire.

Il apparaît ainsi nécessaire de disposer d'outils facilitant et automatisant au maximum la conception et l'implémentation de noyaux de systèmes d'exploitation à destination d'architectures matérielles embarquées. Les noyaux produits devraient être parfaitement adaptés à l'architecture matérielle qu'ils sont censés gérer, et offrir si possible des mécanismes leur permettant d'être enrichis fonctionnellement et adaptés à d'éventuelles nouvelles conditions d'exécution.

## 3.3 Caractéristiques des systèmes de demain

Cette section présente les caractéristiques souhaitées des systèmes d'exploitation de la prochaine génération.

### 3.3.1 Systèmes spécialisables

Il y a quelques années le marché informatique tendait vers une certaine homogénéisation des architectures matérielles. Aujourd'hui au contraire, la diversité des spécifications matérielles des nouvelles plates-formes informatiques rend le développement d'un système d'exploitation universel très difficile (voir impossible).

Si la définition d'un système générique semble compromise, il semble au contraire intéressant de faire porter l'effort de recherche sur la conception de nouveaux outils facilitant le développement de systèmes d'exploitation spécialisés. Ainsi, en identifiant les invariants structurels des différentes classes de système, nous serons à même de définir un modèle architectural fondamental servant de base à la conception de tout type de système.

### 3.3.2 Systèmes configurables

Les politiques de contrôle des ressources des systèmes d'exploitation actuels sont conçues pour offrir les meilleures performances moyennes sur une gamme d'applications donnée. Or, pour une application précise, il existe bien souvent une autre politique de contrôle des ressources obtenant de meilleurs résultats. Par exemple, la connaissance de paramètres sémantiques liés à l'application elle-même permet souvent d'optimiser significativement les algorithmes d'allocation de ressources.

Un système configurable est un système qui permet, lorsque l'on connaît les conditions de son utilisation, de modifier tout ou partie de sa composition à des fins d'optimisation. Le niveau de configurabilité dépend directement de l'étendue des modifications possibles.

### 3.3.3 Systèmes adaptables

La plupart des systèmes d'exploitation actuels ne sont pas (ou sont très peu) modifiables dynamiquement. Pourtant, avec le développement de l'informatique mobile en particulier, l'environnement d'exécution d'un système est susceptible d'évoluer de façon importante durant son exécution. Ces évolutions peuvent nécessiter des changements en profondeur du système. Par exemple, le déplacement d'un objet mobile au sein d'un environnement non sécurisé peut nécessiter la mise en place de mécanismes de protection au niveau du système d'exploitation de l'objet.

Un système dynamiquement adaptable est un système capable de détecter des modifications de son environnement d'exécution et d'y répondre par une modification de ses mécanismes et structures internes.

## 3.4 Noyaux extensibles

Certains projets de recherche se sont intéressés à l'adaptation d'un noyau de système par son extension dynamique. La plupart de ces projets ont concentré leurs efforts sur les mécanismes permettant de garantir qu'une extension d'un noyau ne remette pas en cause sa cohérence globale.

### 3.4.1 Principes

L'extension dynamique d'un noyau de système consiste à charger dynamiquement certains modules logiciels au sein même du noyau afin d'augmenter ses possibilités fonctionnelles. L'extension d'un noyau peut être dangereuse : l'ensemble du noyau s'exécutant dans le même domaine de protection, tout dysfonctionnement du nouveau module peut entraîner la défaillance du noyau lui-même. Pour se prémunir contre les risques d'une telle opération, plusieurs techniques ont été imaginées.

Une première technique consiste à trouver un moyen logiciel d'isoler l'exécution d'une extension. Lorsqu'une défaillance se produit, elle ne se répercute ainsi pas sur l'ensemble du système.

La deuxième technique couramment employée consiste à utiliser un langage sûr pour l'implémentation des extensions. Les langages sûrs (comme, par exemple, Modula-3 [?]) offrent tous les mécanismes de protection nécessaires permettant de faire face à une éventuelle défaillance d'un programme.

### 3.4.2 Points forts de l'approche

L'extensibilité dynamique d'un noyau permet l'ajout de fonctionnalités sans recompilation ni redémarrage. Après ajout, la fonction est immédiatement prise en compte par le noyau et disponible pour les applications. Les mécanismes de sécurisation des extensions assurent de plus que l'ajout d'une extension ne viendra pas corrompre l'exécution du système.

### 3.4.3 Points faibles de l'approche

Les noyaux extensibles n'ont qu'une approche partielle de la reconfiguration : ils ne permettent que l'ajout de fonctionnalités au noyau. La suppression d'une extension est en effet une opération beaucoup plus complexe. Avant de supprimer une extension, il est nécessaire en effet de s'assurer que l'extension n'est pas en cours d'utilisation et ne sera plus utilisée à l'avenir. Cette opération nécessite des connaissances complètes sur le système, que les systèmes extensibles actuels ne proposent pas.

Un autre point négatif concerne les performances des noyaux extensibles. En effet, l'ajout d'une extension s'accompagne d'une baisse sensible des performances du système. Cette baisse de performances est à attribuer aux mécanismes de sécurisation des extensions.

Pour résumer, les noyaux extensibles ne constituent qu'une amélioration fonctionnelle des noyaux existants. Si les techniques d'extension et de sécurisation des extensions proposées sont tout à fait intéressantes et réutilisables, elles ne constituent pas pour autant une vraie réflexion architecturale sur les systèmes d'exploitation de demain.

### 3.4.4 Exemples de noyaux extensibles

Parmi les différents noyaux extensibles ayant vu le jour, certains ont acquis une certaine notoriété. En voici une description.

#### 3.4.4.1 Vino

Vino est un système d'exploitation extensible développé par l'Université de Harvard [SESS96]. Il permet l'extension du système par le chargement dynamiques d'extensions (une extension étant un module C++).

Vino utilise un mécanisme d'isolation logicielle pour assurer la protection du noyau contre les extensions malveillantes. De plus, chaque appel à une extension du noyau s'exécute au sein d'une transaction : en cas de panne, le système peut faire marche arrière et retourner dans un état cohérent.

Vino assure une sécurité maximale lors de l'extension du système, mais les mécanismes de sécurisation employés ont un surcoût très important à l'exécution. Le modèle de sécurisation de Vino est ainsi peu généralisable.

#### 3.4.4.2 Spin

Spin est un micro-noyau extensible développé à l'Université de Washington [BSP+95]. Spin permet l'extension du noyau par le téléchargement et la liaison dynamique d'extensions. Une extension peut ajouter de nouveaux services au noyau, remplacer une politique existante ou simplement transférer une fonctionnalité applicative au sein de l'espace d'adressage du noyau lui-même.

Spin se protège des risques d'une extension boguée ou malveillante grâce à l'utilisation du langage Modula-3 et de ses mécanismes de sûreté. Bien sûr, l'utilisation de ce langage s'accompagne d'une dégradation des performances : les mécanismes de sûreté du langage se traduisent à l'exécution par un certain nombre de vérifications coûteuses. Ces vérifications peuvent être désactivées à la compilation, mais on perd alors les propriétés du langage.

## 3.5 Noyaux réflexifs

La réflexivité est une technique issue de recherches dans le domaine de l'intelligence artificielle. Théorisée très tôt par Smith [Smi82] et Maes [MN88], elle fut plus longue à déboucher sur des applications concrètes. Aujourd'hui toutefois, de nombreux systèmes utilisent des techniques réflexives pour offrir des primitives de reconfiguration du système.

### 3.5.1 Principes

Un système est dit réflexif s'il est capable d'abstraire une représentation de lui-même et de s'en servir pour effectuer des modifications sur ses propres structures. Ainsi, peuvent être considérés comme réflexifs les systèmes offrant une représentation de leur structure interne, des outils permettant de manipuler cette représentation, et des procédures internes permettant de refléter les modifications de la représentation sur le système lui-même.

L'architecture traditionnelle d'un système réflexif comporte deux niveaux : le niveau de base et le méta-niveau (voir figure 3.1). Le système se situe au niveau de base. Le méta-niveau est constitué par la représentation du système. Les deux niveaux sont les reflets l'un de l'autre, d'où le terme "réflexivité". Tout changement sur l'un des niveaux se répercute (se reflète) immédiatement sur l'autre niveau grâce à deux opérations : la réification et la réflexion. La réification

désigne l'opération rendant explicite un concept du système au niveau de sa représentation. La réflexion désigne l'opération répercutant une modification de la représentation sur le système lui-même. Réification et réflexion sont duales l'une de l'autre.

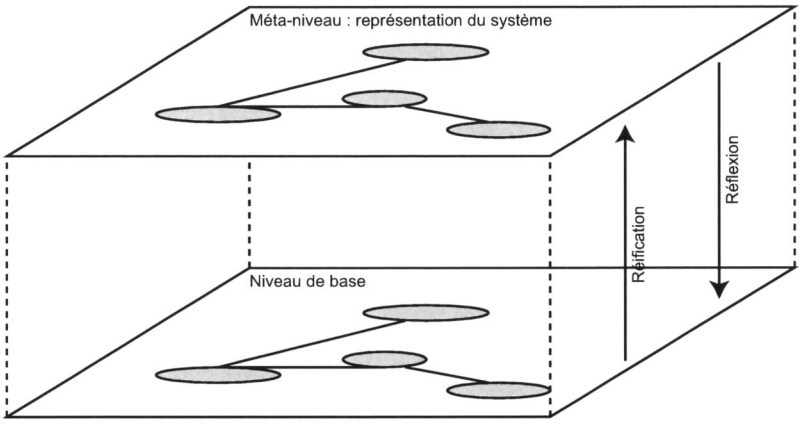


FIG. 3.1 – *Architecture d'un système réflexif*

L'implémentation des opérations de réification, de réflexion, du niveau de base et du méta-niveau sont dépendantes du système. En particulier, le niveau d'abstraction de la représentation du système peut varier, afin de cacher certains détails d'implémentation que l'on ne souhaite pas rendre modifiable.

### 3.5.2 Points forts de l'approche

Un système réflexif, en maintenant une représentation de ses structures internes, offre un support intéressant pour des fonctions de reconfiguration complexes. Puisque le système maîtrise à la fois le niveau d'abstraction de la représentation et les fonctions de réification et de réflexion, il garde le contrôle des opérations et peut s'assurer ainsi de leur validité.

La réflexivité offre ainsi une vision architecturale intéressante où le découplage du système et de sa représentation permettent d'opérer des reconfigurations sur une abstraction du système et non sur le système lui-même.

### 3.5.3 Points faibles de l'approche

Un noyau réflexif présente un surcoût mémoire non négligeable à l'exécution, principalement du fait du maintient de la représentation en mémoire du système. Bien sûr, ce coût peut être réduit en jouant sur les fonctions d'abstraction du système, mais plus la représentation du système sera partielle, moins les opérations de reconfiguration seront puissantes.

La réflexivité est en outre une vision architecturale théorique difficile à mettre en œuvre si l'on ne dispose pas d'un support d'exécution adéquat. Il est par exemple très complexe d'extraire une représentation abstraite d'un noyau de système composés de simples modules C interdépendants. La mise en œuvre d'une mécanique réflexive implique donc l'utilisation de techniques de structuration de code avancées.

### 3.5.4 Exemples de noyaux réflexifs

Cette section donne une brève description de quelques systèmes d'exploitation de recherche se définissant comme réflexifs.

#### 3.5.4.1 Apertos

Apertos [Yok92] [ILY95] est un système d'exploitation réflexif orienté objet. Dans Apertos chaque objet possède un environnement d'exécution qui lui est propre, appelé méta-espace ("meta-space"). Les aspects comportementaux d'un objet sont définis à l'intérieur du méta-espace qui lui est associé. Pour changer le comportement d'un objet, il suffit de le faire migrer vers un autre méta-espace. Un objet peut par exemple migrer vers un méta-espace sécurisé s'il évolue dans un environnement potentiellement hostile, ou migrer vers un méta-espace offrant certaines fonctions de débogage lors de la mise au point de l'application.

L'accès au méta-niveau se fait par l'intermédiaire de réflecteurs ("reflectors"). Tous les réflecteurs font partie d'une même hiérarchie dont la racine est `mCommon`. Le réflecteur `mDriveObject` fournit par exemple des méta-opérations de gestion de périphériques, `mBuiltinClass` fournit des primitives de gestion de classes, `mRealTime` fournit des primitives pour les systèmes temps réel, etc... Le développeur peut également définir son propre réflecteur comme sous-classe d'un réflecteur existant.

L'architecture du système Apertos est basée sur un méta-objet particulier nommé `MetaCore`. `MetaCore` est un méta-objet terminal qui ne possède pas de méta-espace propre. Son rôle est similaire à celui d'un micro-noyau dans les systèmes d'exploitation actuels.

Apertos représente la première tentative d'utilisation de la réflexivité dans un système d'exploitation.

#### 3.5.4.2 Merlin

Merlin est un système d'exploitation réflexif développé à l'université de Sao Paulo [?]. Merlin reprend une grande partie de l'architecture d'Apertos : le noyau est constitué d'objets évoluant dans des méta-espaces. Merlin est toutefois plus

flexible qu'Apertos grâce à l'utilisation du langage Self [?], un langage développé par Sun et s'apparentant à SmallTalk. L'ensemble du système est développé dans ce langage, et le système lui-même n'est qu'une mini-machine virtuelle Self. Le système est ainsi beaucoup plus souple et s'adapte à de multiples plates-formes : seule la machine virtuelle est dépendante des caractéristiques techniques du matériel.

## 3.6 Noyaux à composants

Quelques travaux de recherche récents en matière de systèmes d'exploitation s'intéressent à la recherche de nouveaux modèles structurels favorisant la flexibilité et l'ouverture du système. Parmi les différents modèles proposés, l'un semble peu à peu s'imposer : le modèle de structuration par composants.

Les modèles de structuration par composants sont issus des intergiciels. En effet, de nombreuses plates-formes d'exécution s'appuient sur un modèle à composants pour offrir plus de souplesse aux applications (comme la paramétrisation et la reconfiguration dynamique de la plate-forme par exemple). C'est ainsi que sont par exemple apparus les modèles CCM (Corba Component Model) [CCM02] et RM-ODP [X.995]. L'idée d'appliquer de tels modèles dans le domaine des systèmes d'exploitation n'est venue que bien plus tard.

### 3.6.1 Principes

Un système d'exploitation à composants est un système dans lequel les différentes entités logicielles du noyau et des services système sont organisées en modules distincts, appelés composants. Chaque composant est caractérisé par le service qu'il offre, et deux composants offrant le même service sont a priori interchangeables. Les composants sont reliés entre eux par des liaisons qui leur permettent de communiquer. La figure 3.2 présente un exemple de système conçu à l'aide de composants.

Les modèles de composition se distinguent selon deux critères : la granularité d'un composant et les possibilités de composition du modèle.

#### 3.6.1.1 Granularité d'un composant

Les modèles de composition diffèrent d'abord selon la granularité qu'ils associent à la notion de composant. Certains systèmes optent pour des modèles à gros grain, où un composant représente un service système ou une application dans son ensemble. Pour d'autres modèles, un composant est une unité logicielle indépendante très réduite, comme un pilote de périphérique ou une sous-partie

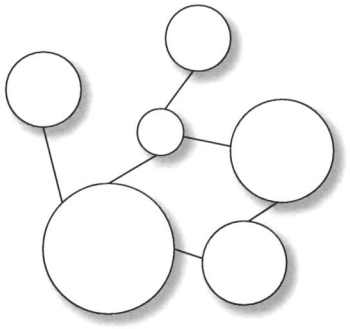

FIG. 3.2 – *Un exemple de système à composants*

d'une couche protocolaire. Enfin, certains systèmes n'associent aucun grain particulier à la notion de composant.

Les possibilités de configuration d'un système sont augmentées lorsque le système opte pour des composants à grain fin. Toutefois, les coûts liés au modèle de composition augmentent également avec un grain fin. Ainsi, lors du choix de la granularité d'un système à composants, il est nécessaire de trouver le juste compromis entre le niveau de configurabilité du système et ses performances.

#### 3.6.1.2 Possibilités de composition

Les systèmes à composants diffèrent également selon les différentes possibilités de composition offertes par leur modèle. Deux types de modèle existent, les modèles à plat et les modèles hiérarchiques.

Dans un modèle de composition à plat, un composant n'offre que des interfaces externes, c'est à dire à destination de son environnement (voir figure 3.3). La seule relation pouvant être établie entre deux composants est une liaison entre leurs interfaces externes. Une composition peut donc être représenté par un graphe dont les nœuds sont les composants et les arcs sont les liaisons.

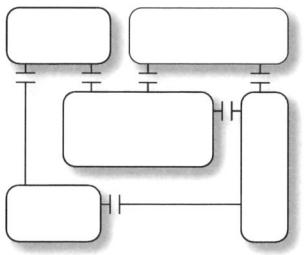

FIG. 3.3 – *Modèle de composition à plat*

Le modèle de composition hiérarchique ajoute au modèle à plat la relation d'appartenance. Dans un modèle hiérarchique, un composant peut appartenir à un autre composant d'ordre supérieur. Les interactions entre le composant englobant (qu'on appelle composite) et son sous-composant passent par des interfaces internes, c'est à dire dirigées vers l'intérieur du composant (voir figure 3.4). De plus, selon les modèles, le composite peut factoriser un ensemble de propriétés et de comportements d'administration pour l'ensemble de ses sous-composants.

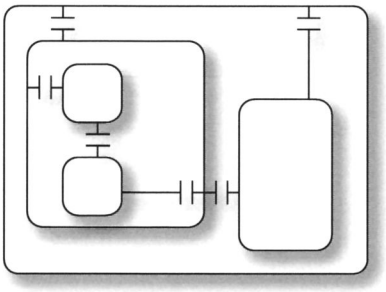

FIG. 3.4 – *Modèle de composition hiérarchique*

L'utilisation d'un modèle hiérarchique aboutit à une meilleure structuration du système qu'un modèle à plat. Elle facilite également la mise en place de mécanismes tels que la gestion de la sécurité grâce à la possibilité laissée au développeur de "cacher" l'implémentation de parties du système en les englobant au sein d'un composite. Toutefois, la composition hiérarchique peut être lourde et coûteuse lorsqu'elle est mal maîtrisée, chaque nouveau niveau de hiérarchie augmentant le coût global du modèle de composition. Son utilisation doit donc se faire avec prudence, en particulier dans des environnements matériels fortement contraints.

### 3.6.2 Points forts de l'approche

L'approche par composants présente de nombreux avantages par rapport aux modèles de structuration classiques des systèmes. En voici une liste non exhaustive.

Le paradigme de structuration par composants permet en premier lieu de rendre indépendants les développements des différents modules d'un système. En effet, l'établissement des liaisons entre composants n'est plus effectuée par l'étape d'édition de liens mais directement à l'exécution, au moment de l'initialisation du système. Chaque composant est donc développé et testé de manière indépendante avant d'être intégré au système. La productivité est fortement

augmentée par l'utilisation d'un tel modèle de structuration, de même que le taux de réutilisation de code.

D'autre part, la notion de composant permet de détacher explicitement la description du service que rend un comportement de son implémentation grâce à la notion d'interface. Il est ainsi possible d'associer plusieurs implémentations à une même interface. Dans une situation donnée, l'implémentation la plus avantageuse sera choisie, cela de façon totalement transparente vis-à-vis du reste du système qui, lui, ne perçoit que l'interface d'un composant.

La séparation du code en unités fonctionnelles distinctes permet en outre d'expliciter les dépendances entre modules : une dépendance entre deux composants correspond à une liaison. L'implémentation d'opérations de reconfiguration dynamique d'un système est ainsi facilitée.

Enfin, la structure de composant offre le support idéal pour l'intégration de fonctions de contrôle au dessus du code fonctionnel. En effet, un composant peut être imaginé comme une capsule de contrôle entourant un module fonctionnel. Toute interaction entre le module fonctionnel et son environnement traverse les limites de la capsule et peut a priori être interceptée. Ainsi, des techniques de contrôle avancées, qui nécessitent souvent des capacités d'observation du comportement du code fonctionnel (c'est par exemple le cas de la réflexivité) peuvent être implémentées à l'aide de composants.

### 3.6.3 Points faibles de l'approche

Si un système d'exploitation structuré sous forme de composants présente de nombreux atouts, l'un de ses principaux défauts réside le surcoût induit par l'encapsulation d'unités logicielles au sein de composants. Si ce surcoût est souvent négligeable dans le domaine des intergiciels (où les délais de communication sur le réseau lui sont très largement supérieurs), il n'en est rien pour un système d'exploitation où tout coût supplémentaire, même minime dans l'absolu, s'avère proportionnellement très important.

D'autre part, un système à composants exhibe sa structure interne et offre certaines possibilités avancées de reconfiguration qui peuvent s'avérer dangereuses si elles sont mal exploitées, soit par un processus malveillant, soit par un processus erroné.

Un système d'exploitation construit selon le paradigme de structuration à composants doit veiller à apporter des solutions concrètes aux deux problèmes majeurs auxquels il peut être soumis : les performances et la sécurité.

### 3.6.4 Exemples de noyaux à composants

Les noyaux se basant sur un paradigme de structuration par composants sont de plus en plus nombreux. Si certains se contentent d'un modèle de composition statique, d'autres en revanche proposent des mécanismes de configuration dynamique complexes.

#### 3.6.4.1 OSKit

OSKit est une architecture de développement de noyaux à la carte développée à l'Université d'Utah [FBH+97]. OSKit fournit un ensemble de 34 bibliothèques de composants systèmes (la plupart des composants sont extraits de Linux) permettant de composer un noyau selon ses besoins particuliers .. La composition s'effectue grâce à une architecture de composition nommée Knit [?].

OSKit est une architecture de composition de noyaux tout à fait intéressante. Le travail de décomposition qui a été opéré pour extraire de Linux des composants indépendants est par exemple fort appréciable. Toutefois, OSKit ne permet que la composition statique d'un noyau, aucun mécanisme de configuration dynamique n'étant prévu.

#### 3.6.4.2 eCOS

eCOS (embedded Cygnus Operating System) est un système d'exploitation libre et configurable pour architectures matérielles embarquées proposé par RedHat (le système fut développé par Cygnus, qui fut ensuite racheté par RedHat) [?].

L'atout majeur du système eCOS réside dans son mécanisme de configuration. Ce mécanisme de configuration permet au développeur d'une application d'imposer des contraintes sur les composants exécutifs du noyau, que ce soit en terme de fonctionnalités ou en terme d'implémentations. L'empreinte d'un système eCOS est ainsi largement réduite, car toutes les fonctionnalités inutiles sont supprimées du noyau. L'architecture à composants d'un noyau eCOS permet également aux développeurs d'intégrer leurs propres composants au noyau.

Le système eCOS s'adapte à une large gamme d'architectures matérielles grâce à l'interposition d'une couche d'abstraction entre le matériel et le système. Cette couche, nommée HAL (Hardware Abstraction Layer, en référence au fameux ordinateur du film de Stanley Kubrick "2001, odyssée de l'espace"), masque les spécificités du processeur et de la plate-forme matérielle cible.

#### 3.6.4.3 MMLite

MMLite est un système d'exploitation à composants proposé par Microsoft [HF98] basé sur le modèle de composition de microsoft COM [?]. MMLite fournit un ensemble de composants qui, assemblés dynamiquement, forment un système totalement fonctionnel. Il est en outre possible de remplacer n'importe quel composant du système à l'exécution. Le mécanisme de remplacement de composants fournit par MMLite, nommé "Mutator", permet de ne remplacer que la partie normalement immuable d'un composant (c'est à dire l'implémentation des méthodes), tout en conservant l'état du composant remplacé. Toutefois, MMLite laisse à la charge du développeur du système le soin de s'assurer de la validité des opérations de remplacement de composant.

#### 3.6.4.4 PeCoS

PeCoS (Pervasive Component Systems) [?] est un projet européen dont le but est de permettre le développement de systèmes embarqués à composants. Ce projet a aboutit à la conception d'un environnement supportant la spécification, la composition, la vérification de configuration et le déploiement de systèmes embarqués à l'aide de composants.

Dans le modèle de composition définit par PeCoS, un composant est un module logiciel. Un composant peut être actif (lorsqu'il modélise un processus du système), passif (lorsque son exécution est totalement synchrone) ou réactif (lorsque son exécution est liée à la réception d'un événement). Un langage de description permet de spécifier, pour un composant, ses interfaces fonctionnelles, ses propriétés non-fonctionnelles et ses connexions avec les autres composants du système. Un outil graphique permet ensuite de composer le système et de générer le noyau correspondant.

PeCos permet uniquement la composition statique d'un système. Après son chargement, aucun mécanisme de reconfiguration n'autorise l'évolution du noyau.

#### 3.6.4.5 THINK

Think est une architecture de développement de noyaux de systèmes développé par Jean-Philippe Fassino à France Télécom R&D [FS01a] [FSLM02].

L'ensemble d'un noyau THINK (ressources logicielles comme ressources matérielles) est structuré sous forme de composants. Les composants communiquent entre eux au travers de liaisons flexibles. Une liaison est également un composant, et peut prendre diverses formes (de la plus simple association entre un nom et une adresse mémoire à la composition complexe de plusieurs composants). THINK adopte en outre la philosophie des exo-noyaux : le noyau minimal

d'un système THINK ne contient que des interfaces réifiant les ressources matérielles de la plate-forme cible. Toute abstraction de plus haut niveau est ainsi optionnelle.

Si aujourd'hui THINK ne dispose pas de mécanismes de reconfiguration dynamique, son architecture à composants combinée à sa philosophie exo-noyau offre un cadre idéal au développement de telles fonctions à moindre coût.

## 3.7 Bilan

L'extensibilité dynamique d'un noyau fut la première approche adoptée pour résoudre les problèmes de l'adaptation d'un système à son environnement. Le principal intérêt de ces architectures fut de proposer des solutions à la sécurisation d'un noyau évolutif. Mais, malgré l'intérêt évident des techniques proposées, les noyaux extensibles ne présentent que peu d'éléments architecturaux fondamentaux pouvant servir de base à la définition d'un noyau de système reconfigurable.

Les noyaux réflexifs représentent une réflexion plus poussée sur l'architecture d'un noyau reconfigurable. La réflexivité propose en effet une architecture générale d'un système reconfigurable et distingue les différentes entités impliquées dans une reconfiguration. L'architecture proposée est toutefois très générale et ne dit rien sur la structuration même du système.

Le paradigme de structuration d'un noyau par composants représente l'évolution la plus intéressante. En effet, plutôt que de proposer des nouvelles techniques de reconfiguration, ce paradigme propose une nouvelle structuration des systèmes d'exploitation censée faciliter le développement de telles techniques. Des noyaux extensibles, de même que des noyaux réflexifs, peuvent être construits sur une base de noyaux à composants.

Parmi les différents systèmes d'exploitation à composants, THINK paraît l'un des plus prometteurs. Contrairement aux autres systèmes, il échappe en effet à la tentation de se définir comme un noyau à part entière mais se définit plutôt comme une architecture de développement de noyaux. Il combine les avantages de deux visions architecturales : la philosophie des exo-noyaux et la structuration par composants. La philosophie des exo-noyaux permet à un système Think d'être spécialisé en vue de son utilisation dans un cadre logiciel et matériel précis. Sa structuration par composants offre le support d'exécution idéal pour l'intégration de fonctions de reconfiguration avancées.

L'utilisation de THINK comme base logicielle pour un environnement de construction de systèmes d'exploitation embarqués reconfigurables nous a ainsi semblé naturelle.

# Chapitre 4

# THINK : un environnement de développement de systèmes d'exploitation

Ce chapitre présente THINK, un environnement de développements de systèmes d'exploitation. THINK, développé à France Télécom R&D par Jean-Philippe Fassino, permet l'instanciation de noyaux de systèmes de tous types et pour n'importe quelles architectures matérielles. [Fas01] et [FS01a] présentent une description plus détaillée de l'architecture.

## 4.1 Objectifs

L'espace de conception des systèmes d'exploitation est large et regroupe de nombreux types d'architectures différentes. L'objectif de l'environnement THINK est de fournir une vision unifiée du modèle de programmation et de mettre à disposition des développeurs les outils et structures permettant de constuire tout système faisant partie de ce spectre de développement. THINK doit en cela posséder les caractéristiques suivantes :

**flexibilité :** La flexibilité d'un système se mesure à sa capacité à s'accomoder, de façon dynamique ou non, aux changements de son environnement d'exécution (matériel ou logiciel).

**ouverture :** Un système est dit ouvert si toutes ses interfaces de programmation sont explicites et accessibles. Un système ouvert autorise la modification ou le remplacement de ses fonctions.

**modularité :** La modularité d'un système caractérise sa capacité à être décomposé en sous-systèmes d'ordre inférieur. En modularisant un système, on

simplifie sa conception et son évolution, et on améliore le niveau de réutilisation de code lors du portage du système sur différentes architectures matérielles.

**administrabilité :** Un système est dit administrable s'il dispose d'une instance de contrôle permettant l'observation et la gestion des ressources utilisées par les différents acteurs du système (services ou applications).

**uniformité :** Un système est dit uniforme si le même modèle de programmation est utilisé à tous les niveaux du système. L'uniformité d'un système facilite l'interaction de ses différentes entités. Elle préserve également le continuum système / applications : une application peut, si elle en a besoin et si le système l'y autorise, contrôler elle-même les ressources matérielles et logicielles de la machine.

Bien sûr, toutes ces caractéristiques sont inter-dépendantes; par exemple, modulariser un système le rend flexible et ouvrir un système le rend administrable. THINK définit un canevas logiciel minimum qui permet de ne compromettre aucune de ces caractéristiques a priori. Le canevas sert ensuite de base à l'instanciation de noyaux de systèmes d'exploitation qui feront eux-même et a posteriori les compromis qu'ils estiment nécessaires sur chacune des caractéristiques considérées.

## 4.2 Philosophie

L'environnement THINK suit une philosophie de conception originale, basée sur des techniques de structuration à composants, inspirée par des visions logicielles minimalistes et guidée par le besoin de spécialisation des noyaux de système.

### 4.2.1 Paradigme de structuration par composants

THINK suit le paradigme de structuration par composants. La décomposition suit une règle simple : si, pour un module logiciel donné, il est envisageable de remplacer une partie de l'implémentation par une autre, fonctionnellement identique mais algorithmiquement différente, le module doit être décomposé en deux modules indépendants d'ordre inférieur. Ainsi, d'un point de vue algorithmique, chaque partie d'un système THINK est indépendante.

THINK effectue une décomposition totale du système : tout ce qui peut être décomposé doit l'être, même au plus bas niveau. De même, dans un noyau Think, rien n'existe qui ne soit ou ne fasse partie d'un composant. La décomposition totale du système contribue au haut niveau de flexibilité de l'architecture.

### 4.2.2 Infrastructure logicielle minimaliste

THINK n'est pas un noyau : c'est une infrastructure logicielle légère destinée à la construction de noyaux spécialisés. Autrement dit, THINK fournit avant tout les mécanismes et structures logicielles nécessaires à la construction de noyaux de systèmes d'exploitation par composition. Ces mécanismes et structures ont été conçus pour être minimaux en terme de coût processeur et mémoire.

THINK limite au possible les abstractions et concepts imposés par défaut aux développeurs de noyaux de systèmes d'exploitation. Les seules abstractions absolument nécessaires à un noyau THINK sont celles qui réifient le processeur (ou le micro-contrôleur) de la machine hôte. Toute autre entité peut être construite par composition. Même des concepts aussi primitifs qu'un allocateur mémoire ne font pas partie des composants de base d'un système THINK : en effet, certains systèmes, pour des raisons de sécurité et de déterminisme d'exécution, allouent statiquement la mémoire nécessaire à leur exécution. Dans ce cas, il n'est nul besoin d'inclure un allocateur mémoire : il ne serait jamais utilisé et le noyau occuperait plus d'espace en mémoire pour rien.

Cette vision architecturale extrême rapproche THINK d'un exo-noyau, qui exporte au sein de librairies applicatives l'ensemble des services systèmes et ne laisse au sein du noyau que les mécanismes de multiplexage des ressources (voir section 2.3.5). THINK va en réalité encore plus loin qu'un exo-noyau : même les mécanismes de multiplexage de ressources ne font pas a priori partie du noyau.

### 4.2.3 Implémentations spécialisées

Les objectifs de THINK laissent penser qu'il s'agit d'une implémentation générique de système fonctionnant sur l'ensemble des architectures du spectre de développement visé. Il n'en est rien : THINK ne fournit pas d'implémentation générique de système. Bien au contraire, le portage d'un noyau THINK sur une nouvelle architecture implique la réécriture de l'ensemble des couches de bas niveau du système. Cela concerne notamment les composants réifiant les possibilités matérielles du processeur ou du micro-contrôleur de la machine. THINK facilite néanmoins les choses, en limitant la réécriture des composants aux seuls composants strictement dépendants de l'architecture matérielle considérée. Les autres composants ne seront réécrits que dans le cas où d'autres implémentations sont nécessaires (pour des raisons de performance par exemple). Ainsi, un noyau construit avec THINK est optimal : chaque implémentation est spécialisée pour correspondre aux besoins exacts de la machine et des applications cibles.

Pour les composants de plus haut niveau, une interface générique est fournie quand cela est possible. Par exemple, un allocateur mémoire offrira toujours les

méthodes `malloc()` et `free()`, même si l'implémentation de ces méthodes dépend fortement de l'architecture de la machine cible. Pour l'allocateur mémoire, une interface générique est fournie. Cette interface sera déclarée par tous les composants allocateurs de mémoire, et ceux quelque-soit la machine pour lequel ils sont conçus.

Bien évidemment, les implémentations de certains concepts de haut niveau sont communes, et ceux quelque-soit les architectures considérées. Ainsi, les protocoles réseaux TCP/IP par exemple sont communs à toutes les architectures. Pour ceux-ci, THINK fournit une implémentation standard sous forme de composants. Le découpage en composants permet ici de détacher chaque protocole et de les rendre indépendants entre eux , et indépendants du reste du système. Un composant implémentant le comportement de TCP pourra donc être employé directement, sans aucune modification de son code, par tous les systèmes qui en ont le besoin.

## 4.3 Modèle général de composition

Cette section présente le modèle général de composition proposé par l'architecture THINK. Ce modèle est directement inspiré des travaux de recherche sur la conceptualisation de liaisons entre objets répartis menés par Gordon Blair et Jean-Bernard Stefani dans le cadre de la définition du standard RM-ODP [X.995] et de la plate-forme logicielle de communication Jonathan [Kra02]. Ses objectifs sont néanmoins plus larges que ceux de RM-ODP, dans lequel on se limite aux systèmes distribués. Ici, il s'agit de concevoir un modèle de composition général, pouvant servir de base à la construction de n'importe quel système, quelque soit son type.

Le modèle de composition de THINK est organisé autour d'un ensemble très réduit de concepts. Cet ensemble a été pensé pour être complet et minimal : complet car il permet l'instanciation de n'importe quel système, et minimal car il ne regroupe que les concepts strictement nécessaires à sa complétude. Ces concepts sont au nombre de cinq : le composant, l'interface, la liaison, le nom (et, par extension, le contexte de désignation) et la fabrique.

### 4.3.1 Composants

Le composant est l'unité logicielle de l'architecture THINK. Un composant réifie un élément (matériel, logiciel, structurel ou autre) d'un système THINK. Il encapsule un état, qui représente l'état de l'élément réifié. Il offre à son environnement une (ou plusieurs) abstractions de cet état : ce sont les services offerts par l'élément réifié. Considérons par exemple l'élément matériel "mémoire" d'un

système; cet élément peut être réifié en un composant. Son état sera constitué des informations sur l'allocation des zones en mémoire. L'abstraction qui en sera offerte à l'environnement pourra prendre la forme d'un allocateur mémoire.

La granularité d'un composant n'est pas imposée par le modèle. La taille et la complexité de l'élément réifié par un composant est ainsi parfaitement arbitraire. Une plate-forme de répartition J2EE peut être réifiée par un composant Think, au même titre qu'un registre de configuration d'un micro-contrôleur.

Le modèle n'émet aucune hypothèse sur la nature logicielle d'un composant, et notamment sur le langage de programmation utilisé pour l'implanter. Un composant peut ainsi être constitué de routines assembleurs, de modules C ou même d'objets Java, avec pour seule contrainte la nécessité de posséder l'environnement d'exécution adéquat. De même, la nature des interactions entre un composant et son environnement n'est pas imposée non plus. Interactions synchrones (appels de procédures, etc.) ou asynchrones (communication par événements, etc.) sont envisageables, pour peu que le système comporte les mécanismes permettant de les mettre en œuvre.

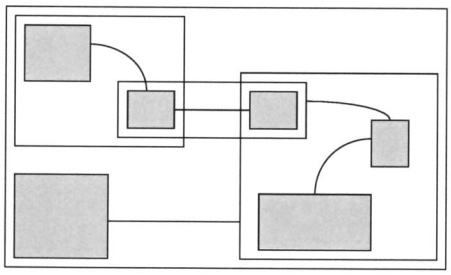

FIG. 4.1 – *Un exemple de composition de composants*

Un système THINK est une composition plus ou moins complexe de composants (voir figure 4.1). Le modèle permet l'assemblage de composants selon deux relations différentes de composition :

**relation d'appartenance :** cette relation permet d'inclure un ou plusieurs composants au sein d'un autre composant d'ordre supérieur. Un composant d'ordre supérieur est responsable de l'administration de l'ensemble de ces sous-composants. La relation d'appartenance permet ainsi de créer des compositions récursives, ou chaque composant est partie intégrante d'un composant de plus haut niveau, formant ainsi un arbre dont le composant racine représente le système lui-même.

**relation de liaison :** cette relation permet de lier deux (ou plus) composants appartenant à un même niveau de hiérarchie. La liaison s'effectue au niveau des interfaces et véhicule les interactions entre les composants liés.

Le modèle ne suppose pas de relation unique d'appartenance pour un composant donné : un composant peut appartenir à plusieurs composants d'ordre supérieur (une telle situation apparaît d'ailleurs sur la figure 4.1). Autrement dit, le partage de composants est autorisé.

### 4.3.2 Interfaces

Un composant interagit avec son environnement au travers de points d'accès nommés interfaces. Une interface permet de manipuler (lire ou modifier) l'état de l'élément réifié par le composant auquel elle appartient. A chaque interface correspond une vue (une abstraction) de l'état du composant. Ainsi, un composant offre autant d'interfaces différentes que d'abstractions différentes de son état interne.

Au sein d'un système Think, il existe deux natures d'interfaces :

**les interfaces serveur :** les interfaces de cette nature modélisent un service offert par le composant.

**les interfaces client :** les interfaces de cette nature modélisent un service requis par le composant. Un service requis par un composant doit lui être fourni par son environnement d'exécution (sous la forme d'un service offert par un autre composant).

Le principe des interfaces serveur et client est illustré par la figure 4.2, où deux composants A et B sont reliées par leurs interfaces (le composant A offre un service au composant B). Le mécanisme d'interaction mis en œuvre dans cet exemple est de type appel de procédure, mais le modèle ne pose pas de contrainte sur la nature des interactions entre composants.

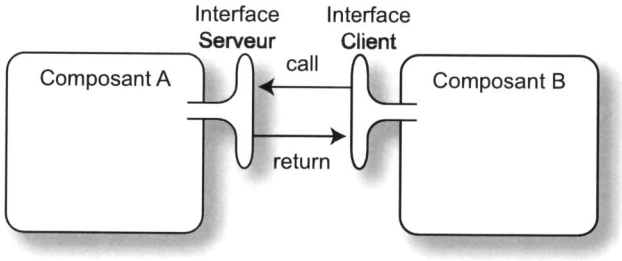


FIG. 4.2 – *Interfaces serveur et interfaces client*

Les interfaces sont typées. Le type d'une interface est lié au service représenté par l'interface. Il est indépendant de sa nature : une interface serveur et une interface client peuvent être de même type. Le type d'un composant est un ensemble constitué des types de chacune de ses interfaces (serveur et client).

Le modèle de composition de THINK ne suppose aucun système de type a priori. La seule contrainte sur ce dernier concerne sa structure : il doit être organisé en un treillis, être ordonnancé par une relation de sous-typage et posséder un plus grand élément commun à tous les types. Le choix d'un système de type particulier et de son implémentation est effectué par le développeur du système lui-même.

Le concept d'interface typée permet de séparer de façon explicite la spécification d'un service de son implémentation effective. Cela augmente significativement le niveau de flexibilité de l'architecture : le développeur peut choisir (à la conception ou dynamiquement si le système le permet) entre plusieurs implémentations alternatives d'un même service.

### 4.3.3 Liaisons

Il est dit plus haut qu'une des relations de composition du modèle est la relation de liaison. Celle-ci se traduit directement dans le modèle par le concept de liaison.

Une liaison modélise un canal d'interactions entre deux composants ou plus. Elle réifie l'ensemble de la chaîne de communication entre les interfaces des composants connectés. Aucune sémantique n'est associée au concept de liaison : la chaîne de communication englobée par une liaison est arbitraire. Elle peut par exemple comporter des éléments protocolaires complexes : ce sera le cas d'une liaison entre deux composants situés sur des machines différentes sur le réseau. Elle peut au contraire n'être constituée que d'un pointeur vers une adresse mémoire : ce sera le cas d'une liaison bas-niveau entre deux composants du noyau. Il est néanmoins possible de regrouper les différents types de liaisons en trois catégories distinctes (voir figure 4.3) :

**liaison langage :** il s'agit de l'association entre un symbole langage et une structure mémoire (ou un registre de périphérique). Autrement dit, une liaison langage représente le lien, dans un programme, entre un nom de variable (ou de méthode) et sa valeur (ou son implémentation). Cette liaison est en général établie par le compilateur ou par l'éditeur de liens.

**liaison simple :** la liaison simple offre la plus simple réification de la liaison entre composants. Elle est uniquement constituée d'un pointeur vers le descripteur d'interface d'un composant. Elle représente la forme la plus simple des liaisons dont l'établissement peut se faire dynamiquement. La différence entre une liaison langage et une liaison simple réside justement dans la dynamicité de son établissement : une liaison simple peut en effet être considérée comme une liaison langage non résolue par l'édition de

liens.

**liaison complexe :** les liaisons dont la sémantique est avancée sont réifiées par une liaison complexe. L'objet de liaison complexe est un composant THINK à part entière. Tout ce qu'il est possible de dire sur un composant s'applique donc à la liaison complexe. La connexion d'une liaison complexes aux composants qu'elle a pour rôle de relier est effectuée par des liaisons simples.

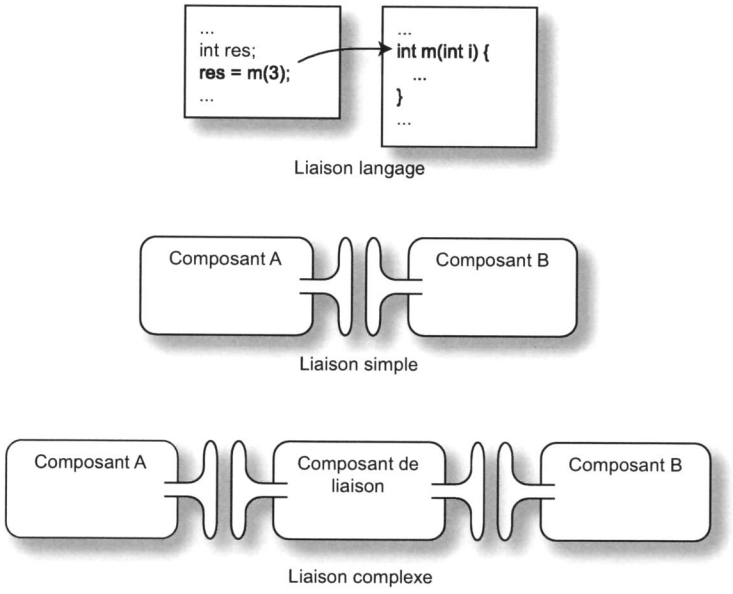


FIG. 4.3 – *Les différentes catégories de liaison*

Plusieurs types de liaisons peuvent cohabiter au sein d'un système THINK à un instant donné. Le type de la liaison est déterminé au moment de son établissement, qui peut avoir lieu soit statiquement (dans le cas des liaisons langages par exemple), soit dynamiquement. Le modèle permet en outre la modification d'une liaison à l'exécution. L'établissement et la modification dynamique des liaisons permet de pratiquer des choix d'implémentation beaucoup plus fins.

### 4.3.4 Noms et contextes de désignation

L'établissement d'une liaison s'effectue entre des interfaces de composant. Afin de pouvoir manipuler ces interfaces à l'exécution, il est nécessaire de les identifier. L'identification d'une interface au sein d'un système THINK est réalisé à l'aide de noms et de contextes de désignation.

Au sein d'un système Think, chaque interface possède un nom. Un nom est valable dans un contexte de désignation particulier. Un contexte de désignation

définit des règles de nommage. Il est responsable de la création des noms et de leur association avec une interface. Le contexte de désignation doit inclure au sein du nom les informations qui vont lui permettre, ultérieurement, de retrouver l'interface qui lui est associée.

Aucune contrainte n'est imposée sur la structure d'un nom. Toutefois, à partir d'un nom, deux opérations doivent être possibles : retrouver le contexte de désignation associé au nom et obtenir une forme sérialisée du nom. A l'inverse, un contexte de désignation doit permettre de reconstruire un nom à partir de sa forme sérialisée.

Un système THINK peut comporter plusieurs contextes de désignation. Une interface peut posséder plusieurs noms, chacun valide dans un contexte de désignation particulier. Le modèle de composition n'impose pas de contrainte sur l'organisation de ces contextes.

Notons enfin qu'un nom et qu'un contexte de désignation sont eux-même des composants THINK.

### 4.3.5 Fabriques

L'instanciation dynamique d'un composant THINK est effectuée par une fabrique. Chaque fabrique a la charge de l'instanciation d'un type de composants particulier. Il existe ainsi autant de fabriques de composants différentes que de types de composants différents.

Une fabrique peut prendre des formes très diverses. Il peut par exemple s'agir d'un simple constructeur (équivalent au `new()` du langage Java); d'autres fabriques, plus complexes, prennent la forme d'un composant THINK. Le modèle de composition n'impose ici non plus aucune contrainte particulière sur la structure d'une fabrique.

Certaines fabriques ont un rôle particulier. Les fabriques de liaison, par exemple, ont pour rôle d'instancier un composant de liaison et d'établir ainsi une communication entre plusieurs composants (voir figure 4.4). Selon la nature de la liaison, cette instanciation peut elle-même impliquer l'appel à plusieurs autres fabriques (pour instancier des piles protocoles de communication par exemple). Un autre type de fabrique de liaison existe et a déjà été présenté plus haut : les contextes de désignation. En effet, le rôle d'un contexte de désignation est, entre autres, d'instancier des noms. A ce titre, il peut être qualifié de fabrique de noms.

Le fait qu'une fabrique de composants puisse elle-même être un composant pose le problème de l'instanciation de la fabrique elle-même. Celle-ci peut elle-même être effectuée par une fabrique de composants (autrement dit, il peut

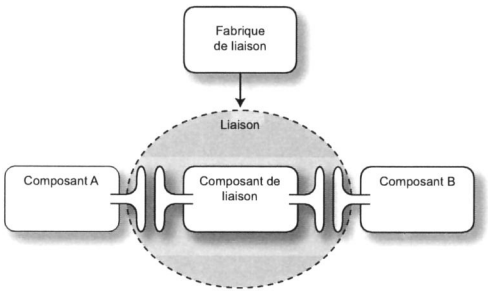


FIG. 4.4 – *Un exemple de fabrique : la fabrique de liaison*

exister des fabriques de fabriques). Le système doit mettre fin lui-même à cette récursion, en instanciant et chargeant lui-même des fabriques "primitives" à son initialisation.

## 4.4 Instanciation de l'architecture Think

L'architecture THINK a été présentée dans la section précédente; cette section montre maintenant son instanciation dans un cadre matériel et logiciel précis.

L'instanciation du modèle de composition de THINK commence par la recherche d'une mise en œuvre efficace de chacun des concepts du modèle et par la caractérisation des interfaces de programmation nécessaires au développement d'un noyau de système d'exploitation. Il est ensuite nécessaire de mettre à la disposition des concepteurs de systèmes une librairie au sein de laquelle ils pourront récupérer des implémentations par défaut de certains composants (pilotes de périphériques, gestionnaires mémoire, ordonnanceurs, etc.).

Les sections 4.4.1 et 4.4.2 présentent respectivement l'implémentation du modèle de composition et une bibliothèque de composants à destination des architectures matérielles de type PowerPC nommée Kortex. La section 4.4.3 détaille les différents outils d'aide à la construction mis à la disposition du développeur de systèmes.

### 4.4.1 Implémentation du modèle de composition

Cette section présente l'implémentation de THINK telle que proposée par Jean-Philippe Fassino. Le cœur du modèle de composition est implémenté en langage C et les interfaces de programmation sont définies en Java. Si l'implémentation présentée ici cible particulièrement les architectures PowerPC [?], la majorité du code présenté ici est directement réutilisable sur d'autres cibles matérielles.

#### 4.4.1.1 Structure exécutive d'une interface

A l'exécution, l'instance d'une interface d'un composant se manifeste par une structure de données appelée descripteur d'interface. Celui-ci contient deux pointeurs, nommés respectivement meth et data (voir figure 4.5). Le pointeur meth pointe vers une structure contenant les références vers chacune des implémentations des méthodes définies par l'interface. Le pointeur data, lui, référence les éventuelles données d'instances du composant.

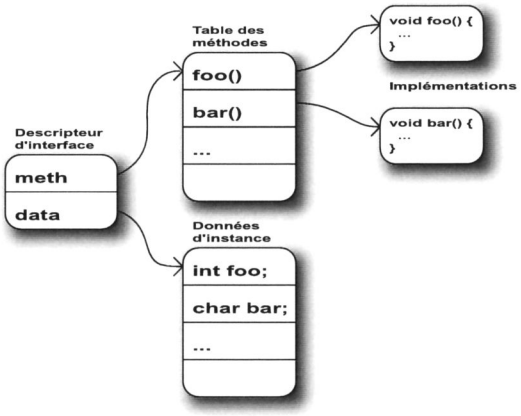


FIG. 4.5 – *Structure d'une interface*

Les données d'instance sont partagées entre toutes les interfaces du composant : les pointeurs data de toutes les interfaces d'un composant pointent sur la même structure de données. Toutefois, une optimisation est effectuée dans le cas d'un composant ne définissant qu'une seule interface. Dans ce cas (et uniquement dans ce cas), toutes les données d'instance du composant sont placées directement dans le descripteur d'interface à la suite du pointeur `meth`, ce qui évite une indirection (voir figure 4.6).

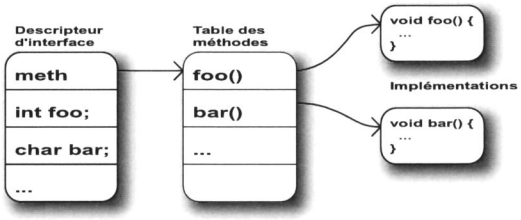


FIG. 4.6 – *Structure optimisée dans le cas d'un composant ne définissant qu'une seule interface*

Une référence d'interface est constituée par un pointeur vers le descripteur

d'interface correspondant. L'accès à une méthode de l'interface correspond à un accès à l'adresse mémoire contenue dans le champ correspondant à la méthode appelée au sein la structure pointée par `meth`.

La technique d'implémentation des interfaces présentée ici est très similaire au mécanisme des V-Table proposé par C++ pour l'implémentation des interfaces d'objet [?]. Elle est très efficace: l'accès aux méthodes définies par l'interface se fait par adressage direct.

#### 4.4.1.2 Structure exécutive d'un composant

Un composant THINK comporte un état, accessible au travers de méthodes définies par des interfaces. A l'exécution, l'instance d'un tel composant se caractérise essentiellement par un ensemble de données d'instance (référencées par chacune des interfaces du composant) et un ensemble de méthodes correspondant aux différentes interfaces offertes par le composant (voir figure 4.7).

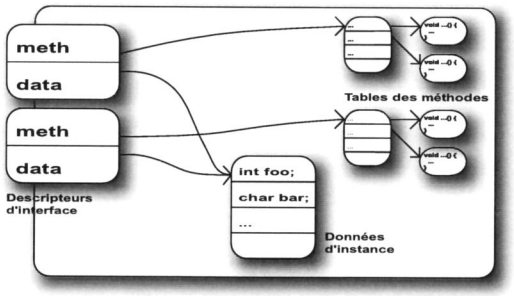


FIG. 4.7 – *Une instance d'un composant*

Plusieurs instances d'un même composant peuvent cohabiter au sein du même système. Lorsque deux instances d'un même composant sont présentes en mémoire au même instant, chacune possède ses propres instances d'interface et ses propres données d'instances. Le code des méthodes d'une interface est, lui, partagé par les instances (voir figure 4.8).

Lorsqu'une méthode est appelée, elle manipule les données d'instance du composant auquel elle appartient. Or, sans plus d'information et puisque le code des méthodes est partagé par les différentes instances du composant, une méthode n'a aucun moyen de déterminer sur quelles données effectuer son travail (ce problème est un classique dans le domaine des langages à objets). Pour remédier à ce problème, chaque méthode comporte un paramètre appelé `this`. Ce paramètre est affecté au moment de l'appel au descripteur d'interface dont l'appel est issu. L'accès aux données d'instance du composant se fait ensuite à

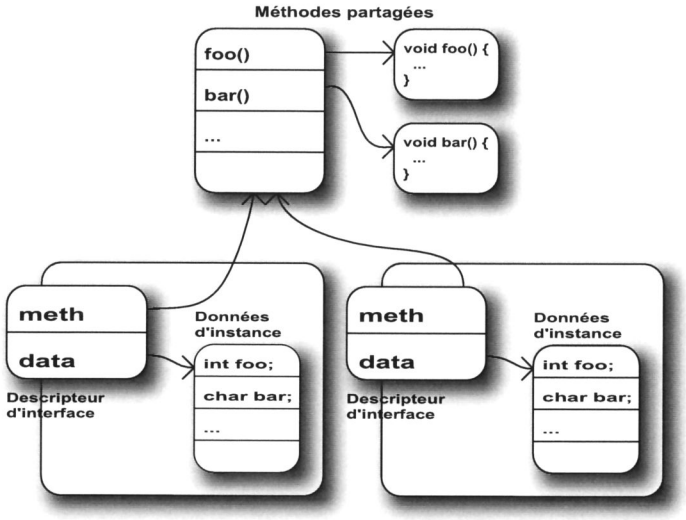


FIG. 4.8 – *Deux instances d'un même composant*

travers le champ `data` du descripteur d'interface et plus aucune confusion n'est ensuite possible.

#### 4.4.1.3 Interfaces de programmation

Les structures exécutives présentés ci-dessus sont complétées par un ensemble d'interfaces de programmation. Chacune de ces interfaces traduit un concept du modèle de composition et sera implémentée par un composant de la bibliothèque Kortex (voir section 4.4.2). Le langage de description d'interface utilisé est Java, mais rien ne s'opposerait à l'utilisation d'un autre langage (IDL de Corba [COR01] par exemple).

La section 4.3.2 précise que les interfaces THINK sont typées et que le système de type s'organise sous forme de treillis ordonné et possédant un plus grand élément. Ce dernier se manifeste par l'interface `Top` (voir listing 4.1). `Top` est le type commun à toutes les interfaces d'un système THINK (chaque interface de THINK hérite implicitement de `Top`).

Listing 4.1: *Interface Top*

```
interface Top {
}
```

Les concepts de nom et de contexte de désignation, présentés en section 4.3.4, se traduisent par les interfaces `Name` et `NamingContext`. L'interface `Name` offre les opérations `getDefaultNC()` et `toByte()` (voir listing 4.2). L'opération

`getDefaultNC()` permet d'obtenir la référence du contexte de nommage à partir duquel le nom a été créé. L'opération `toByte()`, elle, permet de récupérer une chaîne d'octets représentant le nom sous sa forme sérialisée.

Listing 4.2: *Interface Name*

```
interface Name {
    NamingContext getDefaultNC();
    String toByte();
}
```

L'interface NamingContext offre les opérations `byteToName()` et `export()` (voir listing 4.3). L'opération `byteToName()` permet de récupérer un nom à partir de sa forme sérialisée (à la condition que le nom passé en paramètre ait été créé par ce contexte de désignation). C'est la méthode duale de l'opération `toByte()` définie par l'interface `Name`. L'opération `export()` permet de créer un nom à partir de l'interface passée en paramètre. Plus précisément, `export()` crée un lien de désignation entre une interface et un nom instancié pour l'occasion. Au sein d'un contexte de nommage, un tel lien est unique : si un tel lien existait, aucun nouveau nom n'est créé (le nom précédemment instancié est simplement retourné).

Listing 4.3: *Interface NamingContext*

```
interface NamingContext {
    Name byteToName(String strname);
    Name export(Top itf);
}
```

Au sein d'un système Think, tous les composants sont instanciés par des fabriques. Ainsi, une liaison entre deux composants est instanciée par une fabrique de liaisons (voir sections 4.3.3 et 4.3.5). Les fabriques de liaisons implémentent l'interface `BindingFactory` (voir listing 4.4).

Listing 4.4: *Interface BindingFactory*

```
interface BindingFactory {
    Top bind(Name name);
}
```

L'interface `BindingFactory` ne définit qu'une seule opération, appelée `bind()`. L'opération `bind()` crée une liaison vers l'interface désignée par le nom passé en paramètre et renvoie une interface locale d'appel à l'interface distante. Dans le cas de liaisons complexes, cette opération peut inclure des processus tels que la création de talons et de squelettes de communication. Au contraire,

dans le cas de liaisons primitives, `bind()` renvoie directement l'interface distante (autrement dit, l'interface d'appel et l'interface distante sont les mêmes). Les opérations `bind()` de l'interface `BindingFactory` et `export()` de l'interface `NamingContext` sont duales.

### 4.4.2 Bibliothèque de composants Kortex

Le modèle de composition de THINK ne définit qu'un ensemble minimal de concepts. Tout concept de plus haut niveau doit être construit par composition. Toutefois, afin de faciliter la tâche de développement d'un noyau de système, THINK est accompagné d'une librairie de composants optionnels nommée Kortex.

La librairie Kortex propose une implémentation par défaut de nombreux services systèmes (ordonnanceurs, gestionnaires mémoire, etc.) et de pilotes de périphériques (pilotes clavier, réseau, etc.) Elle a été développée pour des architectures PowerPC, mais de nombreux composants sont réutilisables sur d'autres cibles matérielles (moyennant une nécessaire recompilation). Cette section dresse une liste non exhaustive des composants disponibles au sein de Kortex.

#### 4.4.2.1 Modèle de composition

Certains composants sont nécessaires à l'implémentation du modèle de composition lui-même. Kortex fournit notamment un mini-courtier d'interfaces permettant, à partir d'une chaîne de caractères, d'obtenir le nom d'une d'interface. Deux fabriques de liaisons sont également disponibles, `localBF` et `remoteBF`. La fabrique `localBF` permet de créer des liaisons primitives entre composants (la liaison primitive est présentée dans la section 4.3.3). La fabrique `remoteBF` est utilisée pour créer des liaisons entre composants situés sur des machines distantes sur le réseau.

Le tableau 4.1 récapitule les composants nécessaires à l'implémentation du modèle de composition.

| Composant | Interface | Description |
|---|---|---|
| `trader` | `Trader` | Mini-courtier d'interfaces |
| `localBF` | `BindingFactory`, `NamingContext` | Fabrique de liaisons primitives |
| `remoteBF` | `BindingFactory`, `NamingContext` | Fabrique de liaisons distantes |

TAB. 4.1 – *Composants du modèle de composition*

#### 4.4.2.2 Mémoire

Les composants de la bibliothèque Kortex permettent deux modes de gestion de la mémoire : la mémoire paginée et la mémoire plate. La mémoire paginée sera par exemple utilisée par les systèmes multi-processus, alors que l'utilisation de la mémoire plate sera préférable dans le cas de systèmes dédiés à une application.

Les composants nécessaires à l'implémentation de ces deux modes de gestion de la mémoire sont présentés dans le tableau 4.2.

| Composant | Interface | Description |
|---|---|---|
| mmu | MMU | Unité de gestion de la mémoire du PowerPC |
| buddy | Page | Buddy-système |
| dlmalloc | Allocator | Allocateur de mémoire |
| sbrk | Sbrk | Routine sbrk pour mémoire plate |
| sbrkmap | Sbrk | Routine sbrk pour mémoire paginée |
| flat | Space | Espace d'adressage pour mémoire plate |
| pagetable | Space | Espace d'adressage pour mémoire paginée |
| spacefactory | Space | Usine à espaces d'adressages |

TAB. 4.2 – *Composants de gestion de la mémoire*

#### 4.4.2.3 Processeur

La bibliothèque Kortex fournit un ensemble de composants réifiant les capacités du processeur PowerPC. Parmi les services proposés, on trouve notamment la gestion des interruptions et de la concurrence. Plusieurs algorithmes d'ordonnancement de processus concurrents sont disponibles : ordonnancement round-robin, à priorités, ou coopératif. C'est au développeur de faire un choix entre ces différents algorithmes à la conception du système.

Le tableau 4.3 résume les différents composants liés à la gestion du processeur.

| Composant | Interface | Description |
|---|---|---|
| trap | Trap | Interruption PowerPC |
| irq | IRQ | Gestionnaire d'interruptions |
| roundrobin | Scheduler | Ordonnanceur préemptif circulaire |
| priority | Scheduler | Ordonnanceur préemptif à priorités |
| cooperative | Scheduler | Ordonnancement coopératif |

TAB. 4.3 – *Composants de gestion du processeur*

#### 4.4.2.4 Réseau

Le support réseau offert par la bibliothèque Kortex est constitué d'un ensemble de protocoles et de pilotes de carte réseau. Chaque protocole est implémenté par un composant particulier. Ceux-ci s'assemblent pour former une pile de protocoles; les règles d'assemblage de protocoles suivent le modèle x-Kernel [HP91].

Les protocoles Ethernet, ARP, IP, UDP, TCP, NFS et SunRPC sont actuellement disponibles au sein de Kortex. En outre, La bibliothèque supporte les cartes réseau les plus courantes sur Power-Macintosh (cartes mace, bmac, gmac et tulip).

Le tableau 4.4 résume les différents composants nécessaires à la gestion du réseau.

| Composant | Interface | Description |
|---|---|---|
| mace | Net | Pilote carte mace |
| bmac | Net | Pilote carte bmac |
| gmac | Net | Pilote carte gmac |
| tulip | Net | Pilote carte Tulip |
| ip | ProtocolHigh | Protocole Ethernet |
| udp | ProtocolHigh | Protocole UDP |
| tcp | ProtocolHigh | Protocole TCP |
| arp | ARP | Protocole ARP |
| nfs | VFS | Protocole NFS |

TAB. 4.4 – *Composants de gestion du réseau*

#### 4.4.2.5 Périphériques

La bibliothèque Kortex propose les pilotes de quelques contrôleurs de périphériques courants. Kortex prend notamment en charge le bus PCI, les contrôleurs clavier et les cartes graphiques.

Le tableau 4.5 liste les différents pilotes de périphériques disponibles au sein de Kortex.

### 4.4.3 Outils d'aide à la construction

THINK est un environnement complet de développement de systèmes. A ce titre, il offre (en sus du modèle de composition, de son implémentation et de la bibliothèque Kortex) certains outils d'aide à la conception et à la composition de systèmes. Ceux-ci sont actuellement au nombre de deux : un méta-compilateur

| Composant | Interface | Description |
|---|---|---|
| fw | Firmware | Pilote firmware Power-Macintosh |
| uninorth | PCI | Pilote PCI uninorth |
| grackle | PCI | Pilote PCI grackle |
| pmac | PCI | Pilote PCI bandit et chaos |
| openpic | PIC | Pilote PIC openpic |
| pmacpic | PIC | Pilote PIC macio et GC |
| pmu | VIA | Pilote ADB pmu |
| cuda | VIA | Pilote ADB cuda |
| offb | FrameBuffer | Pilote carte vidéo |
| f8x16 | Font | Police de caractères |
| console | Console | console graphique |

TAB. 4.5 – *Composants de gestion de périphériques*

d'interfaces et un configurateur. Chacun intervient en un point précis de la chaîne de génération d'un noyau THINK (voir figure 4.9).

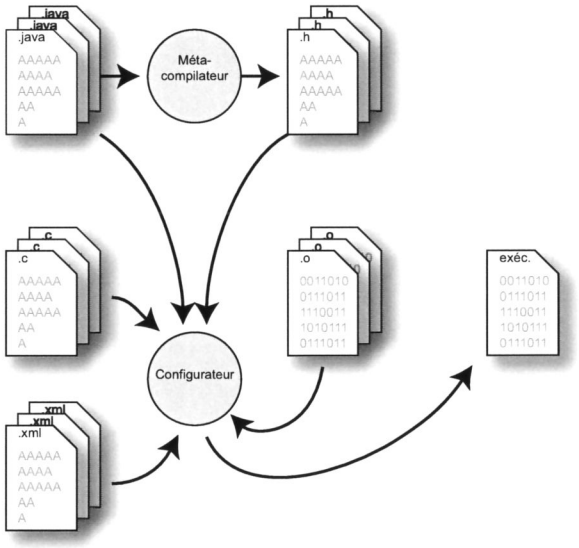


FIG. 4.9 – *La chaîne de génération d'un noyau Think*

Le méta-compilateur d'interfaces génère, à partir d'une description en langage Java d'une interface, les structures C nécessaires à son utilisation par un composant du noyau. Son fonctionnement est décrit par la figure 4.10. Dans un premier temps, l'interface Java est compilée. La classe générée est ensuite inspectée et une représentation abstraite de l'interface est construite. Enfin, un formateur utilise cette représentation pour générer les structures logicielles correspondant à l'interface de départ. Plusieurs formateurs différents peuvent être

employés. Chacun génèrera des structures logicielles différentes à partir d'une même interface Java.

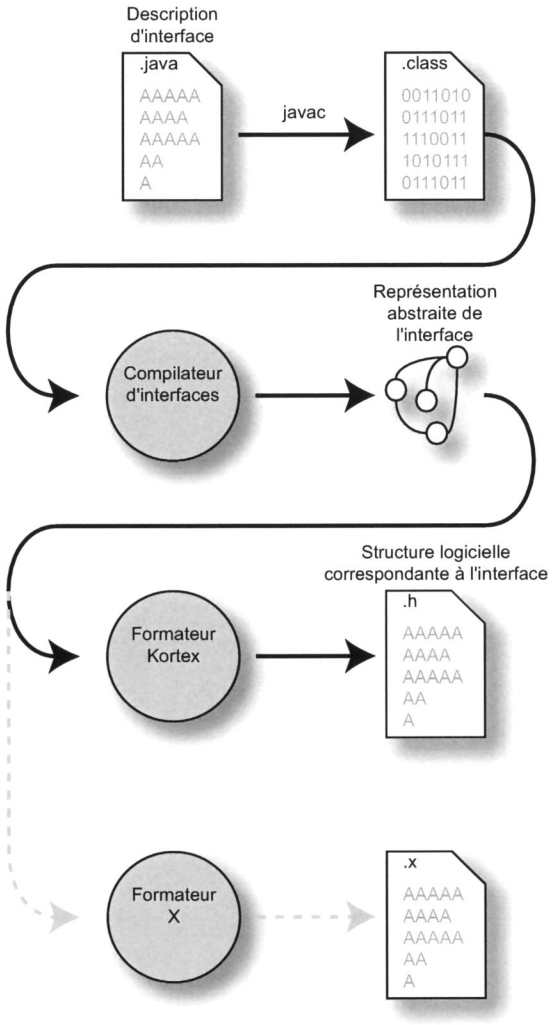


FIG. 4.10 – *Le fonctionnement du méta-compilateur*

La séparation explicite entre l'analyse de la description d'une interface et la génération de la structure logicielle correspondante justifie l'emploi du terme "méta-compilateur" pour qualifier cet outil. Elle permet de rendre générique les mécanismes d'analyse de la description d'une interface : si la plate-forme cible change, il suffit d'écrire un formateur compatible avec la nouvelle plate-forme.

Le configurateur, lui, base son travail sur une description XML des composants et de la composition globale du système (qui lui est fournie par le concepteur du système). Il vérifie que, pour la composition globale proposée, les dépendances entre composants sont résolues (c'est à dire que toutes les inter-

faces requises sont fournies). Si tel est le cas, il génère une procédure globale d'initialisation du noyau qui ordonnance la création des instances de tous les composants du système. Il compile ensuite (si besoin) chaque composant du noyau et les lie pour former le noyau final. La figure 4.11 décrit le mécanisme du configurateur.

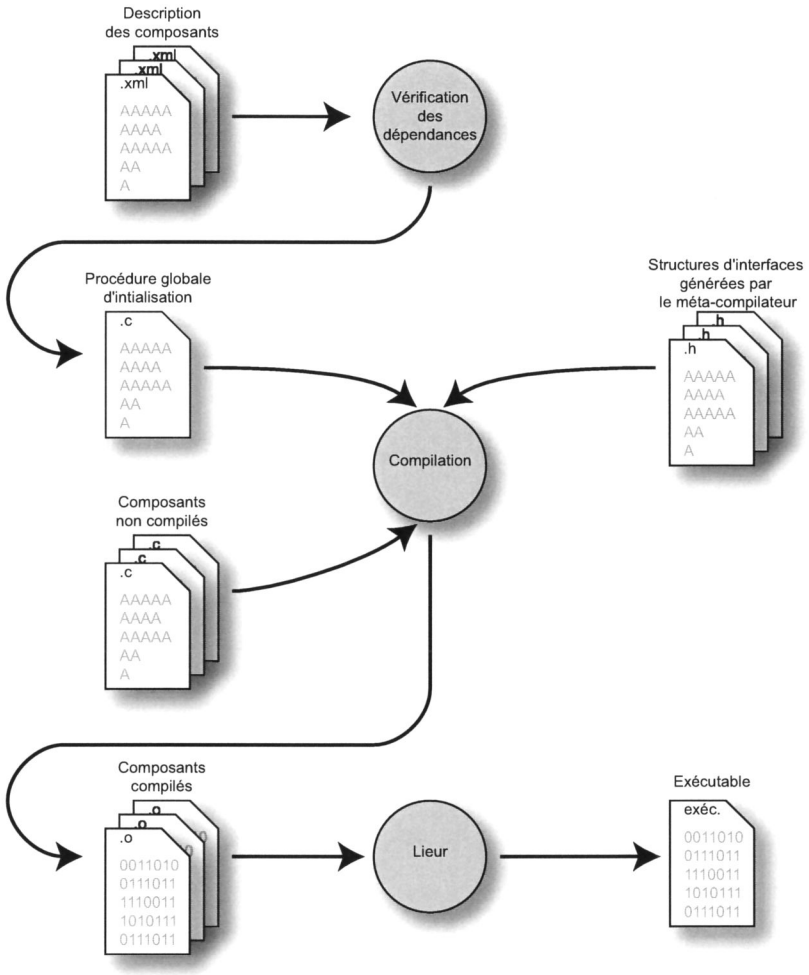


FIG. 4.11 – *Le fonctionnement du configurateur*

## 4.5 Vers une nouvelle implémentation de Think

L'implémentation actuelle de THINK couvre l'ensemble des concepts de base du modèle de composition mais reste néanmoins perfectible. Il serait en outre intéressant d'explorer plus en avant les possibilités particulières qu'offre ce dernier, notamment en matière de reconfiguration et de flexibilité.

Cette section fait le point sur les avantages et les inconvénients de l'implémentation actuelle et pose les bases d'une nouvelle implémentation.

### 4.5.1 Atouts de l'implémentation actuelle

L'implémentation actuelle de THINK présente un certain nombre d'atouts qui la rendent très intéressante du point de vue de la construction de noyaux de systèmes performants.

Tel qu'il est implémenté à l'heure actuelle, le modèle de composition de THINK permet la construction de noyaux de systèmes d'exploitation à composants de façon très efficace. Les coûts d'exécution liés au modèle de composition lui-même sont extrêmement faibles. Par exemple, les liaisons primitives entre composants suivent une structure optimisée qui les rend particulièrement performantes : un appel de méthode à travers une liaison primitive n'implique que six instructions et ne prend que huit cycles d'exécution sur PowerPC G4, soit 16 nanosecondes sur un PowerPC G4 cadencé à 500MHz. De façon similaire, l'espace mémoire occupé par le modèle est réduit à son minimum : il se limite aux structures des interfaces des composants du noyau (la place occupée par les données d'instance d'un composant n'est pas, de part sa nature fonctionnelle, attribuée au modèle de composition). Pour chaque interface, cela se traduit par un descripteur d'interfaces (deux pointeurs) et une table des méthodes (autant de pointeurs que de méthodes définies par l'interface). Enfin, l'empreinte minimale d'un noyau THINK est très faible : seuls les composants réifiant le processeur de la machine hôte sont strictement nécessaires. Tout autre entité du système est optionnelle.

Le niveau de flexibilité d'un noyau THINK construit selon l'implémentation actuelle est particulièrement intéressant. Le concept d'interface permet au développeur du système de choisir, au moment de la conception, entre plusieurs implémentations alternatives d'un composant (sous la condition que chaque implémentation propose exactement les mêmes fonctionnalités). En outre, la bibliothèque Kortex opère un découpage du noyau en entités logicielles extrêmement fines. L'importante modularité du noyau permet des choix d'implémentation très fins et contribue fortement au niveau de flexibilité global de l'architecture. Enfin, l'indépendance réciproque des différents composants du noyau assure un niveau de réutilisation important.

Les outils mis à la disposition du développeur du système pour lui faciliter les tâches de conception et de validation d'une composition lui permettent de travailler sur une abstraction du système. Leur conception générique et leur combinaison avec un langage de description de composants basé sur XML permet

leur utilisation dans le cadre de différentes implémentations et cibles matérielles.

### 4.5.2 Faiblesses de l'implémentation actuelle

Si l'implémentation actuelle de THINK présente des atouts indéniables, elle reste néanmoins non exempte de certaines faiblesses.

Le modèle de composition de THINK définit deux relations de composition : l'appartenance et la liaison (voir section 4.3.1). Seule la relation de liaison est effectivement transcrite dans l'implémentation actuelle. Aucun support logiciel n'existe permettant de composer de façon hiérarchique des composants pour former un seul composant de plus haut niveau. Ainsi, il n'est par exemple pas possible de définir des comportements d'administration communs pour un ensemble de composants par ce biais là. Il n'est pas non plus possible de manipuler une composition complexe de composants comme un composant unique. Si cette limitation ne pose pas de réels problèmes pour des noyaux relativement basiques, cela n'est évidemment plus le cas lorsque l'on a affaire à des noyaux plus complexes, où le besoin de hiérarchisation est réel. Il n'est en effet pas réaliste de supposer que les composants de haut niveau que sont les applications manipuleront directement l'ensemble des composants de très bas niveau du système : il est de loin préférable de donner une vue abstraite de certaines sous-parties du système sous forme d'un composant regroupant en son sein plusieurs entités d'ordre inférieur. Ne pas disposer d'une implémentation de la relation d'appartenance limite donc fortement la gamme de systèmes modélisables par l'architecture THINK.

Un des objectifs affirmés de l'architecture THINK se rapporte à la reconfiguration dynamique d'un noyau. Théoriquement, il doit être possible de modifier, remplacer, supprimer ou ajouter toute partie du système en cours d'exécution. Pourtant, aucun mécanisme de ce type n'est proposé dans l'implémentation actuelle. La reconfiguration d'un noyau ne peut se faire qu'au prix d'une recomposition statique, et donc d'un redémarrage du système. La valeur ajoutée que représenteraient des mécanismes de reconfiguration dynamiques au sein d'un noyau de système serait pourtant immense et permettrait d'envisager de nombreuses applications dans des domaines très variés.

Il est un troisième aspect, lié à la reconfiguration dynamique, qui n'est pas non plus traité par THINK pour l'instant : celui des informations disponibles sur le système et ses composants à l'exécution. Il serait par exemple utile de connaître, pour un composant donné, les interfaces dont il dispose, ses éventuels sous-composants et leur organisation, ou encore une abstraction de son état interne. Il serait également envisageable de définir certaines propriétés sur un

composant, comme un nom (actuellement seules les interfaces sont nommées) ou un type. Enfin, disposer de telles informations à l'exécution sont obligatoires si l'on désire proposer des mécanismes de reconfiguration dynamique. Il est donc nécessaire que THINK offre les structures nécessaires au stockage de méta-informations sur un composant et fournisse des primitives de manipulation de ces informations (de façon similaire à ce qui est fait dans le paquetage java.lang.reflect du langage Java, qui permet d'accéder aux méta-informations d'un objet, d'une classe ou d'une méthode Java).

Enfin, les liaisons entre composants THINK telles que proposées par l'implémentation actuelle ne fournissent qu'un mode de communication par appel de méthodes synchrone. Or, le modèle de composition ne suppose pas de mécanisme d'interaction à priori. Certaines gammes de systèmes privilégient même, pour des raisons de performance et de prédictabilité, les communications asynchrones (communications de type événementielles ou autre). Il serait intéressant, pour répondre à la plupart des systèmes, de disposer de plusieurs modes d'interaction alternatifs.

### 4.5.3 Objectifs d'une nouvelle implémentation

Il existe encore de grandes différences entre le modèle théorique de composition de THINK et le prototype d'implémentation qui en est proposé. Certaines constructions particulières (la hiérarchisation de composants par exemple) et fonctionnalités avancées (la reconfiguration dynamique par exemple) lui font cruellement défaut. Toutefois, un des avantages du prototype actuel réside justement dans son extrême simplicité, qui procure au modèle de très bonnes performances.

Une nouvelle implémentation se doit d'apporter le minimum de restrictions au modèle de composition théorique, tout en offrant des performances équivalentes à celles du prototype actuel. Toute restriction au modèle (nécessaire pour des raisons de performance ou autre) doit être effectuée ultérieurement, au moment de la conception du noyau par le développeur lui-même. L'implémentation doit permettre la création de structures hiérarchiques de composants. Elle doit offrir les mécanismes autorisant le chargement, le remplacement ou la suppression d'un composant du système à l'exécution. Les structures de données nécessaires au stockage d'informations sur le système et les mécanismes permettant d'y accéder doivent également être définies avec précision.

Nous proposons dans le chapitre suivant une nouvelle implémentation du modèle théorique de composition de THINK. Celle-ci ne remet pas fondamentalement en cause les techniques performantes d'implémentation des notions

de composant du prototype actuel. Au contraire, elle se base sur celles-ci pour construire un modèle plus complet architecturalement et plus riche fonctionnellement.

# Chapitre 5

# Reconfiguration dynamique d'un noyau Think

Ce chapitre présente une nouvelle implémentation du modèle de composition de THINK. L'accent est mis sur les possibilités de reconfiguration dynamique d'un noyau, qu'elles soient structurelles ou fonctionnelles. Pour ce faire, nous nous sommes intéressés au canevas de composition Fractal [Con03] et à son éventuelle utilisation comme canevas de reconfiguration d'un noyau THINK.

## 5.1 Introduction

Les possibilités fonctionnelles que laisse entrevoir le modèle théorique de composition de THINK sont particulièrement intéressantes. Toutefois, son implémentation actuelle n'exploite que partiellement les capacités du modèle. En particulier, aucune fonction de reconfiguration dynamique n'est proposée.

Les travaux présentés dans ce chapitre ont pour objectif la définition d'une nouvelle implémentation du modèle de composition offrant des fonctions de reconfiguration dynamique. Nous proposons pour ce faire de définir et d'intégrer au sein de l'implémentation actuelle un canevas d'inspection et de reconfiguration s'appuyant sur le modèle théorique de composition de THINK.

### 5.1.1 Que rendre reconfigurable?

Avant d'aller plus loin dans la présentation du canevas de reconfiguration dynamique, il est nécessaire de définir avec précision ce que l'on souhaite pouvoir inspecter et reconfigurer.

Dans un système à composants, deux types de reconfiguration existent, la reconfiguration comportementale et la reconfiguration structurelle.

La reconfiguration comportementale consiste à remplacer un module donné du système implémentant un fonctionnalité particulière par un module différent implémentant la même fonctionnalité (mais d'une manière différente), sans toucher au reste du système. Une reconfiguration de ce type laisse invariante la structure du système. Seul le module remplacé sera affecté par une opération de reconfiguration.

La reconfiguration structurelle, quant à elle, consiste à modifier la structure même du système, c'est à dire les relations de liaison ou d'appartenance entre les composants. Ce type de reconfiguration n'est pas forcément conservatrice des comportements du système. Bien souvent, elle implique de profonds changements dans la nature même du système reconfiguré.

Disposer de mécanismes de reconfiguration comportementale suffit lorsque le système est conçu pour être invariant fonctionnellement et quand les seuls modifications prévues concernent l'implémentation de certaines politiques du système. Toutefois, il est certains cas où l'on ne peut se passer de modifications structurelles : c'est déjà nécessaire lorsque l'on désire rajouter des fonctionnalités à un système.

L'architecture THINK doit autoriser les reconfigurations comportementales aussi bien que les reconfigurations structurelles. Cela implique notamment les opérations suivantes sur les composants :

**Ajout d'un composant au sein d'un composite** Un composite doit pouvoir, si nécessaire, autoriser l'environnement à manipuler son contenu en ajoutant certains composants.

**Suppression d'un composant au sein d'un composite** De façon symétrique, un composite doit pouvoir autoriser l'environnement à manipuler son contenu en supprimant certaines composants.

**Remplacement d'un composant** Tout composant d'un système doit pouvoir être remplacé par un composant fonctionnellement identique (c'est à dire offrant les mêmes interfaces que son prédécesseur). Le remplacement d'un composant doit idéalement être effectué en conservant l'état interne du composant remplacé.

Ces trois opérations nous suffisent à mener n'importe quelle opération de reconfiguration. En effet, toute entité d'un système THINK est un composant. Dès lors, si l'on dispose des opérations de reconfiguration énumérées ci-dessus, toute entité d'un système THINK est reconfigurable. Cette affirmation inclut même les liaisons, qui sont elles-mêmes des composants à part entière.

L'application systématique d'un modèle de reconfiguration à chacun des composants d'un système THINK peut toutefois être préjudiciable pour les per-

formances globales du système. Il est ainsi préférable d'utiliser un schéma mixte, dans lequel les mécanismes de reconfiguration sont associés de façon discrète et explicite aux composants d'un noyau. Ainsi, les composants ne nécessitant pas de reconfiguration ne sont pas pénalisés.

### 5.1.2 Qui engage une reconfiguration ?

Pour définir un canevas de reconfiguration cohérent, il est nécessaire de définir quelles entités prendront la responsabilité de démarrer une reconfiguration du système.

Trois stratégies sont envisageables :

1. offrir à l'utilisateur final du système une interface de reconfiguration;
2. désigner une entité responsable de toute reconfiguration du système;
3. laisser la possibilité à n'importe quelle entité du système de démarrer une reconfiguration sur la partie du système qui lui est visible.

La première stratégie consiste à offrir à l'utilisateur final tous les outils lui permettant de pratiquer les reconfigurations qu'il juge nécessaire sur le système (le terme d'"utilisateur" qualifie ici toute application s'exécutant au-dessus du système). L'utilisateur manipule une représentation abstraite du système et toute modification effectuée sur cette représentation doit se répercuter automatiquement, par réflexion, sur le système lui-même. Cette solution pose néanmoins quelques problèmes. Tout d'abord, il sera difficile de proposer à l'utilisateur une vue entière du système. Si cela s'avère possible, il est ensuite nécessaire d'assurer que l'utilisateur ne risque pas de corrompre le système en le reconfigurant. Or, le développeur du système ne maîtrise pas l'utilisateur et aura donc tendance à lui offrir peu de possibilités de reconfiguration.

La deuxième stratégie consiste à désigner une autorité responsable de toutes les reconfigurations du système (cette autorité pouvant par exemple prendre la forme d'un service système clairement identifié). L'autorité en question dispose de tous les droits de modification du système. Elle s'assure de toujours effectuer des reconfigurations cohérentes. Elle a la possibilité de fournir aux utilisateurs une interface leur permettant d'effectuer des reconfiguration limitées. Cette solution reste toutefois difficile à mettre en œuvre concrètement. Il est difficilement envisageable de développer un service ayant la connaissance suffisante pour pratiquer potentiellement n'importe quelle reconfiguration sur n'importe quelle partie du système.

La troisième et dernière stratégie consiste à ne pas faire de supposition a priori sur l'entité responsable d'une reconfiguration. Chaque composant du sys-

tème susceptible d'être reconfiguré implémente lui-même les fonctions de reconfiguration que l'on peut lui appliquer et les offre à l'environnement sous la forme d'une interface standardisée. Tout autre composant du système qui dispose d'une vue de cette interface et peut s'y lier a le droit d'appeler les opérations qu'elle définit. Cette dernière solution présente l'avantage d'éclater l'entité responsable d'une reconfiguration en de multiples entités distinctes, chacune tenant un rôle local. En outre, chaque composant gère lui-même sa reconfiguration interne. Enfin, le droit de reconfiguration est implicitement lié à l'accessibilité d'une interface, ce qui le rend facilement contrôlable.

Nous avons opté pour la dernière stratégie, qui nous permet de mener des reconfigurations à grain extrêmement fin, et de façon beaucoup plus sûre. Ainsi, dans un système THINK, chaque composant reconfigurable devra offrir à l'environnement une ou plusieurs interfaces de reconfiguration.

### 5.1.3 A quel moment pratiquer une reconfiguration?

Le moment où se produit une reconfiguration a une grande importance. Il se peut en effet que certains composants du système se trouvent, à un instant donné, dans un état qui ne leur permet pas d'être reconfigurés. Cela se produit notamment lorsqu'un processus est en cours d'exécution et manipule l'état interne du composant à reconfigurer. Le remplacement du composant et le transfert de l'état de l'ancien composant vers le nouveau n'est alors pas possible.

La reconfiguration d'un composant doit donc avoir lieu lorsque le composant est dans un état stable et globalement cohérent. Or, la détermination d'un tel état est très dépendante de la sémantique même du composant. Il semble ainsi naturel que chaque composant reconfigurable offre lui-même une méthode qui, après avoir été exécutée, assure que l'état du composant permet sa reconfiguration.

Dans un système THINK, chaque composant reconfigurable devra offrir à l'environnement une interface permettant son arrêt et son redémarrage. Chaque composant pourra implémenter cette interface comme il le souhaite : la sémantique associée à l'arrêt et au redémarrage d'un composant ne sera pas imposée par le modèle.

## 5.2 Stratégies de conception

L'implémentation du modèle de reconfiguration au sein de THINK suit certaines stratégies de conception que l'on expose dans cette section.

### 5.2.1 Adopter un modèle contrôleur/ contenu

Le modèle contrôleur / contenu suppose l'existence d'une "coque" active (c'est à dire une coque susceptible de contenir du code) autour d'un module logiciel (voir figure 5.1). La coque est appelée "contrôleur", et ce qu'elle encapsule "contenu". Le tout forme un composant.

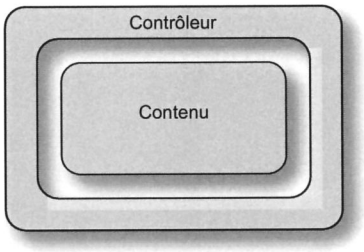


FIG. 5.1 – *Le modèle contrôleur / contenu*

Dans le cas d'un composant composite, le contenu est constitué d'un ensemble de composants d'ordre inférieur. Dans le cas d'un composant primitif, le contenu est constitué de code métier (sans restriction sur sa nature : code C, structure complexe d'objets Java, etc.). Dans les deux cas, la coque peut être elle-même un composant ou un assemblage complexe de plusieurs composants.

Le contrôleur joue deux rôles distincts :

**Administration du contenu** Seul le contrôleur dispose de droits d'administration sur le contenu qu'il encapsule. Ces droits incluent notamment la possibilité de modifier les liaisons entre les composants du contenu et l'ajout, la suppression ou le remplacement d'un composant du contenu.

**Interception des interactions** Le contrôleur peut intercepter toutes les interactions (entrantes ou sortantes) entre l'environnement et le contenu qu'il administre. Le contrôleur dispose de tous les droits sur les interactions qu'il intercepte, et en particulier celui de les surcharger ou de les ignorer.

Le modèle contrôleur / contenu possède plusieurs atouts qui le rendent très intéressant dans la perspective de son utilisation pour l'implémentation d'un système Think.

En premier lieu, ce modèle permet une séparation explicite du code fonctionnel du composant (le contenu) et de son code d'administration (le contrôleur). Cette distinction joue en faveur de l'indépendance des comportements fonctionnels vis à vis de leurs fonctions d'administration. Le développement de composants est ainsi plus aisé.

Ce modèle introduit ensuite un moyen de construire des hiérarchies de composants. En effet, l'ensemble contrôleur / contenu forme un composant. Or, comme il est dit plus haut, le contenu peut être une composition plus ou moins complexe de composants. Le contrôleur est donc un moyen de factoriser plusieurs composants en un seul composant d'ordre supérieur.

Enfin, l'utilisation d'un contrôleur offre le support au développement d'opérations d'administration avancées. Ses rôles d'administration et d'interception lui donnent en effet un fort pouvoir de contrôle sur le contenu. En outre, le contrôleur décide de la vue abstraite qu'il offre de son contenu à l'extérieur, ce qui lui permet d'en cacher si besoin certaines parties.

### 5.2.2 Profiter des avantages de l'existant

L'implémentation originale du modèle de composition de THINK permet la construction de compositions à plat très performantes. Le coût des liaisons primitives entre composants, notamment, est quasi-nul. Ainsi, plutôt que de refaire une implémentation de THINK de la base, il nous est apparu intéressant de conserver la majeure partie de l'implémentation actuelle et de l'étendre afin de lui permettre de supporter les compositions hiérarchiques et la reconfiguration dynamique. Il restait toutefois à déterminer la manière dont cette extension allait s'opérer.

Une des propriétés cruciales en matière de configurabilité concerne l'uniformité d'un système. Disposer d'un seul et même modèle de composition à tous les niveaux d'un ensemble logiciel (des plus basses couches du système aux couches les plus abstraites des applications, en passant par les éventuels intergiciels) permet à chaque couche d'intervenir directement sur toutes les autres selon un processus unifié et connu de tous. THINK fait partie intégrante du consortium ObjectWeb, qui propose justement une solution à l'hétérogénéité des systèmes et des applications et à leurs besoins de reconfiguration. Cette solution prend la forme d'un ensemble de plates-formes logicielles; toutes reposent (ou reposeront dans un avenir proche) sur un même modèle de composition appelé Fractal.

Le modèle de composition Fractal s'inspire, tout comme celui de THINK, des travaux de recherche menés par Gordon Blair et Jean-Bernard Stefani dans le cadre de la définition du standard RM-ODP [X.995] et de la plate-forme logicielle de communication Jonathan [Kra02]. En plus d'un modèle théorique de composition hiérarchique et de reconfiguration dynamique, il propose un ensemble d'interfaces de programmations réifiant chacun des concepts du modèle. Il n'est pas destiné à une classe particulière de logiciels : systèmes, intergiciels ou applications peuvent en bénéficier.

Nombre d'arguments plaidaient en faveur de l'intégration de Fractal au sein de l'implémentation originale de THINK. L'intégration du modèle de composition Fractal au sein de THINK revêt le double avantage de faire bénéficier THINK des apports de Fractal en matière de reconfiguration dynamique tout en préservant la compatibilité avec des technologies intergicielles et applicatives d'ores et déjà développées selon ce modèle. La conservation du modèle original permet en outre d'espérer de bonnes performances du produit final.

## 5.3 Le canevas de composition Fractal

Cette section présente Fractal, un canevas de composition développé par France Télécom R&D au sein du consortium ObjectWeb. Cette présentation s'attarde sur les aspects du modèle montrant un intérêt direct dans le cadre d'une intégration de Fractal à Think; la documentation officielle de Fractal [Con03] contient une description détaillée.

### 5.3.1 Objectifs du canevas

Le consortium ObjectWeb regroupe des plates-formes logicielles permettant le développement d'applications à tous les niveaux d'un système. Historiquement, toutes se basaient sur un modèle de composition logicielle, mais chacune définissait son propre modèle de composition (bien que tous étaient plus ou moins identiques du point de vue conceptuel).

Fractal est une tentative d'unification des différents modèles de composition des plate-formes logicielles développées au sein du consortium. Le modèle intègre de plus les structures exécutives permettant de reconfigurer dynamiquement les systèmes et applications bâties selon ses règles.

### 5.3.2 Modèle conceptuel de composition

Le modèle conceptuel de composition de Fractal est très similaire à celui de THINK[1] (voir section 4.3). Toutefois, il comporte un certain nombre de constructions spécifiques et de restrictions visant à faciliter son implémentation.

Un composant Fractal est une unité logicielle composée de deux parties : un contrôleur et un contenu. Le contenu d'un composant est formé d'un nombre fini d'autres composants (appelés sous composants), sous le contrôle du contrôleur de leur composite englobant (voir figure 5.2). Le modèle est ainsi récursif et le

1. Cette similitude rend par ailleurs l'intégration des technologies Fractal et THINK très naturelle.

niveau de récursivité est arbitraire. La récursion prend fin avec les composants primitifs.

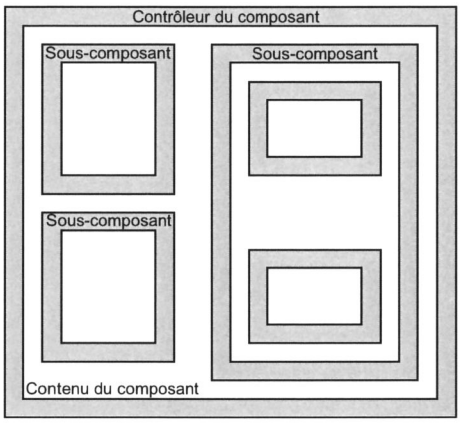


FIG. 5.2 – *Un exemple de composant Fractal*

Les contenus respectifs de différents composants peuvent être égaux. En d'autres termes, un composant peut être partagé par plusieurs composites (voir figure 5.3). Un composant partagé par deux composants ou plus est soumis au contrôle de chacun des contrôleurs des composites englobant. La sémantique exacte de la configuration résultante est en général déterminée par le composant englobant tous les composants appartenant à cette configuration.

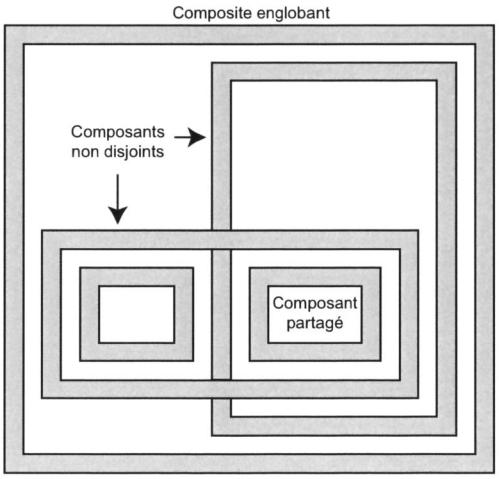


FIG. 5.3 – *Un exemple de composant partagé Fractal*

L'interaction d'un composant avec son environnement s'effectue par l'invocation d'opérations sur des interfaces identifiées. La visibilité de l'interface d'un sous-composant à l'intérieur et à l'extérieur du composite est déterminée par

le contrôleur englobant. Un contrôleur peut ainsi cacher l'interface d'un de ses sous-composants, ou choisir au contraire de l'exporter explicitement vers son environnement. Les interfaces d'un composant regroupent donc les interfaces externes (qui ne sont visibles que par l'environnement), les interfaces exportées de ses sous-composants et les interfaces internes (qui ne sont visibles que par ses sous-composants). La figure 5.4 expose les différents niveaux de visibilité d'une interface Fractal.

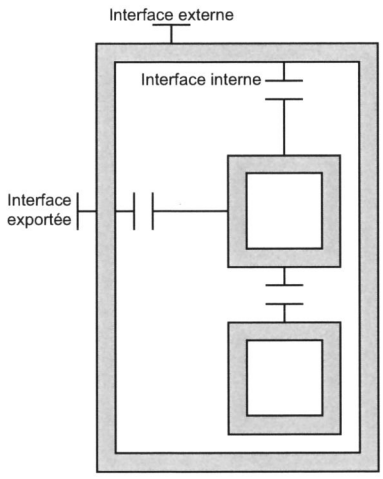


FIG. 5.4 – *Visibilité des interfaces Fractal*

Il existe deux types d'interfaces : les interfaces client et les interfaces serveur. Une interface serveur réceptionne les appels de méthodes émis par son environnement, alors qu'une interface client émet des appels de méthode.

Le contrôleur d'un composant définit un comportement d'administration. A ce titre, il est capable d'intercepter les appels entrant et sortant, de fournir une représentation causalement connectée[2] de son contenu et de superposer certains comportements aux comportements des composants qu'il encapsule.

### 5.3.3 Interfaces de contrôle

Le modèle de composition et de reconfiguration de Fractal se traduit par un ensemble d'interfaces de programmation. Parmi celles-ci, certaines sont qualifiées d'interfaces de contrôle.

Les interfaces de contrôle permettent d'appliquer diverses opérations d'inspection et de reconfiguration d'un composant. Elles sont au nombre de quatre. Chacune regroupe des méthodes spécifiques à un aspect précis du composant.

2. Pour une explication de ce qu'est une représentation causalement connectée, veuillez vous référer aux travaux de Pattie Maes sur la réflexivité [MN88].

On trouve ainsi les interfaces `ComponentIdentity` (qui permet l'identification d'un composant), `LifeCycleController` (qui permet de contrôler le cycle de vie d'un composant), `ContentController` (qui offre le moyen de manipuler le contenu d'un composant) et `BindingController` (qui permet l'établissement de liaisons entre le composant et son environnement). Un composant peut fournir chacune de ces interfaces, mais aucune n'est obligatoire à l'exception de l'interface `ComponentIdentity` qui doit être systématiquement offerte par tous les composants d'un système Fractal (voir figure 5.5).

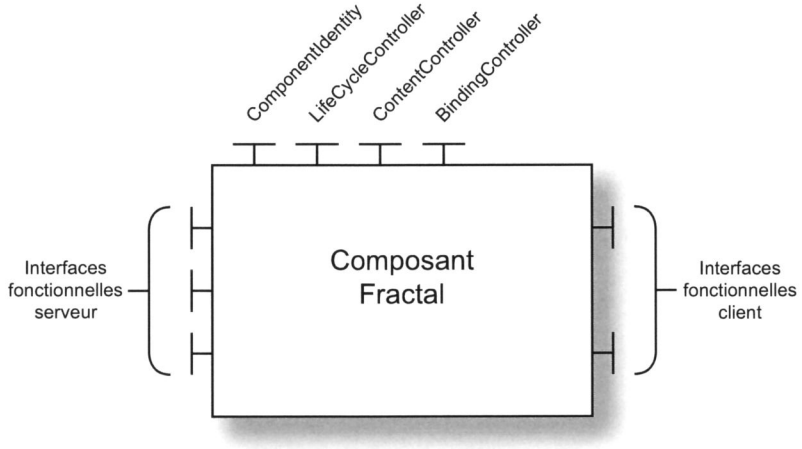


FIG. 5.5 – *Interfaces de contrôle d'un composant Fractal*

Les sections suivantes présentent les attributions de chacune des interfaces de contrôle. Les autres interfaces du modèle ne sont pas présentées ici, mais il est possible de les étudier en détails sur [Con03].

#### 5.3.3.1 Interface `ComponentIdentity`

L'interface `ComponentIdentity` permet d'identifier un composant de façon non ambiguë. En effet, pour offrir des outils de reconfiguration dynamique, il est nécessaire de pouvoir identifier les interfaces d'un composant et le composant lui-même. Dans le modèle Fractal, une une interface est identifiée par son nom et un composant est identifié par son interface `ComponentIdentity`.

Au travers de l'interface `ComponentIdentity` d'un composant, il est possible de récupérer les interfaces produites par le composant et de connaître son type. L'interface fournit ainsi les méthodes `getInterface`, `getInterfaces` et `getType` (voir listing 5.1).

Listing 5.1: *Interface* `ComponentIdentity`

```
interface ComponentIdentity {
    InterfaceReference getInterface(String interfaceName);
    InterfaceReference[] getInterfaces();
    Type getType();
}
```

#### 5.3.3.2 Interface `LifeCycleController`

L'interface `LifeCycleController` offre les primitives permettant de contrôler le cycle de vie du composant auquel elle appartient. Le cycle de vie d'un composant est supposé correspondre à un automate dont chaque état identifie un état d'exécution du composant. Dans le cas standard, le cycle de vie correspond à un automate à deux états, `STARTED` et `STOPPED`, et aux quatre transitions possibles entre ces états (voir figure 5.6).

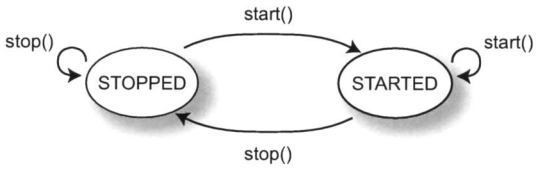


FIG. 5.6 – *Le cycle de vie d'un composant Fractal*

La sémantique des états `STARTED` et `STOPPED` est directement liée aux activités s'exécutant au sein du composant. En règle générale, dans l'état `STARTED` un composant peut recevoir et émettre des appels vers d'autres composants. Dans l'état `STOPPED` au contraire, un composant ne peut émettre d'appels vers d'autres composants et ne peut recevoir d'appels sur ses interfaces fonctionnelles.

L'interface `LifeCycleController` définit trois méthodes, `start()`, `stop()` et `getState()`, permettant respectivement de démarrer un composant, de l'arrêter, et de récupérer une abstraction de l'état d'instance du composant (voir listing 5.2).

Listing 5.2: *Interface* `LifeCycleController`

```
interface LifeCycleController {
    String getState();
    void start();
    void stop();
}
```

#### 5.3.3.3 Interface `ContentController`

L'interface `ContentController` permet de contrôler le contenu d'un composant composite. Il est dit plus haut que le contenu d'un composant est lui-même composé d'un ensemble de composants et que le modèle est hiérarchique. Un contrôleur de contenu permet de maîtriser la hiérarchie de composants.

L'interface `ContentController` définit trois primitives, respectivement nommées `addSubComponent`, `removeSubComponent` et `getSubComponents` (voir listing 5.3). Grâce à ces primitives, il est possible d'ajouter ou de supprimer un sous-composant, et d'obtenir l'ensemble des sous-composants d'un composite. La sémantique associée aux opérations d'ajout et de suppression d'un composant n'est pas inhérente au modèle.

Listing 5.3: *Interface* ***`ContentController`***

```
interface ContentController.java {
    void addSubComponent();
    void removeSubComponent();
    ComponentIdentity[] getSubComponents();
}
```

#### 5.3.3.4 Interface `BindingController`

L'interface `BindingController` est utilisée pour manipuler dynamiquement les liaisons entre un composant et son environnement. Un composant proposant cette interface offre les outils permettant à d'autre composants de créer des liaisons avec l'une de ses interfaces et de supprimer des liaisons existantes.

L'interface `BindingController` définit trois méthodes, `bind`, `unbind` et `lookup`, permettant respectivement de créer une liaison vers une interface serveur du composant, de supprimer une liaison existante et d'obtenir la référence d'une interface à partir d'une liaison (voir listing 5.4).

Listing 5.4: *Interface* ***`BindingController`***

```
interface BindingController {
    String bind(String clientItfName, InterfaceReference serverItf
        );
    InterfaceReference lookup(String clientItfName);
    void unbind(String clientItfName);
}
```

## 5.4 Intégration du modèle Fractal à THINK

Les modèles conceptuels de composition de Fractal et de THINK sont très similaires, mais leurs traductions en un modèle d'exécution ne le sont pas. Fractal propose (sous forme d'un ensemble de structures d'exécution et d'interfaces de programmation) un vrai support d'exécution à la reconfiguration dynamique, chose qui n'est pas fournit par le prototype de THINK.

L'intégration des fonctionnalités de reconfiguration de Fractal au sein même de THINK engendre un certain nombre de modifications du modèle d'exécution de THINK. Cette section présente les évolutions principales du modèle.

### 5.4.1 Comportement de contrôle

La structure d'un composant THINK reconfigurable reprend celle du prototype initial de THINK en y ajoutant les interfaces de contrôle Fractal. Comme avec le modèle Fractal, l'interface `ComponentIdentity` doit être fournie par tous les composants du système, les autres interfaces restant optionnelles.

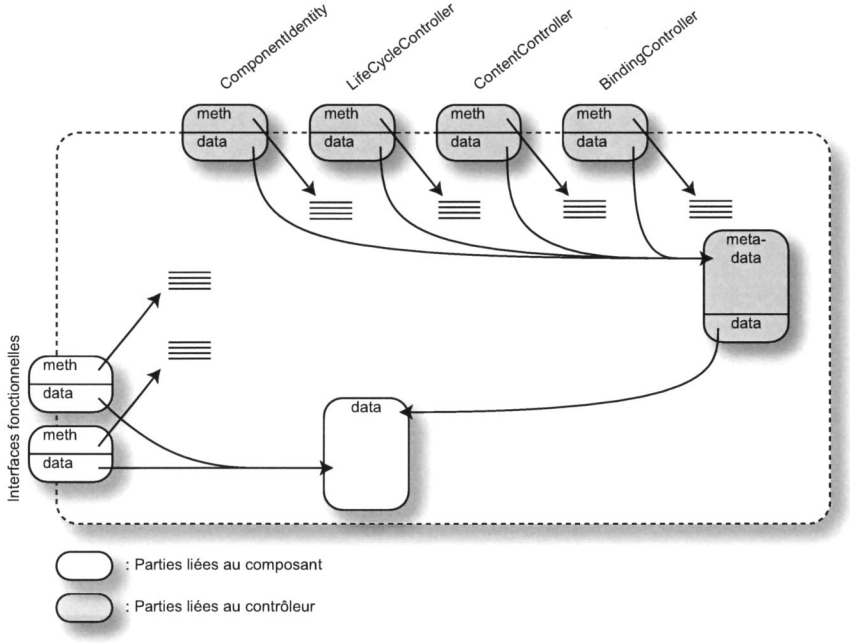


FIG. 5.7 – *Un composant* THINK *reconfigurable*

Le comportement de contrôle d'un composant THINK est centralisé au niveau d'un contrôleur (voir figure 5.7). Le contrôleur est un composant THINK à part entière et dispose ainsi de ses propres données d'instance. En effet, certaines fonctions fournies par les interfaces de contrôle d'un composant THINK

nécessitent des connaissances sur le composant lui-même. Par exemple, il est nécessaire de garder en mémoire la liste des interfaces fournies par un composant pour implémenter les méthodes de l'interface `ComponentIdentity`. Ces méta-informations, comme toutes les autres informations liées au contrôle du composant, sont stockées dans les données d'instance du contrôleur.

De la même façon qu'une interface fonctionnelle d'un composant travaille sur les données d'instance du composant, une interface de contrôle travaille ainsi sur les données de contrôle du composant. Ainsi, le champ `data` de chaque interface de contrôle d'un composant pointe sur la structure contenant les méta-données.

Certaines méthodes de contrôle doivent manipuler directement les données d'instance du composant. C'est par exemple le cas de la méthode `getState` de l'interface `LifeCycleController` qui, lorsqu'elle est appelée, retourne une valeur correspondant à l'état d'instance du composant. Pour ce faire, les données de contrôle contiennent un pointeur vers les données d'instance du composant. Ainsi, toutes les interfaces de contrôle ont accès, en plus des données de contrôle, aux données d'instance du composant.

L'ensemble formé du composant THINK contrôlé et du composant contrôleur constitue un composant Fractal, c'est à dire un composant THINK reconfigurable.

### 5.4.2 Interception des interactions

Les interactions avec les interfaces fonctionnelles d'un composant peuvent s'effectuer de manière directe, ou être interceptées par le contrôleur du composant.

Dans le cas où il intercepte des interactions fonctionnelles, le contrôleur doit mettre en place des copies des interfaces fonctionnelles (voir figure 5.8). Tout appel à une interface fonctionnelle s'effectue en réalité sur l'interface dupliquée. Un code d'interception (appelé intercepteur), fournit par le contrôleur, est alors exécuté en lieu et place du code fonctionnel. Un appel au code fonctionnel peut toutefois être effectuée par le contrôleur au moment du traitement de l'interception.

Un contrôleur peut toutefois décider de ne pas intercepter les appels vers son contenu. Dans ce cas, l'appel à une interface fonctionnelle est effectué directement, sans passer par le contrôleur. Aucun code lié au contrôleur n'est alors exécuté.

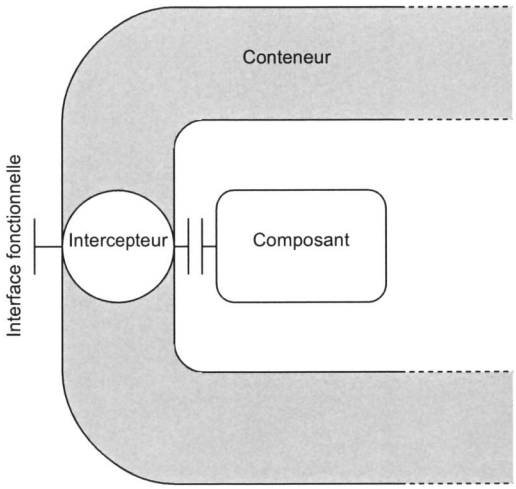


FIG. 5.8 – *Interception des interactions*

### 5.4.3 Conséquences sur les interfaces de programmation

L'intégration de Fractal au sein de Think se traduit également par une évolution des interfaces de programmation du modèle de composition.

#### 5.4.3.1 Adaptation des interfaces Fractal

Afin de conserver la compatibilité avec le modèle initial de composition de THINK, les interfaces de programmation Fractal ont dû subir quelques modifications. Pour la plupart, ces modifications ne sont que des adaptations de type et ne changent en rien la structure des interfaces. Seule l'interface `LifeCycleController` se voit agrémentée d'une nouvelle méthode, `setState` (son rôle sera expliqué dans la suite du document).

Le listing 5.5 présente l'adaptation des interfaces de contrôle Fractal à THINK.

#### 5.4.3.2 Intégration d'un système de type

Lors d'une reconfiguration dynamique, il est nécessaire de vérifier certaines contraintes d'intégrité. Par exemple, lors du remplacement d'un composant, il faut s'assurer que le nouveau composant et l'ancien sont de deux types compatibles.

Le concept de type est réifié par la nouvelle interface de programmation `Type` (voir listing 5.6). Cette interface offre deux méthodes, `equals` et `isSubTypeOf`.

L'opération `equals` teste si le type représenté par le composant offrant l'interface `Type` est égal à un type passé en paramètre. L'opération isSubTypeOf

Listing 5.5: *Adaptation des interfaces Fractal à* ***Think***

```
interface ComponentIdentity {
    Top getInterface(Name interfaceName);
    Top[] getInterfaces();
    Type getType();
}

interface LifeCycleController {
    String getState();
    void setState(String state);
    void start();
    void stop();
}

interface ContentController.java {
    void addSubComponent();
    void removeSubComponent();
    ComponentIdentity[] getSubComponents();
}

interface BindingController {
    void bind(Name clientItfName, Top serverItf);
    Top lookup(Name clientItfName);
    void unbind(Name clientItfName);
}
```

Listing 5.6: *Interface* ***Type***

```
interface Type {
    boolean equals(Type type);
    boolean isSubTypeOf(Type type);
}
```

teste si le type représenté par le composant offrant l'interface `Type` est un sous-type du type passé en paramètre.

#### 5.4.3.3 Une nouvelle interface : `ComponentFactory`

Afin de pouvoir charger, décharger ou remplacer un composant du système dynamiquement, il est nécessaire de strictement séparer le code d'instanciation d'un composant de son code métier. Le modèle conceptuel de Think prévoit pour cela le concept de fabrique de composant : une fabrique de composant instancie les composants d'un certain type (voir section 4.3.5).

Le concept de fabrique de composant n'est appliqué que de manière partielle dans le prototype actuel de THINK : les seules fabriques existantes sont les fabriques de liaison et les fabriques de nom. De plus, aucune interface de programmation ne régit l'implémentation d'une fabrique.

Nous proposons ainsi d'ajouter au modèle exécutif une interface de programmation supplémentaire, réifiant le concept de fabrique de composant du modèle conceptuel. Cette interface se nomme `ComponentFactory` et offre une unique méthode, `instantiate` (voir listing 5.7).

Listing 5.7: *Interface* `ComponentFactory`

```
interface ComponentFactory {
    ComponentIdentity instanciate();
}
```

La méthode `instanciate` crée une instance d'un composant d'un certain type. La procédure d'instanciation crée les structures de données (variables d'instance et méta-données), instancie chacune des interfaces (fonctionnelles et de contrôle) et renvoie une référence vers l'interface `ComponentIdentity` du composant. Le composant peut être ajouté à un composite, lié à d'autres composants et si besoin (c'est à dire s'il comporte une interface de type `LifeCycleController`) démarré.

La méthode `instanciate` permet également d'instancier des composants composites (autrement dit, des hiérarchies de composants). Dans ce cas, en plus des simples mises en place des structures exécutives du composant et de ses interfaces, un appel à `instanciate` génère des appels aux fabriques de composants correspondant à chacun des sous-composants du composite. L'instanciation de la hiérarchie complète se poursuit de façon récursive jusqu'aux composants primitifs.

## 5.5 Implémentation des fonctions de reconfiguration

La similitude des modèles conceptuels de THINK et de Fractal laisse penser que leur intégration peut s'effectuer à moindre coût et sans compromettre la cohérence globale du modèle de composition de THINK. En réalité, l'adaptation de mécanismes de reconfiguration au sein même de noyaux de systèmes d'exploitation impose certaines contraintes d'implémentation.

Cette section présente une implémentation par défaut des mécanismes de reconfiguration de Fractal intégrés à THINK. Cette implémentation est proposée comme une aide à la construction de systèmes reconfigurables, mais son utilisation n'est bien entendu pas obligatoire. Le développeur d'un composant peut tout à fait, s'il le désire, proposer sa propre implémentation de chacun des mécanismes de reconfiguration du modèle.

### 5.5.1 Identification d'un composant

La gestion de l'identification d'un composant est assurée par l'interface de contrôle `ComponentIdentity`. Cette interface définit les méthodes `getInterface` (qui récupère une interface en fonction de son nom), `getInterfaces` (qui récupère la liste des interfaces du composant) et `getType` (qui récupère le type du composant).

Il est à la charge du contrôleur du composant de donner accès aux interfaces du composant (qu'elles soient fonctionnelles ou non). Le contrôleur maintient à cette fin une liste des interfaces qu'il fournit. Cette liste est stockée dans les données d'instance du contrôleur (c'est à dire dans les méta-informations du composant). Elle est formée d'un ensemble de triplets (nom, référence d'interface, type de l'interface).

La méthode `getInterface` recherche dans la liste le couple dont le nom correspond à celui fournit en paramètre et renvoie la référence d'interface correspondante. La méthode `getInterfaces`, elle, renvoie l'ensemble des références d'interface du composant.

La méthode `getType` instancie et retourne un composant `Type` (c'est à dire offrant une interface de type `Type`). Le type d'un composant est caractérisé par l'ensemble des types de ses interfaces. Le type d'une interface est caractérisé par l'ensemble des signatures des méthodes qu'elle définit. Ainsi, si `C1` et `C2` sont deux composants :

- `C1` et `C2` sont de même type si et seulement si ils offrent exactement les mêmes interfaces;

- `C1` est d'un sous-type du type de `C2` si `C2` fournit au moins les interfaces de `C1`.

Le composant Type renvoyé lors d'un appel à `getType` implémente ces deux relations de typage.

### 5.5.2 Gestion du cycle de vie d'un composant

La gestion du cycle de vie d'un composant est assurée par l'interface de contrôle `LifeCycleController`. Deux types d'opérations sont fournies : celles liées au cycle de vie même du composant, et celles liées à la manipulation de l'état d'instance du composant.

#### 5.5.2.1 Démarrage, arrêt et reprise d'un composant

Comme un composant Fractal, un composant THINK peut être dans l'un des deux états d'exécution `STARTED` ou `STOPPED`. Le passage d'un état à l'autre est assuré par les méthodes `stop` et `start`. Nous associons les sémantiques suivantes à ces états :

**État** `STARTED` **:** Un composant dans cet état est susceptible d'émettre et de recevoir des appels de méthode sur chacune de ses interfaces, qu'elles soient fonctionnelles ou de contrôle. L'état d'instance d'un composant dans l'état `STOPPED` n'est pas stable; au contraire, il évolue constamment en fonction des appels de méthode sur ses interfaces.

**État** `STOPPED` **:** Un composant dans cet état n'émet aucun appel de méthode vers l'extérieur. Les appels reçus ne sont pas exécutés mais sont enregistrés dans l'attente du redémarrage du composant. L'état d'instance d'un composant dans l'état `STARTED` est stable.

Un composant fournissant l'interface `LifeCycleController` possède une structure et un comportement particulier. Il comporte notamment un compteur de processus dont la valeur représente le nombre de processus en cours d'exécution sur le composant. Ce compteur est incrémenté avant et décrémenté après chaque appel de méthode.

Un appel à la méthode `stop` se déroule de la façon suivante :

1. Des intercepteurs sont mis en place derrière chaque interface fonctionnelle : chaque table des méthodes est remplacée par une table adhoc qui enregistre tout nouvel appel en vue de son exécution ultérieure.
2. L'exécution est ensuite mise en attente jusqu'à ce qu'aucun processus ne soit en cours d'exécution sur le composant (c'est à dire jusqu'à ce que la valeur du compteur de processus soit égale à zéro).

3. Le composant est mis dans l'état `STOPPED` et l'appel à `stop` se termine.

Une fois stoppé, un composant se contente d'enregistrer tout appel sur ses interfaces fonctionnelles en vue de leur exécution ultérieure. Tout appel à `stop` est ignoré. Le composant ne peut être redémarré que par un appel à `start`.

Un appel à la méthode `start` se déroule de la façon suivante :

1. Les intercepteurs mis en place lors de l'arrêt du composant sont supprimés.
2. Les éventuels appels mis en attente sont exécutés.
3. Le composant est mis dans l'état `STARTED` et l'appel à `start` se termine.

Une fois (re)démarré, les appels vers les interfaces fonctionnelles du composant s'exécutent normalement. Tout appel à `start` est ignoré. Le composant ne peut être arrêté que par un appel à `stop`.

#### 5.5.2.2 Sauvegarde et restauration de l'état d'un composant

En plus des opérations classique de gestion du cycle de vie d'un composant, l'interface `LifeCycleController` offre un moyen de manipuler l'état d'instance d'un composant au travers des méthodes `getState` et `setState`.

Bien évidemment, il est très délicat (voire impossible) de caractériser de façon générique et détachée de toute considération sémantique l'état d'instance d'un composant. Seul le développeur d'un composant est à même de nous informer des informations devant être inclues au sein d'une représentation sérialisée de ce composant. Plusieurs moyens d'expression sont envisageables :

- l'utilisation d'un langage de programmation de composants "décoré", permettant de spécifier explicitement les données appartenant ou n'appartenant pas à l'état d'instance d'un composant par l'emploi d'une marque spécifique (une "décoration") en début de déclaration de variable (de façon équivalente au mot clé `volatile` du langage Java);
- l'utilisation de descripteurs d'état associés aux composants, permettant de spécifier dans un langage dédié (XML ou autre) la composition de l'état du composant.

La première solution n'est pas satisfaisante dans notre cas, car elle impose l'utilisation d'un langage de programmation, qui plus est non standard. Nous avons donc opté pour la seconde solution. Ainsi, chaque composant de notre système est accompagné d'une description XML spécifiant la composition de l'état d'instance d'un composant (le langage de description utilisé sera étudié

plus en détails dans le chapitre ??). Au cours de la phase de conception du composant, cette description est analysée et elle sert à la génération des méthodes de capture et de restauration d'état `getState` et `setState`.

La méthode `getState` permet de récupérer une forme sérialisée de l'état du composant, et la méthode `setState` permet, à partir d'une forme sérialisée d'état, de mettre à jour l'état du composant. Spéicifiques à chaque composant, elles sont générées statiquement à partir de leur descripteur. Leur fonctionnement est des plus simple :

- la méthode `getState` lit chacune des valeurs des variables étant identifiées comme partie intégrante de l'état d'instance d'un composant et les enregistre séquentiellement au sein d'un tableau d'octets;
- la méthode `setState` affecte chacune des variables d'instance d'un composant à une valeur issue d'un tableau d'octets qui lui est passé en paramètre.

Bien sûr, afin de pratiquer des lectures (respectivement des écritures) cohérentes de l'état d'un composant, l'opération `getState` (respectivement `setState`) ne doit être exécutée que lorsque le composant se trouve dans l'état `STOPPED`.

### 5.5.3 Gestion du contenu d'un composant

La gestion du contenu d'un composant est assurée par l'interface de contrôle `ContentController`. Les trois opérations possibles sont l'ajout d'un sous-composant (avec l'opération `addSubComponent`), la suppression d'un sous-composant (avec l'opération `removeSubComponent`) et l'obtention de la liste des sous-composants (avec l'opération `getSubComponents`).

Le rôle d'un composant composite est d'administrer ses sous-composants. Ainsi, toute liaison entre composants est créée par leur composite englobant. De même, le composite est responsable de la visibilité des interfaces de ses sous-composants (c'est à dire de leur exportation directe au niveau du composite ou de leur appel indirect au travers d'intercepteurs par exemple). Les fonctionnalités d'administration attribuées au composite nécessitent la mise en place de mécanismes et de structures de données et particulières.

#### 5.5.3.1 Nouvelles structures de données

La relation d'appartenance à un composite se manifeste simplement, à l'exécution, par le maintient d'une référence au sous-composant dans le composite. En effet, un composant composite possède un lien vers l'interface `ComponentIdentity` de chacun de ses sous-composants. L'ensemble de ces liens constituent une liste,

stockée dans les informations d'instance du contrôleur. Cette liste est mise à jour à chaque modification du contenu du composant.

Les interfaces d'un sous-composant ne sont visibles qu'au sein du composite, sauf si celui-ci décide explicitement de les exporter vers son environnement. Dans ce cas, le composite doit déclarer une interface du type de l'interface exportée, qu'il fait pointer soit directement vers l'interface du sous-composant, soit vers un intercepteur qui redirigera l'appel vers le sous-composant après avoir effectué ses propres traitements.

#### 5.5.3.2 Ajout d'un sous-composant

L'ajout d'un composant au sein d'un composite comporte plusieurs étapes. En premier lieu, la référence de l'interface `ComponentIdentity` du composant à ajouter est insérée dans la liste des sous-composants. Les liaisons nécessaires à l'exécution du nouveau composant sont ensuite créées. Celles-ci comprennent les liaisons avec d'autres sous-composants du même niveau de hiérarchie, mais également les éventuelles liaisons avec le composite englobant. Enfin, les éventuelles interfaces devant être visibles à l'extérieur du composite englobant sont exportées par le composite. L'exportation d'une interface peut être directe (l'interface est alors directement ajoutée dans la liste des interfaces du composite), ou au contraire passer par un intercepteur (dans ce cas, une nouvelle interface est créée au niveau du composite, pointant sur un intercepteur qui fait appel aux méthodes de l'interface exportée).

Certaines contraintes peuvent empêcher l'ajout d'un composant. Par exemple, pour ajouter un composant au sein d'un composite, il est nécessaire de s'assurer que le composite fournit tous les services requis par le nouveau composant (soit sous forme d'interface fournie par un autre sous-composant, soit soir forme d'interface requise par le composite). Si tel est le cas, le composant n'est pas ajouté au composite.

#### 5.5.3.3 Suppression d'un sous-composant

L'opération de suppression d'un sous-composant supprime toutes les structures relatives à ce composant au sein du composite. Cela comprend notamment les entrées dans les listes d'interface et de sous-composants, les intercepteurs et les interfaces dupliquées.

Certaines contraintes peuvent empêcher la suppression d'un composant. Ainsi, si le composant devant être supprimé fournit un service qui ne peut être fournit par ailleurs, le composant ne peut être supprimé du composite.

Il est en outre important de comprendre que ce n'est pas parce qu'un com-

posant est supprimé d'un composite qu'il n'est plus présent en mémoire. Si tel est l'objectif souhaité, le développeur doit rajouter les instructions nécessaires au déchargement du composant.

#### 5.5.3.4 Obtention de la liste des sous-composants

A tout moment, il est possible d'obtenir la liste des sous-composants d'un composite. Cette liste est simplement constituée des références des interfaces de désignation `ComponentIdentity` de chacun des sous-composants.

### 5.5.4 Liaisons dynamiques

Grâce à l'interface `BindingController`, il est possible de contrôler dynamiquement les liaisons d'un composant. Les opérations permettent de créer une liaison vers une interface (avec l'opération `bind`), de supprimer une liaison vers une interface (avec l'opération `unbind`), et de retrouver l'interface à laquelle est liée une interface passée en paramètre (avec l'opération `lookup`). L'interface ne permet que de manipuler les liaisons des interfaces client du composant considéré.

La création d'une liaison dépend fortement de la sémantique même de la liaison. Toutefois, nous avons vu que toute liaison complexe peut être ramenée à un ensemble de composants reliés entre eux par des liaisons primitives. L'opération `bind` permet ainsi de créer une liaison primitive entre deux composants. Elle recherche l'interface client dont le nom lui est passé en paramètre, et l'associe à la référence d'une interface serveur qui lui est également passée en paramètre. A la suite de cette opération, tous les appels vers l'interface client seront automatiquement redirigés vers l'interface serveur.

Supprimer une liaison vers une interface client revient à invalider la référence de l'interface serveur qui lui est associé. Ainsi, l'opération `unbind` recherche la référence associée à l'interface client dont le nom est passé en paramètre et l'affecte à `NULL`.

L'opération `lookup` permet, à partir du nom d'une interface client, de récupérer la référence de l'interface serveur à laquelle elle est liée.

## 5.6 Bilan

Il est temps d'établir un premier bilan de la nouvelle implémentation de `Think` et des outils de reconfiguration dynamique qui y sont intégrés.

L'intégration au sein même de THINK du modèle de reconfiguration Fractal enrichit fortement l'architecture. Les systèmes construits à l'aide de la nouvelle

implémentation de Think sont évolutifs dynamiquement, et ce jusqu'aux couches les plus basses du système.

Le niveau de conservation des modèles initiaux est très important. Les principes de base de THINK ont tous été conservés, de même que l'implémentation, très performante, des liaisons primitives. Si, par contre, toutes les interfaces de Fractal n'ont pas été appliquées à Think, c'est dans un soucis d'efficacité et cela ne remet nullement en cause la compatibilité de l'architecture avec d'autres architectures Fractal.

Une analyse des coûts est toutefois nécessaire pour évaluer l'impact du modèle de reconfiguration sur les performances globales du système. L'expérience du développement d'un système complet où les besoins de reconfiguration sont clairement identifiés serait en cela extrêmement enrichissante, tout comme l'utilisation de l'architecture Think comme base de développement pour des systèmes hautement spécialisés tels que les systèmes embarqués. Cela fait l'objet du chapitre suivant.

# Chapitre 6

# Développement de systèmes d'exploitation pour architectures embarquées à l'aide de THINK

Les chapitres précédents ont présenté les concepts de l'architecture THINK et leur implémentation, ainsi que l'intégration d'un modèle de reconfiguration dynamique et l'implémentation de ce modèle. L'ensemble de ces développements ont tout d'abord été menés sur des architectures matérielles de type PowerMacintosh, afin de bénéficier de la souplesse du processeur PowerPC. Toutefois, la philosophie originale de THINK semble le destiner particulièrement aux architectures matérielles ultra-spécialisées, où la configuration optimale du noyau revêt une importance capitale. C'est notamment le cas de l'informatique embarquée, où de nombreuses contraintes pèsent sur le système d'exploitation.

Ce chapitre présente une utilisation de THINK dans le cadre de la construction de systèmes d'exploitation pour architectures matérielles embarquées. Cette utilisation se traduit par l'implémentation d'une nouvelle librairie de composants, appelée Embedded-THINK. Après une description des architectures matérielles et des applications logicielles que nous visons et des stratégies de conception que nous avons adopté, nous nous intéresserons à la librairie elle-même, tant d'un point de vue conceptuel que technique. Enfin, nous présenterons l'utilisation de cette librairie dans le cadre du développement d'un noyau pour une architecture matérielle précise : le Lego Mindstorm RCX.

## 6.1 Introduction

L'architecture originale de THINK, en l'état, permet théoriquement de couvrir tout le spectre de développement des systèmes d'exploitation. Le modèle

est à la fois suffisamment simple pour convenir aux systèmes les plus exigeants en terme de coût, et suffisamment complet pour permettre la construction de systèmes complexes de haut niveau. Toutefois, son adaptation à une zone précise du spectre de développement passe par la projection de son modèle de composition selon des règles de projection dépendantes de la zone en question. Cette projection prend en général la forme d'une bibliothèque de composants spécialisés.

La projection de THINK vers le domaine spécifique des architectures embarquées semble intéressante pour plusieurs raisons. En premier lieu, les qualités de minimalisation de l'architecture suggèrent la construction de systèmes optimisés (ne contenant que les composants strictement nécessaires au système) et ultra-spécialisés. La flexibilité du système permet en outre de gérer la grande hétérogénéité des plates-formes embarquées. Enfin, les fonctions de reconfiguration, lorsqu'elles sont applicables, fournissent un support naturel à l'adaptation aux conditions d'exécution changeantes de ce type d'architectures.

Nous présentons dans ce chapitre une projection de THINK vers le domaine de l'embarqué. Cette projection prend la forme d'une bibliothèque de composants nommée Embedded-THINK. Les architectures matérielles et logicielles visées par cette bibliothèque doivent être précisément définies.

### 6.1.1 Architectures matérielles visées

La gamme des architectures matérielles recouvertes par l'expression "architectures embarquées" est large. Assistants personnels et micro-contrôleurs de fours à micro-ondes sont ainsi regroupés dans la même catégorie, alors que leurs capacités (processeur, mémoire, etc.) diffèrent parfois d'un facteur 1000 (un assistant personnel d'aujourd'hui dispose de plus de puissance de calcul et de mémoire qu'un micro-ordinateur d'il y a 5 ans). S'il est tout à fait possible d'utiliser un système d'exploitation "standard" pour gérer un assistant personnel, ce ne sera pas le cas pour le micro-contrôleur du four à micro-ondes, qui ne dispose que d'une quantité très limitée de mémoire (tout au plus quelques dizaines de Ko).

La bibliothèque Embedded-THINK ne s'intéresse qu'aux matériels très fortement contraints, c'est à dire dont la capacité mémoire n'excède pas 512 Ko. Les systèmes répondant à cette contrainte se retrouvent principalement dans le domaine de l'électroménager (aspirateurs, machines à laver, etc.) ou des mini-robots industriels (électro-vannes, distributeurs, etc.). Les périphériques d'entrée d'un appareil de ce type sont généralement constitués par des capteurs (de température, de poids, etc.), des boutons (interrupteurs, touches, leviers, etc.) ou

des interfaces de communication réseau. En sortie, le système agit sur des moteurs, des interrupteurs (rupteurs, vannes), des afficheurs (diodes, écrans lcd), etc. Le comportement de ce genre de systèmes consiste essentiellement à déclencher des actions sur ses périphériques de sortie en fonction des événements déclenchés par ses périphériques d'entrée.

### 6.1.2 Applications logicielles visées

La bibliothèque Embedded-THINK vise tout particulièrement les applications logicielles "réactives", c'est à dire qui déclenchent des actions lors de certains événements. Un événement est constitué par l'occurrence conjointe de faits logiciels (une variable qui dépasse un seuil par exemple) ou matériels (un capteur enregistre une valeur particulière). Une action consiste à l'exécution séquentielle d'un certain nombre d'instructions.

## 6.2 Stratégies de conception et d'implémentation

Les architectures matérielles et logicielles ciblées par la bibliothèque Embedded-THINK imposent certaines exigences sur le système d'exploitation. Afin d'y répondre de façon optimale, la bibliothèque est bâtie selon des stratégies de conception particulières et appliquées de façon systématique.

### 6.2.1 Limiter le coût inhérent au modèle de composition

La bibliothèque Embedded-THINK s'adresse à des architectures matérielles fortement contraintes, où la mémoire et le temps processeur constituent des ressources rares et qu'il convient donc d'utiliser à bon escient. Il est ainsi primordial de limiter au possible les coûts liés au modèle de composition pour consacrer le maximum de temps aux opérations fonctionnelles.

Le modèle de composition de THINK a été pensé avec un objectif de minimalisation des coûts. Nous devons conserver le même esprit lors du développement de la bibliothèque Embedded-THINK. Nous devons en particulier prendre garde que les mécanismes que nous introduisons ne coûtent que lorsqu'ils sont utilisés.

### 6.2.2 Effectuer le maximum de travail avant exécution

Le développeur d'un système d'exploitation dispose souvent, au moment de la conception du système, d'informations lui permettant de minimiser les traitements à effectuer à l'exécution du système. Par exemple, il n'est pas utile de donner la possibilité de reconfigurer l'ensemble du système si seule une sous-partie de ce système le sera effectivement. Ainsi, certaines compositions hiérar-

chiques dont on a la certitude qu'elles ne seront jamais reconfigurées peuvent être optimisées et réduites à un composant à plat.

Tout ce qui peut être fait statiquement (soit à la compilation d'un composant, soit à l'assemblage du noyau) doit l'être. Ne subsistent à l'exécution que les opérations strictement nécessaires à l'exécution du noyau qui n'ont pu, par manque d'informations, être effectuées statiquement, à sa conception.

### 6.2.3 Proposer une bibliothèque de composants spécialisés

Afin de faciliter la tâche du développeur, il est nécessaire de lui proposer une implémentation optimisée de certaines des fonctionnalités que l'on retrouve dans la plupart des systèmes embarqués sous forme d'une librairie de composants. L'usage de ces composants doit toutefois rester optionnel et lorsque le développeur exprime un besoin particulier, il doit être libre de les redéfinir.

## 6.3 La bibliothèque Embedded-THINK

La bibliothèque Embedded-THINK est, à la manière de la bibliothèque Kortex, un ensemble de composants logiciels bâtis selon le modèle THINK mais offrant des fonctionnalités de plus haut-niveau que le modèle standard.

### 6.3.1 Objectifs

L'objectif de la bibliothèque Embedded-THINK est de répondre aux besoins des applications embarquées fortement contraintes. Celles-ci sont pour la plupart construites selon un modèle événement / réaction. Dans ce modèle, toute action logicielle ne peut être déclenchée que par l'occurrence d'un événement. D'autre par, la communication entre les modules logiciels s'effectue par émission et réception de valeur. L'émission d'une valeur ne précède pas forcément immédiatement sa réception : on parle ainsi de communication asynchrone entre modules.

Bien souvent, les caractéristiques techniques de la plate-forme matérielle ne permettent pas l'exécution d'un système multi-tâches : l'espace mémoire nécessaire à la sauvegarde des différents contextes d'exécution des tâches du système serait trop important par rapport à la quantité de mémoire totale du système. Il est alors préférable d'utiliser un système synchrone, dans lequel toute tâche est atomique (c'est à dire qu'une fois lancée, elle continue son exécution jusqu'à la fin, sans être interrompue). On économise ainsi l'espace mémoire nécessaire au stockage des contextes des tâches, et le temps processeur alloué aux changement de contexte.

La bibliothèque Embdedded-THINK se doit donc de répondre à de multiples besoins logiciels. Ceux-ci gravitent essentiellement autour de trois axes :

- Le développement d'un mécanisme de communication de type événement / réaction;
- Le développement d'une fabrique de liaisons permettant de créer des canaux de communication asynchrones entre composants.
- La mise sur pied d'un modèle d'ordonnancement synchrone des actions, c'est à dire permettant l'exécution de séquences d'actions sans utiliser de tâches multiples.

C'est donc d'un nouveau modèle d'interaction que nous avons effectivement besoin. Les composants nécessaires à l'implémentation de ce modèle sont fournis par la bibliothèque. Ils doivent bien sûr respecter le modèle de composition de THINK.

### 6.3.2 Aperçu d'un système Embedded-THINK

Le modèle d'interaction défini par la bibliothèque Embedded-THINK est une spécialisation du modèle général d'interaction de THINK. Les notions de composant, d'interface et de liaison sont redéfinis afin d'offrir les caractéristiques souhaitées[1].

Un système construit à l'aide de la bibliothèque Embedded-THINK répond à des règles précises de composition. Un système est ainsi une composition plus ou moins complexe de composants, communiquant au travers de ports de communications, et dont l'exécution est déterminée par un mécanisme d'événements. La figure 6.1 montre un exemple de système contruit à l'aide d'Embedded-THINK. Chaque entité fait l'objet d'une description détaillée dans les sections suivantes.

### 6.3.3 Un composant Embedded-THINK

Un composant Embedded-THINK réifie un comportement du système (voir figure 6.2). Un comportement est une action logicielle qui consiste à calculer des valeurs de sortie en fonction de valeurs d'entrée (le calcul d'une vitesse à partir d'une distance et d'une durée est un exemple de comportement). Les composants communiquent entre eux à travers des ports de communication. Ainsi, les valeurs d'entrées (resp. valeurs de sortie) sont lues (resp. écrites) sur des ports d'entrée (resp. de sortie). Le transfert des données entre les ports de communication des composants est assuré par une liaison asynchrone.

1. Bien sûr, en tant que spécialisations, les nouvelles définitions des concepts de composant, interface et liaison restent compatibles avec celles qui leur sont données dans le modèle général de THINK.

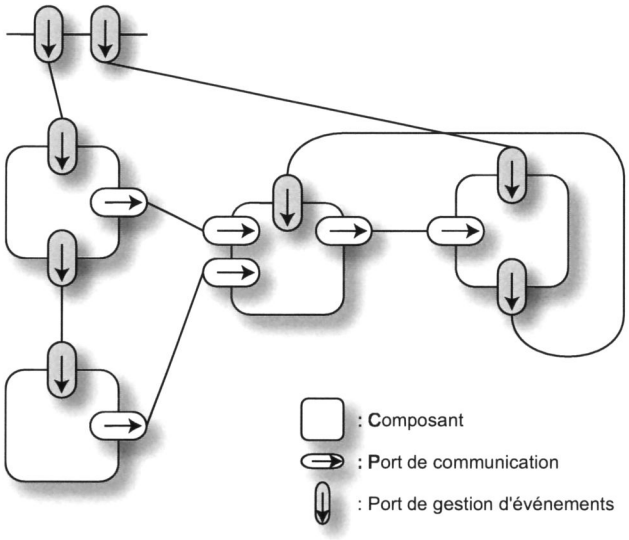


FIG. 6.1 – *Exemple d'un système Embedded-*THINK

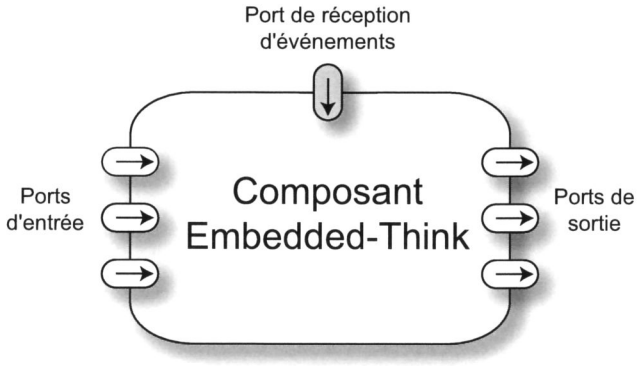


FIG. 6.2 – *Un composant Embedded-*THINK

Le comportement d'un composant est exécuté lorsqu'un événement particulier survient. Ainsi, tout composant comporte au moins un port d'entrée chargé de réceptionner des événements d'un certain type. Lorsqu'un événement est reçu sur ce port, le comportement du composant est exécuté. Les événements sont générés par des composants sur un de leurs ports de sorties et sont reliés par des canaux événementiels.

Un composant possède un état fonctionnel susceptible d'évoluer au cours du temps en fonction de son propre comportement et de celui des autres composants du système. Un composant possède également un ensemble de propriétés non-fonctionnelles, appelées méta-informations.

### 6.3.4 Un port Embedded-THINK

Les composants communiquent avec le monde extérieur grâce à des ports. À chaque port sont associés une direction et un type.

Un port d'un composant est qualifié de "port d'entrée" s'il permet à un composant de recevoir des informations de l'extérieur. Il est au contraire qualifié de "port de sortie" s'il permet à un composant d'envoyer des informations vers l'extérieur. Notons qu'il peut exister des ports de type mixte (entrée et sortie). Il existe également une autre classe de port, qui se trouve en fait être une sous-classe du port d'entrée : le port de réception d'événement. Un port de réception d'événements est utilisé par les composants pour recevoir des événements.

Le type d'un port correspond au type de données que le port est susceptible de transférer. Deux ports peuvent être reliés si et seulement si leur type est équivalent et leurs directions sont complémentaires.

### 6.3.5 Les liaisons asynchrones

Les ports sont reliés entre eux par des liaisons. Une liaison est un composant implémentant une sémantique particulière de transfert d'information. Différentes sémantiques de communication peuvent ainsi cœxister au sein d'un seul et même système.

Une liaison relie deux ports ou plus, à condition que ceux-ci soient compatibles. Ainsi, si une liaison reliant un port d'entrée à N ports de sorties est envisageable, il est par contre impossible de connecter plusieurs ports de sorties par une même liaison. Toutefois, un port peut être connecté à plusieurs liaisons différentes. Ce type de configuration permet par exemple à un composant de diversifier ses sources d'information.

La figure 6.3 présente les différents types de liaison. Une liaison simple relie deux ports, une liaison multi-points relie un port de sortie à N ports d'entrée.

Pour relier plusieurs ports de sortie à un port d'entrée, on utilise des liaisons multiples.

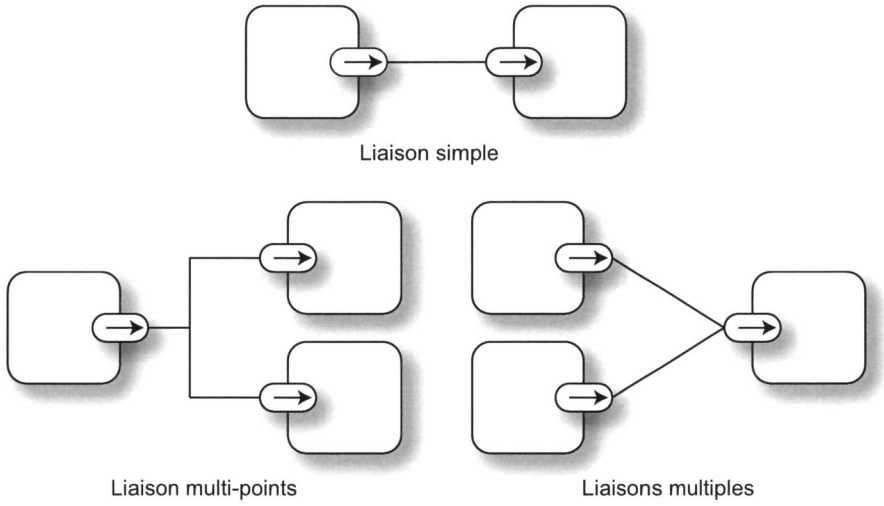


FIG. 6.3 – *Les différents types de liaison*

### 6.3.6 Les canaux d'événements

Les canaux d'événements ont une sémantique bien différente des simples liaisons asynchrones. En effet, un canal d'événement, à la différence des liaisons asynchrones, ne se contente pas de transférer une valeur d'un port à un autre. Il doit en plus démarrer l'exécution des comportements associés à la réception d'un événement sur un port. Toutefois, chacun de ces comportements s'exécute au sein du même flot d'exécution que le canal d'événement : toutes les exécutions sont donc séquentielles (le canal d'événements assure un ordonnancement causal des exécutions des événements).

Un canal d'événement relie `N` ports de sortie à `N` ports de réception d'événements. Lorsqu'un événement est produit sur le port de sortie, il est transmis à tous les ports de réception d'événements reliés au canal et les réactions associées sont exécutées.

## 6.4 Implémentation de la bibliothèque

Le modèle d'interaction présenté dans la section précédente s'implémente en utilisant les concepts originaux du modèle de composition de THINK. Pour des raisons de compréhension, nous présentons les concepts du modèle d'interaction

dans un ordre différent de celui dans lequel ils vous ont été présentés dans la section précédente.

### 6.4.1 Un port Embedded-THINK

Un composant manipule constamment ses ports de communication, les opérations sur les ports doivent donc être très efficace. Le temps d'accès au port doit être très rapide et ne peut supporter une quelconque attente (due, par exemple, à un mécanisme de synchronisation).

Un port est constitué d'une variable. L'accès au port se fait par adressage direct. L'envoi (resp. la réception) d'une donnée vers un port de sortie (resp. d'entrée) correspond ainsi simplement à l'écriture (resp. la lecture) de la donnée à l'adresse du port. Aucun mécanisme de synchronisation n'est nécessaire : l'ensemble du système fonctionne de manière synchrone, évitant ainsi les accès concurrents à un même port. Ainsi, d'un point de vue purement technique, un port de communication est un pointeur vers un espace mémoire de la taille des données transférées par le port (voir figure 6.4).

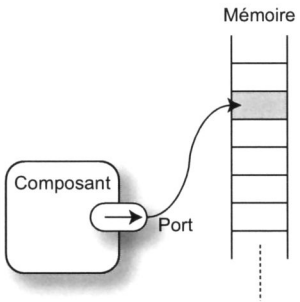


FIG. 6.4 – *Un port de communication*

Il est important de noter que pour des raisons de performance, aucun mécanisme de sécurité n'est prévu par le système : la mémoire est partagée, et rien n'empêche un composant d'écraser la valeur d'un port auquel il n'a théoriquement pas accès. Toutefois, lorsque cela est nécessaire, des mécanismes de liaison sécurisées peuvent être introduits au sein du système sous forme de nouvelles fabriques de liaisons.

Les ports d'émission et de réception d'événements ont un fonctionnement différent des ports de communication classiques. En effet, l'émission d'un événement implique certaines opérations et la communication ne peut donc s'effectuer par simple lecture / écriture sur une variable partagée. Les aspects liés au fonctionnement des ports d'émission et de réception d'événements sont étudiés dans la section 6.4.3.

### 6.4.2 Les liaisons asynchrones

Les liaisons asynchrones sont impliquées dans la plupart des interactions entre composants d'un système construit à l'aide de la bibliothèque Embedded-THINK. De fait, leur optimisation revêt un caractère primordial.

Nous avons adopté une architecture de liaison très simple. Afin de minimiser son occupation mémoire et les opérations de copie de données, elle se base sur l'utilisation de variables partagées. Une liaison asynchrone est ainsi une variable partagée entre les ports connectés par la liaison. Lorsqu'un composant envoie une valeur sur son port de sortie, c'est en réalité une écriture de la variable partagée qui est effectuée. La réciproque est également valable : lorsqu'un composant reçoit une valeur de l'un de ses ports d'entrée, c'est en réalité une lecture de la variable partagée qui est effectuée. Une liaison élémentaire peut également être multi-points : dans ce cas, tous les ports connectés par cette liaison partagent la même valeur. D'un point de vue technique, chaque port pointe vers le même espace mémoire (voir figure 6.5).

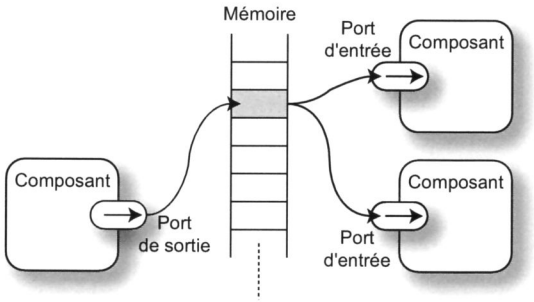


FIG. 6.5 – *Une liaison asynchrone*

Bien évidemment, le mécanisme de liaison décrit ci-dessus peut servir de base à la construction de mécanismes de communication plus évolués, représentés à l'exécution par un (ensemble de) composant(s) complexe(s). Il est par exemple possible de concevoir une liaison disposant d'un tampon de données, permettant de ne pas écraser une valeur lors de plusieurs écritures successives (elles sont alors stockées dans le tampon en attente de leur lecture).

### 6.4.3 Les canaux d'événements

Un canal d'événements est une liaison particulière. Il doit transmettre l'événement à tous les composants qui lui sont connectés et démarrer l'exécution des réactions associées. L'exécution des différentes réactions s'effectuent selon l'ordre causal de réception des différents événements.

Il existe un canal par type d'événements. Un canal est créé lors de la liaison entre un composant producteur d'un type d'événements et un composant consommateur du même type d'événements. Lorsque d'autres composants ont besoin de recevoir les événements émis par le producteur, ils se connectent au canal déjà existant. Lorsqu'un événement est produit, il est transmis à l'ensemble des consommateurs et les réactions de chacun des consommateurs sont exécutées.

Le fonctionnement des canaux d'événements implique quatre entités : le producteur de l'événement, le consommateur, le canal d'événements et le gestionnaire de canaux. Le fonctionnement de chacune de ces quatre entités et leurs relations mutuelles sont représentés sur la figure 6.6.

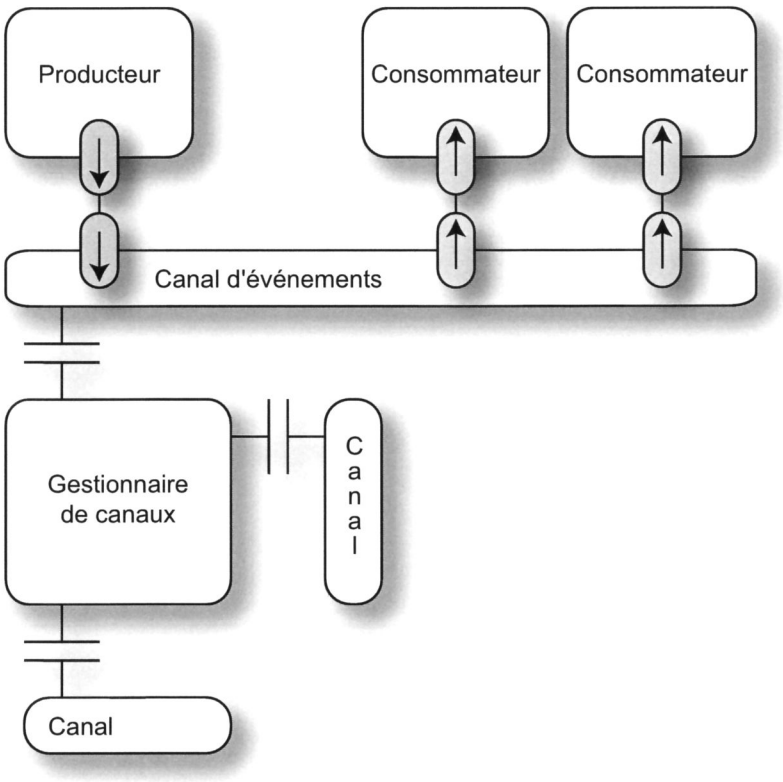


FIG. 6.6 – *Fonctionnement des canaux d'événements*

Un composant émet un événement en envoyant une valeur particulière sur un port de sortie dédié. La procédure d'envoi d'un événement consiste à enregistrer l'événement produit dans la file des événements du canal. Le canal d'événements auquel est connecté le port est alors informé qu'un événement est en attente de traitement.

Le gestionnaire de canaux est le seul processus actif du système. Il est chargé de lancer le traitement des événements de chacun des canaux qu'il gère. Il maintient ainsi une liste de canaux; pour chaque canal dans la liste, il lance le traitement du prochain événement dans la liste d'attente du canal.

La figure 6.7 présente la structure détaillée d'un canal d'événements. Un canal d'événements est un composant THINK. Il offre une interface serveur `Producer`, et autant d'interfaces client `Consumer` que de composants récepteurs d'événements connectés. En sus, une interface serveur `EventChannel` permet de lier le canal au gestionnaire de canaux.

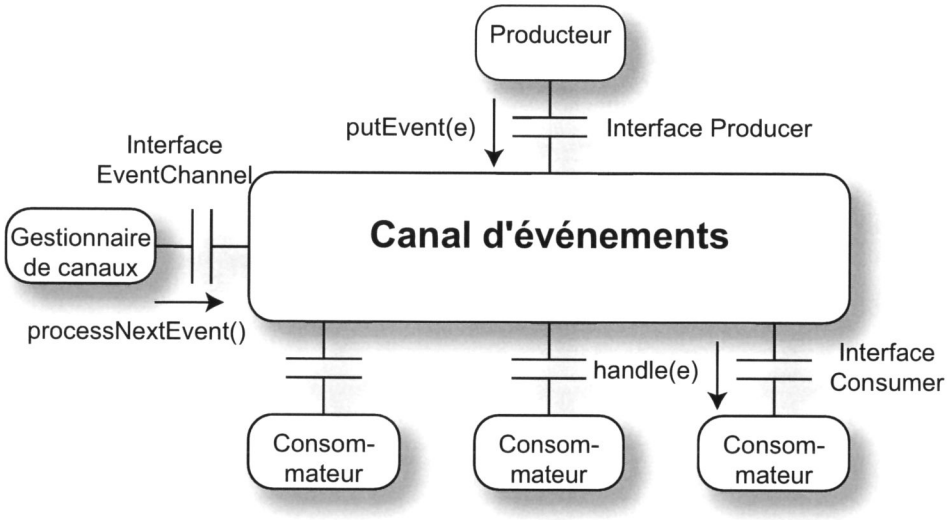


FIG. 6.7 – *Structure d'un canal d'événements*

L'interface `Producer` ne fournit qu'une seule méthode, `putEvent`. Cette méthode prend une donnée en paramètre (un événement) et la stocke dans une liste locale au canal (la liste des événements à traiter). L'interface `EventChannel` n'offre également qu'une seule méthode, `processNextEvent`. Cette méthode, appelée par le gestionnaire de canaux, transmet à tous les composants récepteurs d'événements connectés au canal l'événement en tête de liste et exécute la réaction associée. La réaction est exécutée par un appel à la méthode `handle` sur l'interface `Consumer` de chacun des composants connectés. Lorsque toutes les réactions ont été exécutées, l'événement est supprimé de la liste et le canal rend la main au gestionnaire de canaux.

Les interfaces `Producer`, `Consumer` et `EventChannel` sont présentés sur le listing 6.1.

Bien sûr, L'implémentation proposée ici est rudimentaire. Elle peut toutefois

Listing 6.1: *Les interfaces* `Producer`, `Consumer` *et* `EventChannel`

```
interface Producer {
    void putEvent(Event e);
}

interface Consumer {
    void handle(Event e);
}

interface EventChannel {
    void processNextEvent();
}
```

servir de base à l'implémentation de canaux d'événements plus complexes, implémentant des sémantiques particulières de transport d'événements. Par exemple, on peut imaginer des canaux d'événements chargés de la combinaison logique de plusieurs événements, dont la sémantique pourrait par exemple être "lorsque les événements `A` et `B` se produisent simultanément, générer l'événement `C`".

### 6.4.4 Un composant Embedded-THINK

Un composant Embedded-THINK modélise un comportement logiciel déclenché par un événement. A son initialisation, un composant se connecte au canal transportant le type d'événements souhaité. Lorsqu'un événement sera reçu par le canal, il lancera l'exécution du composant. Lors de cette exécution, le composant pourra à son tour émettre des événements vers d'autres composants du système. A la fin de son exécution, le composant rend la main au canal qui pourra exécuter les autres composants en écoute du même événement.

Un composant Embedded-THINK est avant tout un composant THINK (voir figure 6.8). Les ports d'émission et de réception d'événements sont traités comme des interfaces classiques THINK. Ainsi, un composant offre au minimum une interface fonctionnelle serveur de type `Consumer` afin de recevoir des événements. Si le composant produit des événements, il offre en sus une interface client de type `Producer`. Les ports d'entrée et de sortie d'un composant sont, eux, stockés sous forme d'un tableau de pointeurs au sein de ses variables d'instance. On y accède au travers de deux macros C, `PUT` et `GET`.

Le comportement fonctionnel du composant est définit au sein de la procédure `handle` de l'interface `Consumer`. Au cours de ce comportement, le composant peut récupérer des valeurs sur ses ports d'entrée, émettre des valeurs vers ses ports de sortie et émettre des événements. Le système est synchrone : lorsqu'un comportement est exécuté, il ne sera pas interrompu jusqu'à la fin de son exécution. En conséquence, pour préserver l'efficacité du système, La durée d'exécution d'un comportement doit rester raisonnable.

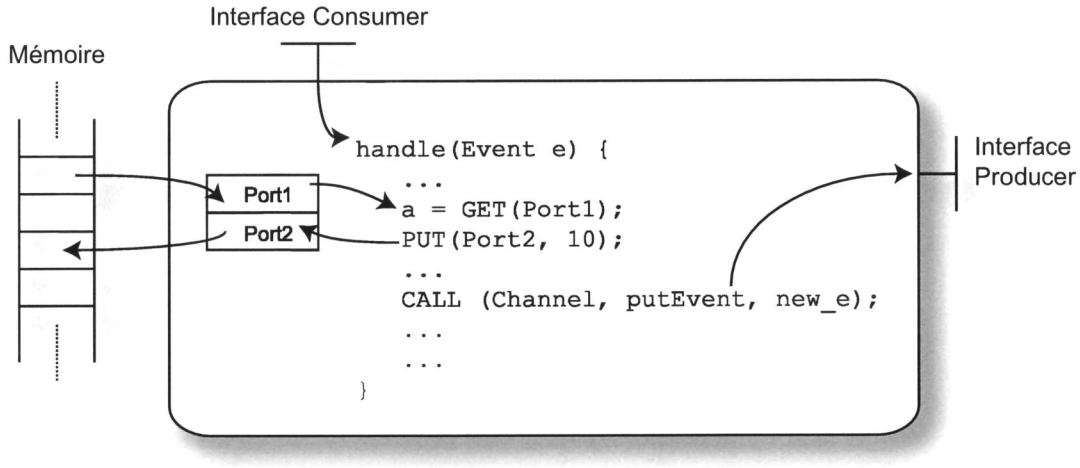


FIG. 6.8 – *Structure d'un composant ET*

### 6.4.5 Gestion des interruptions

Il a été dit plus haut que le comportement fonctionnel d'un composant n'était exécuté que lorsqu'un événement particulier lui était transmis sur son port événementiel. Cette règle est vraie pour l'ensemble des composants, sauf dans un cas : les composants réifiant les interruptions.

Une interruption, par définition, est un événement dont l'occurrence ne peut être contrôlée. Dans la bibliothèque `Embedded-THINK`, les interruptions sont gérées par des composants spéciaux, qui se chargent d'émettre un événement sur un canal dédié lors de l'occurrence d'une interruption. Ainsi, tous les composants connectés au canal recevront l'événement correspondant à l'interruption.

La figure 6.9 montre un exemple de composants traitants d'interruption. A la différence des autres composants `Embedded-THINK`, ils ne comportent pas de port de réception d'événement. Leur comportement est enregistré comme traitant d'une interruption particulière ; il sera automatiquement exécuté lors de l'occurrence de cette interruption.

Notons qu'en l'absence de toute interruption, un système Embedded-THINK doit être amorcé artificiellement par l'émission d'un événement. En effet, tout événement est produit par un composant, dont le comportement a lui-même été déclenché par un événement. Si aucun événement n'est émis par une autre entité qu'un composant Embedded-THINK, le système ne peut démarrer.

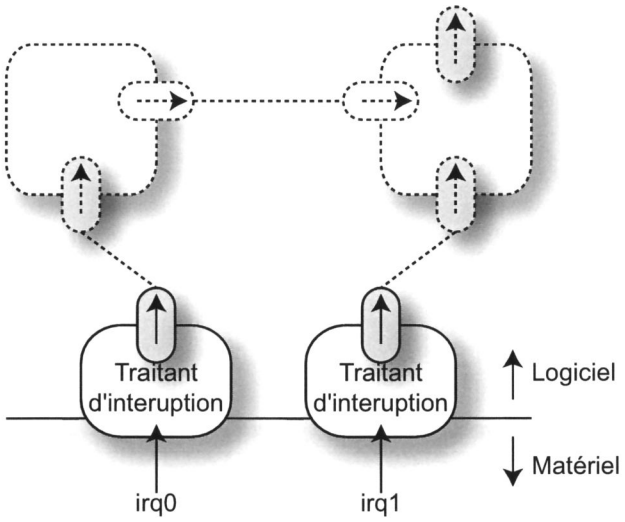


FIG. 6.9 – *Gestion des interruptions*

### 6.4.6 Mise en place d'un système Embedded-THINK

Le premier composant du système est le composant `System`. Ce composant ne définit qu'une seule méthode, `kernelStart`, appelée par le système lors de l'amorçage de la machine. Sa première action est de charger un mini-courtier et l'ensemble des fabriques de composants du système. Au moment de son instanciation, chaque fabrique s'inscrit auprès du courtier sous un nom déterminé. Il lance ensuite l'instanciation des composants du système.

Le mini-courtier maintient deux types d'informations : la liste des fabriques de composants du système et la liste des instances déjà créées pour chaque type de composant. Lorsqu'un composant le contacte pour obtenir la référence d'une instance d'un composant particulier, il regarde si une telle instance existe. Si tel est le cas, la référence de l'instance lui est retournée. Dans le cas contraire, le courtier contacte la fabrique de composants correspondante et crée une nouvelle instance du composant, l'enregistre dans sa liste et renvoie sa référence.

Une fois l'ensemble des composants du système instanciés, le mini-courtier et les fabriques de composants sont déchargés. Le composant `System` lance le gestionnaire de canaux d'événements. Le système est alors totalement opérationnel et n'attend que l'émission d'un événement (par exemple lors d'une interruption) pour s'exécuter.

Ainsi, lors de son exécution, un noyau Embedded-THINK est constitué (en termes de composants THINK) :

- de composants ET;

- de composants traitant d'interruptions;
- de composants de types canaux d'événements;
- d'un gestionnaire de canaux;
- du composant `System` englobant.

## 6.5 Un exemple de modélisation

Supposons l'existence d'un système dont le but est de mesurer la vitesse courante du véhicule au sein duquel il est embarqué, de l'afficher à intervalles réguliers sur un écran à cristaux liquides et d'avertir l'usager du véhicule lorsque la vitesse dépasse un certain seuil pendant une durée minimale de dix secondes grâce à l'émission d'un son.

Matériellement, supposons le système constitué d'un compteur de distance, d'une horloge, d'un "chien de garde", d'un écran à cristaux liquides et d'un haut parleur. Un microprocesseur et quelques kilo-octets de mémoire permettent en sus de contrôler le tout.

Le système ci-dessus peut être modélisé à l'aide de la bibliothèque Embedded-THINK. Une modélisation possible de ce système est présentée sur la figure 6.10. Cette modélisation fait apparaître huit composants différents, dont voici la description :

**`Clock` et `Watchdog`** Ces deux composants sont des traitants d'interruption. Lorsque l'interruption à laquelle ils sont rattachés survient, ils émettent un événement sur le canal auquel ils sont reliés. Ainsi, le composant `Clock` émet un événement toutes les `X` millisecondes (La valeur `X` étant fixée lors de l'initialisation du composant). Le composant `Watchdog`, lui, émet un événement lorsqu'un délai qui a été initialisé par ailleurs arrive à son terme.

**`Distance Computing`** Ce composant est relié à l'horloge par son port de réception d'événement. Ainsi, lorsqu'il reçoit un événement de l'horloge (c'est à dire toutes les `X` millisecondes), il récupère auprès de compteur de distance la distance parcourue depuis le lancement du système et l'envoie sur son port de sortie.

**`Speed Computing`** Ce composant est également relié à l'horloge par son port d'événements. Lorsqu'il reçoit un événement de l'horloge, il récupère la valeur sur son port d'entrée (c'est à dire la distance parcourue) et calcule la vitesse courante du véhicule (grâce aux informations de temps reçues de l'horloge) et génère un événement contenant la vitesse du véhicule.

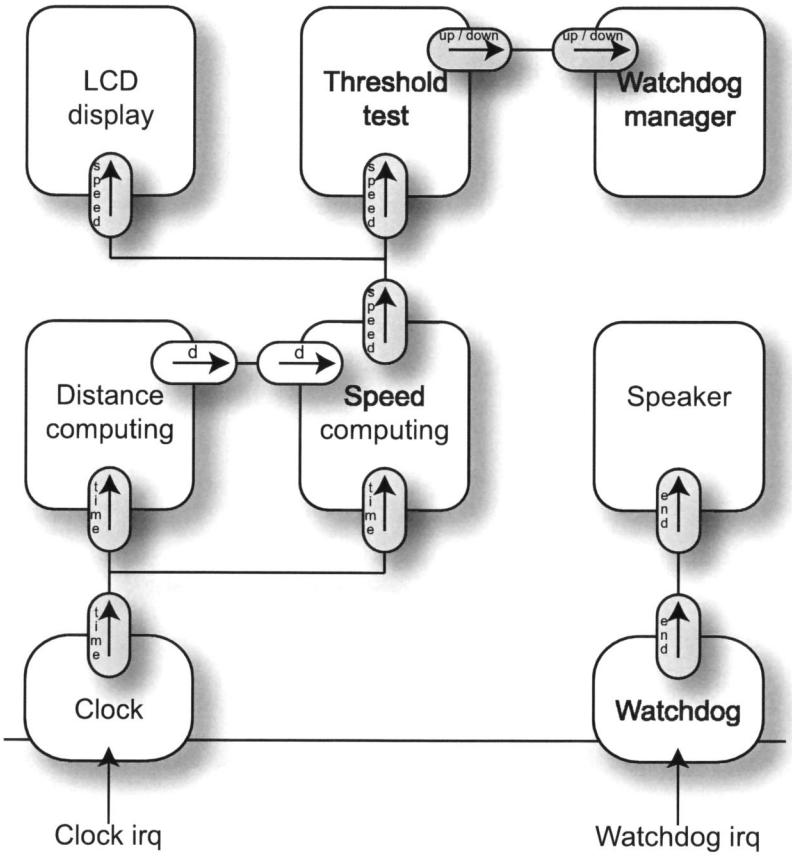


FIG. 6.10 – *Exemple de modélisation : un compteur de vitesse*

`LCD Display` Ce composant est relié au calculateur de vitesse par son port de réception d'événements. Lorsqu'il reçoit un événement, il se sert des données contenues dans l'événement pour afficher la vitesse courante du véhicule sur l'écran à cristaux liquides.

`Threshold Test` Ce composant est également relié au calculateur de vitesse par son port de réception d'événements. Lorsqu'il reçoit un événement, il teste la vitesse pour savoir si elle a franchi un certain seuil (il conserve pour cela l'ancienne valeur de la vitesse dans ses données d'instance). Si tel est le cas, il génère un événement contenant le sens de franchissement du seuil (le sens de franchissement indique si la vitesse vient de passer au dessous, ou au contraire au dessus du seuil).

`Watchdog Manager` Ce composant est relié au testeur de vitesse par son composant de réception d'événement. Lorsqu'il reçoit un événement, il met en place un chien de garde si l'événement indique que la vitesse vient de passer au dessus du seuil, ou il annule le chien de garde courant si l'événement indique que la vitesse vient de tomber au dessous du seuil et si un chien de garde était en place.

`Speaker` Ce composant est relié au composant `Watchdog` par son port de réception d'événement. Lorsqu'il reçoit un événement (c'est à dire lorsque le délai du chien de garde est arrivé à expiration), il ordonne au haut-parleur du système d'émettre un son.

Notons que, pour des raisons de clarté, la figure 6.10 ne fait pas apparaître de façon explicite les canaux d'événements et le gestionnaire de canaux (les canaux d'événements ne sont représentés que par un simple trait). Ceux-ci sont toutefois présents et assurent la transmission des événements du système selon les mécanismes décrits en section 6.4.3.

## 6.6 Outils de composition

Cette section présente le cycle de développement d'un système construit à l'aide d'Embedded-THINK, et les différents outils de composition mis à la disposition du développeur par la bibliothèque.

### 6.6.1 Cycle de développement

La figure 6.11 illustre le cycle de développement classique d'un système conçu à l'aide d'Embedded- THINK. Ce cycle comporte les étapes suivantes :

1. La **modélisation conceptuelle** du système est la première des tâches à accomplir. À cette étape, on identifie les différents composants constituant,

une fois assemblés, le système final. Cette étape est primordiale, car elle conditionne la qualité de l'application finale.

2. La **description logicielle** du système constitue la deuxième étape du processus de développement. À cette étape, le développeur décrit dans un langage de composition spécifique les différents composants du système final.
3. La **composition du système** et sa **vérification**, bien que parfois couplées à l'étape précédente, sont ensuite nécessaires. À cette étape, le développeur assemble les différents composants de son application. Des outils lui permettent de vérifier la cohérence du système final à l'aide de chacune des descriptions des composants issues de l'étape précédente.
4. À partir d'une composition valide, la **génération d'un squelette** du système final, correspondant à la partie du code non fonctionnel directement déductible des spécifications des composants, devient possible et est effectuée.
5. Le développeur doit ensuite compléter le code généré par l'étape précédente par l'**écriture du code fonctionnel et de la partie du code non-fonctionnel non déductible** des spécifications des composants.
6. La **compilation des sources** obtenues par les deux étapes précédentes et leur éventuelle **liaison avec des composants binaires pré-existants** constitue l'avant dernière étape du processus de développement.
7. Enfin, le **chargement** et l'**exécution** du système final sur la machine cible termine le processus de développement du système.

Les étapes 1 à 3 sont indépendantes des caractéristiques de la machine cible. A partir de l'étape 4 par contre, le processus devient directement dépendant de la machine : le code produit doit être conforme aux spécifications de l'architecture matérielle cible. La librairie Embedded-THINK assiste le développeur lors des étapes 2, 3 et 4. L' étape 1 ne nécessite pas réellement d'outil spécifique, et les étapes 6 et 7 sont trop dépendantes de la machine cible pour être généralisées.

La description logicielle d'un système est facilitée par un langage de description de composants. La composition d'un système et sa vérification est effectuée grâce à un outil graphique de composition. Enfin, des générateurs de code permettent la génération des squelettes de composants à destination d'une architecture matérielle donnée.

### 6.6.2 Le langage de description de composants ET

Le langage de composition proposé par la bibliothèque Embedded-THINK permet la description complète, d'un point de vue structurel, d'un composant

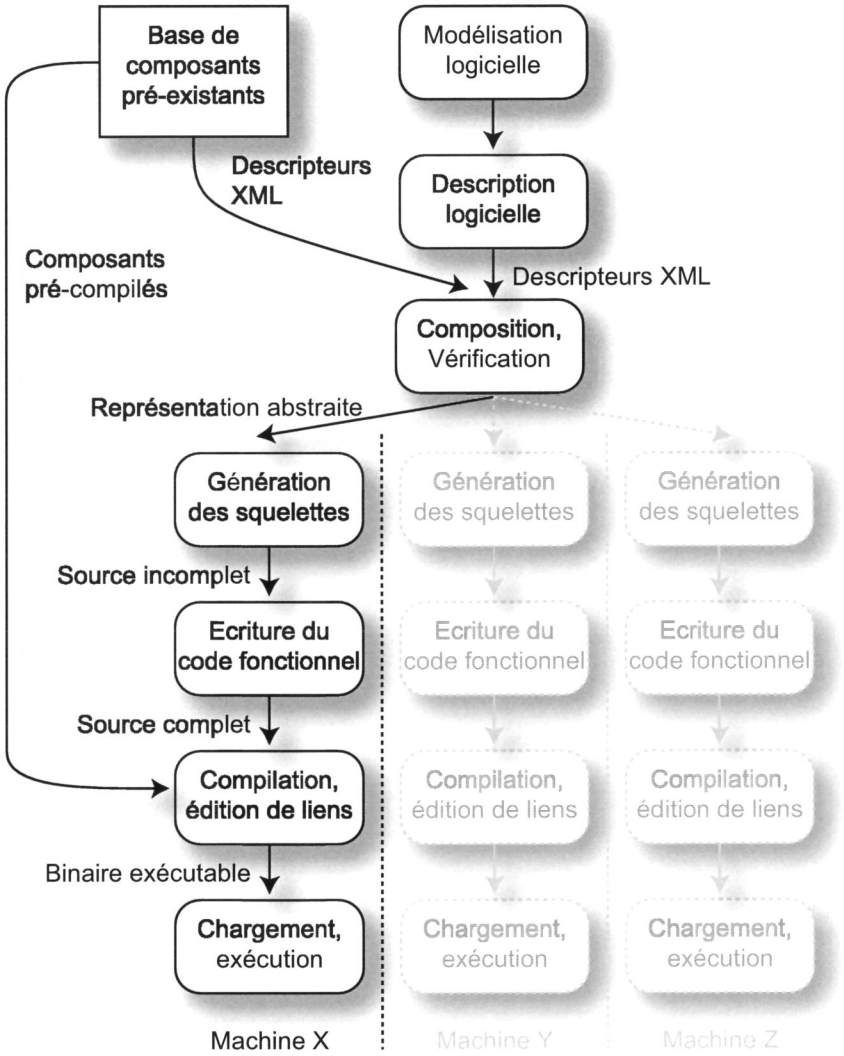


FIG. 6.11 – *Cycle de développement d'un système Embedded-*THINK

ET. Basé sur XML, il se présente sous la forme d'une simple DTD (Document Type Definition). Cette section explique à travers un exemple simple les principaux concepts du langage; la DTD complète, elle, est fournie en annexe.

Reprenons l'exemple de système de la figure 6.10, et plus spécifiquement le composant chargé de calculer la vitesse du véhicule, `Speed computing`. Le composant et sa représentation dans le langage de description de composants ET sont représentés sur la figure 6.12.

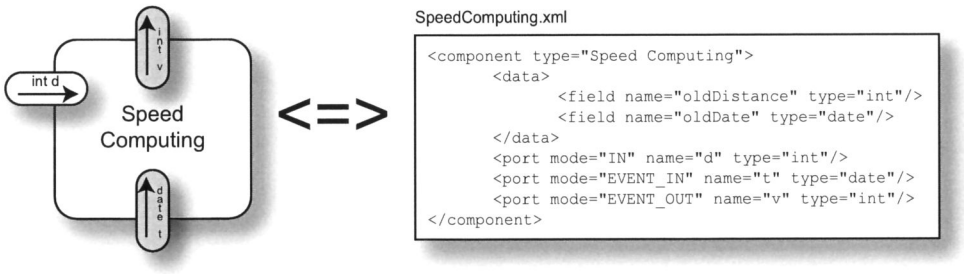


FIG. 6.12 – *Exemple d'une description de composant*

Toute description de composant est encapsulée au sein de balises `<component>`. Une option de cette balise, `type`, permet de spécifier le type du composant en cours de description. Une description correspond bien à un type de composant, et non à une instance de composant. Autrement dit, il existe une seule description pour toutes les instances d'un même type de composant. Sur l'exemple, le composant décrit est de type `SpeedComputing`.

Les balises `<data>` permettent ensuite de spécifier les données d'instance du composant. La description de ces données servira à la génération de la capsule du composant. Ainsi, tout ce qui caractérise l'état d'instance d'un composant doit être spécifié ici. Cela s'effectue grâce aux balises `<field>` qui permettent, grâce à leur options `name` et `type`, de spécifier chacun des champs des données d'instance du composant. Sur l'exemple, on voit ainsi qu'un composant `SpeedComputing` comporte deux variables : la variable `oldDistance` de type `int` et la variable `oldDate` de type `date`.

Les balises `<port>` sont utilisées pour spécifier les ports d'un composant. L'option `mode` détermine la nature du port; quatre modes sont possibles : le mode `IN` (qui désigne un port d'entrée), `OUT` (qui désigne un port de sortie), `EVENT_IN` (qui désigne un port de réception d'événement) et `EVENT_OUT` (qui désigne un port d'émission d'événement. L'option `name` permet de spécifier le nom d'un port. En effet, un composant peut comporter plusieurs ports de même type et de même mode; leur nom permet de les identifier. Enfin, l'option `type`

permet de spécifier le type du port, c'est à dire le type des données transférées par le port. Le composant SpeedComputing comporte trois ports. Le premier, de mode `IN`, de type `int` et de nom `d` permet de recevoir une valeur correspondant à la distance parcourue par le véhicule. Le second, de mode `EVENT_IN`, de type `date` et de nom `t`, permet de recevoir le temps écoulé depuis le lancement du système. Enfin, le dernier port, de mode `EVENT_OUT`, de type `int` et de nom `v`, permet d'émettre un événement contenant la vitesse courante du véhicule.

La description complète du composant est enregistrée dans un fichier XML. L'étape suivante consiste à assembler certains des composants décrits par ces fichiers afin d'obtenir une description complète et cohérente d'un système.

### 6.6.3 L'outil graphique de composition, ETCompose

L'étape de composition et de vérification du système permet, à partir de descriptions de composants, de construire un système complet. Cette étape est facilitée par un outil de composition graphique nommé ETCompose.

ETCompose permet, à partir d'un ensemble de composants décrits en XML, de composer un système cohérent d'un point de vue typage. L'utilisateur charge des descriptions de composants et les assemble à l'aide de la souris. L'assemblage d'un système est garanti cohérent par construction : ETCompose ne permet la liaison de ports que s'ils sont compatibles.

Une fois la construction du système achevée, le développeur peut demander à ETCompose de générer les scripts de compilation du système. Cela a également pour effet de générer le composant `System`, responsable du chargement de tous les composants et de leur initialisation (voir section 6.4.6).

### 6.6.4 Un générateur de squelette de composant

Le code d'un composant contient une partie non fonctionnelle tout à fait déductible de sa spécification XML. Pour éviter au développeur le travail rébarbatif d'écriture de ce code, la bibliothèque Embedded-`Think` fournit des générateurs de squelette de composant.

Un générateur déduit de la spécification XML le code non fonctionnel d'un composant et le génère dans un fichier. En sus, il génère la fabrique qui permettra la création des instances de ce composant. Le résultat est un code "à trous", que le développeur complétera avec la partie fonctionnelle du code du composant.

Un générateur est dépendant de l'architecture matérielle de la machine cible. Embedded-`Think` ne fournit que deux générateurs. L'un est dédié aux architectures basées sur des processeurs PowerPC, l'autre vise les architectures bâties autour du micro-contrôleur Hitachi H8/3292.

## 6.7 Bilan

La bibliothèque Embedded-`Think` offre un ensemble d'outils et de concepts permettant la construction de systèmes à destination d'architectures embarquées fortement contraintes. Ce chapitre a présenté chacun des concepts de la bibliothèque et leur implémentation en termes de concepts `Think`. Il a également présenté les outils d'aide au développement fournis par la bibliothèque.

Le chapitre suivant est consacré à l'évaluation de la bibliothèque Embedded-`Think` au travers de la construction d'un système d'exploitation à destination d'une plate-forme matérielle fortement contrainte, constituée par le Lego Mindstorm RCX.

# Chapitre 7

# Évaluation : développement d'un système pour le Lego RCX

Ce chapitre présente une évaluation de l'architecture de systèmes THINK, de son extension reconfigurable et de la bibliothèque Embedded-THINK. On y présente notamment l'utilisation d'Embedded-THINK dans le cadre du développement d'un noyau pour un matériel embarqué fortement contraint : le Lego RCX. Ce noyau sert ensuite de base à plusieurs séries de tests de performances de l'architecture.

## 7.1 Introduction

Le Lego RCX (Robotic Command System) est une brique Lego programmable permettant la construction de robots autonomes. En y ajoutant des capteurs (de choc, de lumière, etc.), les robots deviennent capables d'interagir avec leur environnement. De plus, grâce à un émetteur / récepteur infrarouge, les robots ainsi construits peuvent dialoguer entre eux ou avec un ordinateur.

Si le RCX s'avère être un matériel relativement complet et performant, le noyau Lego fourni en standard n'utilise pas toutes ses capacités. De plus, il contient un interprète de bytecode qui ne laisse que très peu d'espace mémoire à l'utilisateur pour développer ses propres programmes de gestion du robot. Le développement d'un noyau alternatif, à la fois moins lourd, plus complet et plus flexible, permettrait la programmation de robots beaucoup plus complexes et évolués.

Le RCX fournit un cadre expérimentatoire idéal. C'est un environnement simple (les périphériques sont peu nombreux) et très contraint matériellement (les ressources processeur et mémoire sont limitées), mais c'est un environnement suffisamment complexe pour autoriser la construction de systèmes non-triviaux.

## 7.2 Travaux similaires

De nombreux projets se sont consacrés au développement de noyaux pour le Lego RCX. Les deux plus importants sont sans doute LegOS [Nie00] et LejOS [LFS02].

LegOS, développé par Markus Noga, a été le premier noyau alternatif à destination du RCX. LegOS est très complet : il est multi-tâches, dispose d'un allocateur dynamique de mémoire, permet le chargement dynamique de programmes grâce à l'interface infrarouge et gère l'ensemble des périphériques disponibles pour le RCX. Son développement a permis de comprendre beaucoup de ses mécanismes internes et a ouvert la voie à de nombreux autres travaux. Toutefois, même si LegOS offre un mécanisme de configuration statique, c'est un noyau monolithique, difficilement modifiable et peu évolutif.

LejOS est un projet intéressant à plus d'un titre. Tout d'abord, LejOS reprend une partie du code de LegOS en essayant de le modulariser. Enfin, et c'est là le principal apport du projet, LejOS intègre une micro-machine virtuelle Java et permet le chargement dynamique de classes Java à travers l'interface infrarouge du RCX. Il devient donc possible de programmer ses robots directement en Java. Toutefois, le noyau résultant est relativement lourd, et aucun mécanisme de configuration n'est fourni en standard.

Ces deux tentatives s'appuient sur libRCX[Lib], une librairie développée par Kekoa Proudfoot en 1999. Cette librairie offre une abstraction de l'environnement du RCX permettant le développement de noyaux rudimentaires.

## 7.3 La brique Lego RCX

La brique Lego RCX est une des 717 pièces du *Lego Mindstorms Robotics Invention Kit*, un kit permettant la construction et la programmation de robots autonomes.

### 7.3.1 Au niveau matériel

Le RCX (voir figure 7.1) contient un processeur Hitachi H8/3292 cadencé à 16MHz, une ROM de 16Ko et une RAM de 32Ko. Le processeur H8/3292 supporte un espace d'adressage de 16 bits. Il dispose de 16 registres de 8 bits chacun, pouvant être utilisés comme 8 registres de 16 bits (R0...R7) à des fins d'adressage. Deux registres de contrôle sont également présents : un registre PC (Program Counter) de 16 bits, et un registre CCR (Condition Code Register) de 8 bits.

FIG. 7.1 – *La brique Lego RCX*

Sur la brique est présent un petit écran LCD sur lequel peuvent être affichés quelques caractères et diverses icônes (une batterie, un petit homme, etc..). Le RCX dispose en outre de trois connecteurs en entrée et trois connecteurs en sortie. Sur les connecteurs en entrée, divers capteurs peuvent être branchés : des capteurs de choc, de température, de rotation d'un axe, d'intensité lumieuse. Sur les connecteurs en sortie, l'utilisateur peut soit brancher des moteurs, soit des lampes. Un mini-clavier comprenant quatre boutons est disponible. Enfin, le RCX peut communiquer avec son environnement à l'aide d'un émetteur / récepteur infrarouge.

#### 7.3.1.1 Interruptions

Le micro-contrôleur H8/3292 du Lego RCX dispose de dix-neuf sources d'interruption internes et de quatre sources externes. Les interruptions internes sont généralement inhérentes aux composants du micro-contrôleur lui-même (par exemple, le convertisseur analogique / digital déclenche une interruption à la fin d'une conversion), alors que les interruptions externes sont générées par des événements extérieurs au micro-contrôleur. Les interruptions sont listées dans le tableau 7.1.

| Source | Numéro | Description |
|---|---|---|
| NMI | 3 | Nonmaskable Interrupt (extern) |
| IRQ0 | 4 | (extern) |
| IRQ1 | 5 | (extern) |
| IRQ2 | 6 | (extern) |
| Réservé | 7 | |
| Réservé | 8 | |
| Réservé | 9 | |
| Réservé | 10 | |
| Réservé | 11 | |
| ICIA | 12 | 16-bit timer, Input Capture A |
| ICIB | 13 | 16-bit timer, Input Capture B |
| ICIC | 14 | 16-bit timer, Input Capture C |
| ICID | 15 | 16-bit timer, Input Capture D |
| OCIA | 16 | 16-bit timer, Output Compare A |
| OCIB | 17 | 16-bit timer, Output Compare B |
| FOVI | 18 | 16-bit timer, Overflow |
| CMI0A | 19 | 8-bit timer 0, Compare Match A |
| CMI0B | 20 | 8-bit timer 0, Compare Match B |
| OVI0 | 21 | 8-bit timer 0, Overflow |
| CMI1A | 22 | 8-bit timer 1, Compare Match A |
| CMI1B | 23 | 8-bit timer 1, Compare Match B |
| OVI1 | 24 | 8-bit timer 1, Overflow |
| Réservé | 25 | |
| Réservé | 26 | |
| ERI | 27 | Serial Communication Interface, Receive Error |
| RXI | 28 | Serial Communication Interface, Receive End |
| TXI | 29 | Serial Communication Interface, TDR Empty |
| TEI | 30 | Serial Communication Interface, TEI Empty |
| Réservé | 31 | |
| Réservé | 32 | |
| Réservé | 33 | |
| Réservé | 34 | |
| ADI | 35 | A/D Converter, Conversion End |
| WOVF | 36 | Watchdog Timer, WDT Overflow |

TAB. 7.1 – *Les interruptions du H8/3292*

#### 7.3.1.2 Mémoire

L'espace d'adressage du RCX est de 16 bits. Il référence les différents types de mémoire de la brique. Le tableau 7.2 présente son organisation :

| Plage mémoire | Type de mémoire | Contenu |
|---|---|---|
| 0x0000 - 0x3fff | ROM embarquée | Vecteur d'interruptions du H8/3292, exécutif du RCX |
| 0x8000 - 0xefff | RAM externe | Programmes et données |
| 0xf000 | Registre externe | Registre de périphérique pour les ports de sortie du RCX |
| 0xfd80 - 0xff7f | RAM embarquée | Vecteur d'interruptions du RCX, programmes et données |
| 0xff88 - 0xffff | Registres internes | Registres de périphériques du h8/3292 |

TAB. 7.2 – *Organisation de l'espace d'adressage du RCX*

#### 7.3.1.3 Entrées / sorties

Les programmes s'exécutant sur le RCX communiquent avec les périphériques du RCX à travers des registres de périphériques et grâce aux interruptions. La plupart des contrôleurs de périphériques du RCX sont basés sur les composants d'entrée / sortie du H8/3292 : les registres de périphérique du H8/3292 sont également utilisés comme les registres de périphérique du RCX. La seule exception concerne les ports de sortie du RCX, qui disposent de leur propre registre de périphérique.

Le tableau 7.3 référence les différents périphériques du RCX, ainsi que les registres et les interruptions qu'ils utilisent.

### 7.3.2 Au niveau logiciel

Le RCX est pourvu en standard de deux logiciels. L'un est stocké en ROM, l'autre est chargé en RAM.

Le logiciel chargé en ROM fournit des primitives de bas niveau permettant le contrôle du RCX. Les primitives permettent notamment la mise sous tension du RCX, la lecture des valeurs renvoyées par les capteurs et l'envoi de signaux de démarrage et d'arrêt aux moteurs. En outre, la ROM permet le chargement et le déchargement d'un noyau en RAM.

Par défaut, la RAM contient un noyau contenant un interprète de bytecode et gère les différents périphériques du RCX. Il permet le chargement de cinq programmes utilisateurs (au plus) décrits dans un langage très simple, très

| Périphérique | Registres | Interruptions |
|---|---|---|
| Boutons | Registres des ports d'entrée / sorties 4 et 7, Registres des interruptions IRQ0 et IRQ1 | Le bouton "Run" est connecté à l'interruption IRQ0, le bouton "On/Off" à l'interruption IRQ1 |
| Ports d'entrée | Registres du convertisseur A/D, registre du port d'entrée / sortie 6 | Interruption ADI |
| Niveau de charge de la batterie | Registres du convertisseur A/D | Interruption ADI |
| Émetteur / récepteur infrarouge | Registres de l'interface de communication série, du port d'entrée / sortie 4, de l'horloge 8 bits 1 | Interruptions SCI et de l'horloge 8 bits 1 |
| Écran LCD | Registres du port d'entrée / sortie 6 | Pas d'interruption |
| Haut-parleur | Registre du port d'entrée / sortie 6 | Interruption de l'horloge 8 bits 0 |
| Ports de sortie | Registre dans l'espace mémoire externe | Pas d'interruption |

TAB. 7.3 – *Périphériques du RCX*

contraint et très robuste (car, a priori, destiné à un public jeune et non spécialiste de la programmation). Les capacités du RCX ne sont pas exploitées au maximum par ce langage, et aucun moyen ne permet de charger des programmes développés dans un autre langage. De plus, le noyau est lourd : sur les 32Ko de RAM disponibles, seuls 6Ko restent disponibles pour les utilisateurs. Enfin, le noyau n'est pas modifiable, que ce soit statiquement ou dynamiquement.

Le noyau d'origine peut toutefois être remplacé par un nouveau noyau. En effet, lors du démarrage du RCX, la ROM offre un certain nombre de primitives permettant d'effacer le noyau courant et de télécharger par liaison infrarouge un nouveau noyau. Une fois le téléchargement d'un noyau effectué, la ROM le démarre. Enfin, le noyau étant stocké en RAM, si l'alimentation du RCX est coupée (si les piles sont retirées du boîtier), le noyau est effacé et au prochain démarrage, la ROM se mettra en attente du téléchargement d'un nouveau noyau. Notons que cette dernière caractéristique facilite grandement la mise au point de noyaux : en cas d'erreur conduisant à un blocage du système, le simple fait d'enlever les piles nous permet de restaurer le RCX.

## 7.4 Un noyau Embedded-THINK pour le RCX

La bibliothèque Embedded-THINK nous permet de construire très aisément un noyau de système pour le Lego RCX. De part la nature très réactive du système (un comportement robotique n'est autre que l'application de certaines réactions en fonction d'événements provenant de capteurs), Embedded-THINK paraît en effet être particulièrement adaptée à sa modélisation.

### 7.4.1 Modélisation du noyau

La modélisation du noyau pour le Lego RCX en termes de composants Embedded-THINK comporte deux niveaux : le niveau système, c'est à dire le niveau où évoluent les composants chargés de la gestion du matériel, et le niveau applicatif, qui désigne lui l'ensemble des composants chargés de l'exécution d'un comportement particulier. Cette distinction est uniquement formelle, elle n'a aucune existence réelle à l'exécution du système.

La plupart des périphériques d'entrée / sortie du noyau sont contrôlées par des interruptions. La couche basse du noyau sera donc constituée par les composants associés aux différentes interruptions du RCX. Chacun de ces composants est enregistré comme un traitant pour l'interruption à laquelle il est associée et émet un événement sur le canal auquel il est relié lorsque celle-ci survient.

La couche directement supérieure est constituée par les différents pilotes de périphériques du RCX. Ceux-ci offrent une vue abstraite du matériel auquel ils sont associés. Certains sont abonnés aux événements générés par les interruptions. Selon les cas, ils génèrent à leur tour des événements et/ou ils écrivent des données sur leurs ports de sortie. D'autres pilotes ne sont pas reliés à des composants traitant d'interruptions mais à des événements générés par des composants applicatifs. Ces derniers permettent aux composants applicatifs d'envoyer des ordres vers le matériel. Ils ont donc une vision totale du matériel qu'ils gèrent (il ont la capacité, par exemple, de manipuler directement certains registres de périphérique).

Ces deux types de composants (traitants d'interruption et pilotes de périphériques) constituent le noyau de base au-dessus duquel s'exécutent les différentes applications. Une application est également formée de composants. Chargés au-dessus au dessus des composants du système, ils déterminent le comportement effectif du Lego RCX. Ils agissent au travers des pilotes sur les composants du robot (capteurs, moteurs, écran LCD, etc.). Ils interagissent également entre eux afin de composer des comportements complexes.

### 7.4.2 Exemples de composants

Cette section illustre les différents types de composants du noyau. On y trouve notamment la description d'une horloge, d'un convertisseur A/D, d'un capteur de lumière et d'un détecteur de couleur. Tous ces composants seront utilisés dans l'implémentation du robot présenté dans la section suivante et peuvent donc être vus sur la figure 7.2.

#### 7.4.2.1 Un traitant d'interruption : le convertisseur A/D

Le micro-contrôleur H8/3292 du Lego RCX embarque un convertisseur analogique / digital. Ce convertisseur est utilisé par tous les pilotes de capteur afin d'obtenir, à partir d'une mesure analogique, la valeur digitale correspondante.

Le convertisseur est géré grâce à deux registres de contrôle. Il écrit le résultat de ses conversions dans quatre registres de données. Il signale la fin d'une conversion par l'émission d'une interruption (interruption ADI), à laquelle est associé un composant.

La récupération du résultat de la conversion consiste en une lecture des registres de données. Le convertisseur n'utilise que les dix bits de poids fort pour inscrire le résultat de sa conversion, ce qui explique le décalage opéré sur la valeur obtenue.

Une fois la valeur récupérée, celle-ci est stockée au sein d'un événement qui est émis sur le canal auquel est relié le composant. L'événement sera ainsi traité par tous les composants connectés au canal.

Après l'émission de l'événement, une nouvelle conversion est initialisée : les registres de contrôle du convertisseur sont mis en place de telle façon qu'ils démarrent une conversion selon des paramètres donnés et émettent une interruption lorsqu'elle sera achevée.

#### 7.4.2.2 Un pilote de périphérique : le capteur de lumière

Le composant pilote du capteur de lumière est abonné aux événements issus du convertisseur A/D. Ainsi, lorsqu'une conversion est achevée, il reçoit le résultat de la conversion. Ce résultat est brut et ne signifie rien sans information supplémentaire sur le contexte dans lequel a été effectuée la mesure. Le pilote du capteur de lumière est chargé de l'interprétation de la valeur reçue par le convertisseur dans un contexte où la mesure a été effectuée par un capteur de lumière.

Le composant pilote du capteur de lumière comporte un port de réception d'événement connecté au traitant d'interruption du convertisseur A/D et un port de sortie.

Le pilote convertit d'abord la valeur brute issue du convertisseur A/D en une valeur entre 0 et 100 correspondant à un indice de luminosité (une valeur proche de 0 indiquant un environnement très sombre, et une valeur proche de 100 indiquant un environnement très lumineux). Le résultat est ensuite envoyé vers le port de sortie du composant, rendant l'information disponible à tous les composants connectés à ce port.

#### 7.4.2.3 Un composant applicatif : le détecteur de couleur

Imaginons maintenant un composant applicatif qui, périodiquement et à l'aide de la valeur de la luminosité enregistrée par le capteur de lumière, détermine la couleur probable de la lumière enregistrée par le capteur (cela suppose que le robot évolue dans un environnement au nombre de couleurs restreintes et à l'éclairage uniforme). Si la couleur est différente de celle enregistrée précédemment, le composant applicatif émet un événement contenant la nouvelle couleur sur un port particulier.

Le composant détecteur de couleur comporte un port de réception d'événements relié à l'horloge du système, un port d'entrée relié au pilote du capteur de lumière et un port d'émission d'événements.

Le détecteur de couleur est périodiquement déclenché par un événement provenant de l'horloge. A réception de l'événement, il récupère la valeur de la couleur sur son port d'entrée et regarde à quelle couleur elle correspond. Ensuite, il vérifie si la couleur a changé en la comparant avec la couleur précédemment détectée. Si c'est le cas, il émet un événement contenant la nouvelle couleur et l'enregistre.

## 7.5 Un exemple de robot : le distributeur automatique

A l'aide du noyau Embedded-THINK développé pour le Lego RCX, nous avons construit un mini-robot, dont le rôle est de distribuer des friandises.

### 7.5.1 Description fonctionnelle

Le distributeur automatique de friandises est un robot permettant de délivrer à un utilisateur un nombre de friandises donné. L'utilisateur indique le nombre souhaité de friandises à l'aide d'une carte, qu'il insère dans le robot. Une fois insérée, la carte est lue par le robot puis éjectée. Le nombre de friandises indiqué sur la carte est ensuite distribué (il correspond au nombre de plaques 1x1 présentes sur la carte).

### 7.5.2 Description matérielle

Le robot est composé de deux parties : le lecteur de cartes et le distributeur de friandises.

Le lecteur de cartes se compose de deux capteurs (un capteur de chocs et un capteur de lumière) et d'un moteur. Lorsqu'une carte est insérée dans le lecteur, elle vient heurter le capteur de chocs. Cela provoque le démarrage du moteur et de la molette qui y est reliée, ce qui a pour effet d'entraîner la carte vers l'intérieur de l'appareil. Lors de l'avancée de la carte, chaque plaque 1x1 sur la carte provoque l'enfoncement du capteur de chocs. Un capteur de lumière, placé au fond du lecteur, permet de déclencher l'éjection de la carte. A la suite de cette opération, le nombre de friandises à distribuer est connu et transmis au distributeur.

Le distributeur de friandises se compose d'un capteur de chocs et d'un moteur. Il comporte un réservoir de friandises, fermé par un bras mobile relié au moteur. Lorsque le bras tourne, il libère une friandise et vient heurter le capteur de chocs. Le robot ordonne au bras d'effectuer un nombre de tours correspondant au nombre de friandises à distribuer. Les tours sont détectés par le capteur de chocs. Lorsque le nombre de tours a été effectué, le robot se met en attente de l'insertion d'une carte.

### 7.5.3 Description logicielle

Un robot se programme à l'aide de plusieurs composants applicatifs que l'on vient greffer au-dessus du noyau Embedded-THINK. Chaque composant implémente un comportement primitif relativement simple, et leur composition forme le comportement complexe final du robot.

Le comportement du robot distributeur de friandises est implémenté par la composition présentée sur la figure 7.2. Le noyau total ne comporte que neuf composants, dont six composants système et trois composants applicatifs.

#### 7.5.3.1 Partie système

Le robot est composé de capteurs de chocs, de capteurs de lumière et de moteurs. La partie système du robot ne comporte donc que les composants nécessaires à la gestion de ces trois types de périphérique.

L'horloge et le convertisseur A/D sont des composants traitant d'interruptions, qui réifient les interruptions par l'émission d'événements.

Les moteurs sont contrôlés par deux bits d'un registre spécial, lus périodiquement par le RCX afin de déterminer la commande à envoyer aux moteurs.

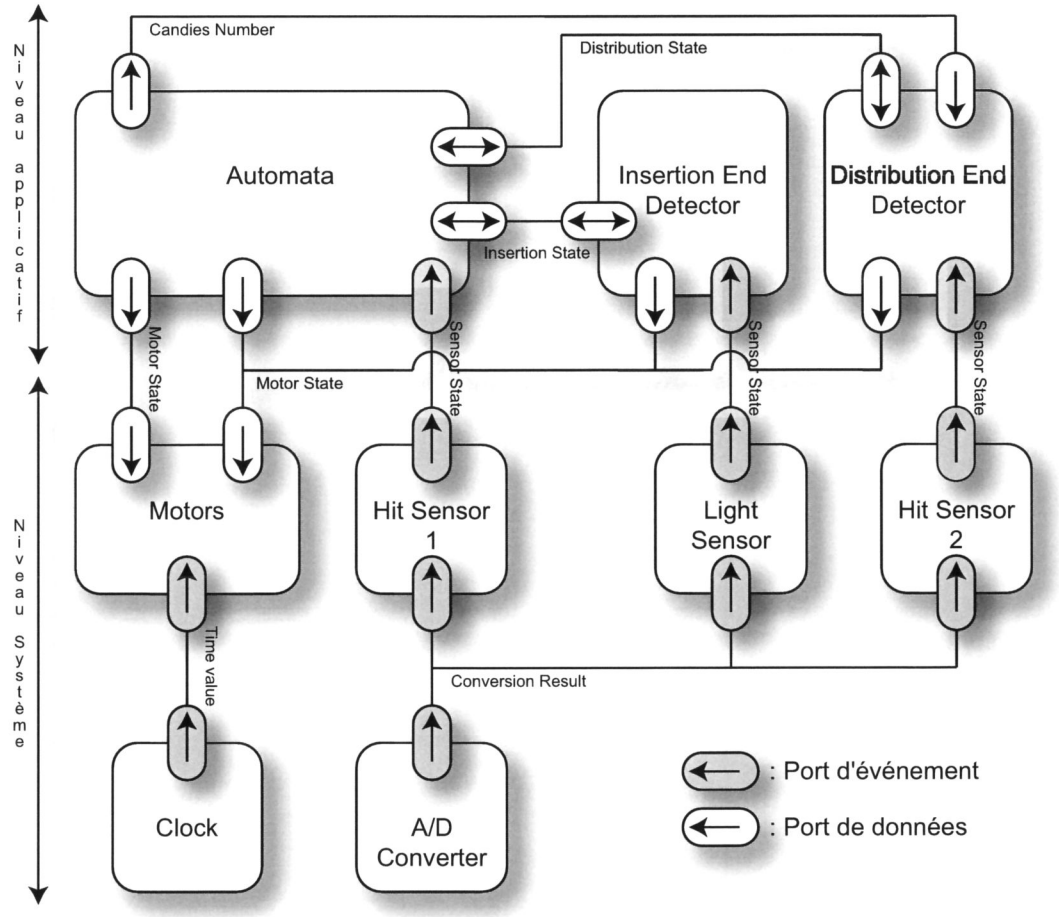


FIG. 7.2 – *Le noyau du distributeur de friandises*

Quatre commandes sont ainsi possibles : la mise en roue libre (l'axe du moteur est libre de tourner dans toutes les directions), la mise en marche dans une direction, la mise en marche dans l'autre direction, l'arrêt (l'axe du moteur est bloqué dans sa position et ne peut donc tourner librement sur lui-même). Le pilote des moteurs est en charge de la mise à jour périodique du registre en fonction des informations qui lui sont communiquées par les composants applicatifs. Il comporte donc deux ports d'entrée, sur lequel les composants applicatifs envoient les états des deux moteurs du système. Son port de réception d'événement est connecté à l'horloge. Lorsqu'il reçoit un événement de celle-ci, il modifie le registre des moteurs en fonction des informations disponibles sur ses ports d'entrée.

Les capteurs de choc et de lumière sont chargés de l'interprétation d'une valeur reçue du convertisseur A/D en une valeur ayant un sens dans leur contexte. Le convertisseur A/D effectue à tour de rôle une conversion pour chacun des trois ports d'entrée du RCX et génère un événement contenant la valeur correspondante et le port concerné. L'événement est capté par le pilote concerné, qui se charge de l'interprétation de la valeur dans son contexte. Selon les cas, il génère ensuite un événement. Les pilotes des capteurs de choc et de lumière génèrent ainsi un événement lorsque les capteurs matériels associés changent d'état (c'est à dire quand il passe, par exemple, de relâché à pressé pour le capteur de chocs ou de blanc à noir pour le capteur de lumière).

#### 7.5.3.2 Partie applicative

La partie applicative est constituée de trois composants : `Insertion End Detector`, `Distribution End Detector` et `Automata`.

Le composant `Insertion End Detector` permet, comme son nom l'indique, de détecter la fin de l'insertion d'une carte dans le lecteur. Il comporte un port de réception d'événement relié au pilote du capteur de lumière. La carte, lorsqu'elle passe devant le capteur de lumière, provoque une brusque hausse de l'intensité lumineuse perçue par le capteur. Lorsque cela se produit, le détecteur en est averti par un événement. Il ordonne alors au moteur de changer de direction (il envoie la commande au moteur sur un port de sortie), puis signale au composant `Automata` la fin de l'insertion de la carte (il envoie cette information sur un port d'entrée / sortie prévu à cet effet).

Le composant `Distribution End Detector` permet lui de détecter la fin de la distribution de friandises. Il comporte un port de réception d'événement relié au pilote du capteur de chocs 2 (c'est à dire celui du distributeur). Le nombre de friandises à distribuer lui est indiqué sur un port d'entrée. Chaque fois qu'une friandise est distribuée, une barre fixée sur l'axe du moteur vient heurter le

capteur de chocs, déclenchant l'émission d'un événement. Cet événement est capté par le détecteur qui incrémente un compteur local et compare sa valeur avec le nombre total de friandises à distribuer. S'ils sont égaux, il ordonne au moteur de s'arrêter (il envoie la commande au moteur sur un port de sortie), puis signale au composant `Automata` la fin de la distribution (il envoie cette information sur un port d'entrée / sortie prévu à cet effet).

Le principal composant du robot est `Automata`. Celui-ci définit l'automate principal du programme. Il comporte un port de réception d'événement connecté au pilote du capteur de choc, trois ports de sortie connectés respectivement au moteur 1, au moteur 2 et au composant `Distribution End Detector` et deux ports d'entrée / sortie connectés respectivement aux composants `Insertion End Detector` et `Distribution End Detector`.

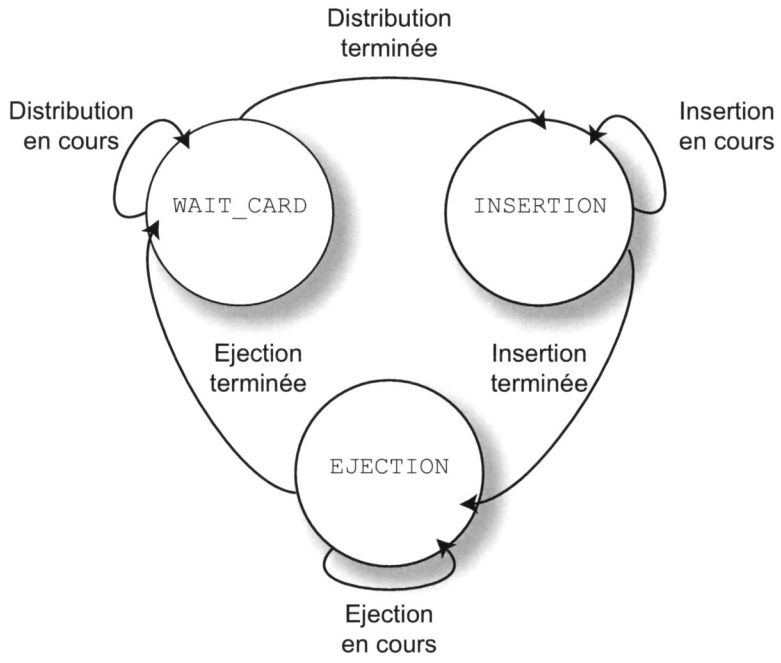


FIG. 7.3 – *L'automate principal du robot*

L'automate (voir figure 7.3) est déclenché chaque fois que le détecteur de chocs 1 est enfoncé. L'automate implémenté par le composant, très simple, comporte trois états dont voici la signification :

`WAIT_CARD` Le distributeur attend l'insertion d'une carte par un utilisateur. Aucun moteur n'est en marche. La réception d'un événement signale le début de l'insertion d'une carte dans le lecteur. Si une distribution de friandise est en cours (cette information est vérifiée auprès du port d'entrée / sor-

tie connecté au composant `Insertion End Detector`), l'événement est ignoré. Dans le cas contraire, le composant ordonne alors au moteur 1 de se mettre en marche pour entraîner la carte à l'intérieur de l'appareil (il envoie la commande au moteur sur un port de sortie). Il initialise le nombre de friandises à distribuer à `1`, signale que l'insertion est en cours sur son port d'entrée / sortie relié au composant `Insertion End Detector` et passe dans l'état `INSERTION`.

`INSERTION` Une carte est en cours d'insertion dans le lecteur. La réception d'un événement signale qu'une plaque 1x1, fixée sur la carte, vient d'heurter le capteur. Si le composant `Insertion End Detector` n'a toujours pas signalé la fin de l'insertion, le nombre de friandises à distribuer, `n`, est incrémenté. Dans le cas contraire, le moteur a changé de sens et la carte est en cours d'éjection. Le nombre de friandises `n` est alors envoyé sur le port de sortie. Le composant passe alors en mode `EJECTION` et un compteur local est initialisé à `n-1`.

`EJECTION` Une carte est en cours d'éjection du lecteur. Le moteur branché sur le port 1 fait reculer la carte en dehors du lecteur. La réception d'un événement signale qu'une plaque 1x1, fixée sur la carte, vient d'heurter le capteur. Le compteur local initialisé à (n-1) dans l'état précédent est décrémenté. S'il vaut zéro, c'est que l'éjection de la carte est maintenant terminée et que la distribution des friandises peut commencer. Le composant ordonne alors au moteur 2 de se mettre en marche pour commencer la distribution, signale que la distribution est en cours sur son port d'entrée / sortie relié au composant `Distribution End Detector` et passe dans l'état `WAIT_CARD`.

### 7.5.4 Scénario-type d'utilisation

Alors qu'aucune distribution n'est en cours, un utilisateur rentre une carte dans le lecteur. Cela provoque un choc sur le capteur 1, qui déclenche une interruption et l'émission d'un événement. L'événement est reçu par le pilote du capteur qui le transmet à l'automate. L'automate vérifie qu'aucune distribution n'est en cours demande au moteur 1 d'avancer et change d'état. Le pilote des moteurs, relié aux événements de l'horloge, détecte un changement de son état et démarre le moteur 1.

Chaque fois qu'une plaque 1x1 heurte le capteur 1, l'automate est lancé et incrémente le compteur de friandises à distribuer.

La fin de l'insertion de la carte est détectée par le capteur de lumière. Celui-ci génère une interruption et l'émission d'un événement. L'événement est reçu par

le pilote du capteur qui le transmet au détecteur de fin d'insertion. Le détecteur de fin d'insertion, lorsqu'il reçoit l'événement, demande au moteur 1 de changer de sens de rotation. Le pilote des moteurs effectue le changement ce qui provoque l'éjection de la carte.

La carte, en reculant, va provoquer à nouveau plusieurs chocs sur le capteur 1. Au premier choc, il envoie le nombre de friandises à distribuer vers le détecteur de fin de distribution, change d'état et initialise le comptage des chocs. Lorsque le nombre des chocs correspond au nombre de friandises, il donne l'ordre au moteur 1 de s'arrêter, signale la fin de l'éjection et le début de la distribution, et demande au moteur 2 de se mettre en marche pour commencer la distribution. Le pilote des moteurs effectue les changements, et la distribution commence.

La rotation du moteur 2 entraîne un bras qui, à chaque tour, vient heurter le capteur 2. Celui-ci provoque une interruption et l'émission d'un événement. L'événement est reçu par le pilote du capteur qui le transfère au détecteur de fin de distribution. Celui-ci incrémente un compteur local et le compare au nombre de friandises à distribuer qu'il a reçu. Si ces deux nombres sont égaux, il demande au moteur 2 de s'arrêter et signale la fin de la distribution. Le pilote des moteurs effectue le changement ce qui met fin à la distribution.

### 7.5.5 Reconfiguration dynamique

Le fonctionnement du robot peut être dynamiquement modifié grâce aux primitives de reconfiguration intégrées aux composants du système.

#### 7.5.5.1 Exemple de reconfiguration

Nous désirons maintenant rendre le distributeur capable de déterminer si un utilisateur a le droit de retirer des friandises ou non. Ainsi, nous introduisons la règle suivante : seules les cartes de retrait comportant une première plaque 1x1 de couleur rouge ont le droit de retirer, toute autre carte doit être éjectée immédiatement.

Les composants logiciels affectés par ce changement de politique sont au nombre de deux : il s'agit des composants `Insertion End Detector` et `Automata`. En effet, le composant `Insertion End Detector` ne doit plus se contenter de détecter un changement d'intensité lumineuse, mais doit également déterminer à quelle couleur correspond l'intensité lumineuse perçue et si cette couleur est autorisée à retirer des friandises. Il doit en outre transmettre cette information au composant `Automata` qui se chargera de distribuer ou non les friandises.

#### 7.5.5.2 Déroulement de la reconfiguration

Supposons la reconfiguration dynamique du noyau déclenchée par un événement extérieur associé à un des boutons du robot. Lorsque l'utilisateur appuie sur le bouton, une routine associée à l'interruption démarre le processus de reconfiguration. Celui-ci comporte les étapes suivantes :

- l'arrêt du composite englobant;
- le remplacement des composants `Insertion End Detector` et `Automata`;
- le redémarrage du composite englobant;

En premier lieu, l'exécution du noyau est arrêtée dans un état cohérent grâce à l'appel de la fonction `stop` de l'interface `LifeCycleController` du composite englobant. La méthode `removeSubComponent` de l'interface `ContentController` permet ensuite de supprimer les deux composants à remplacer, et la méthode `addSubComponent` permet de les remplacer par leur nouvelles versions. Enfin, un appel à la méthode `Start` de `LifeCycleController` redémarre noyau dans sa nouvelle forme.

## 7.6 Bilan de l'expérimentation

Grâce à l'expérimentation que nous avons menée autour de la construction d'un noyau de système à destination du Lego RCX, nous avons pu évaluer les qualités et les faiblesses des modifications apportées au prototype initial de THINK. Les évaluations ont porté sur trois points : le modèle de composition, les fonctions de reconfiguration et la bibliothèque Embedded-THINK.

### 7.6.1 Modèle de composition

Le modèle de composition original de THINK a été largement enrichi. L'évolution majeure vis à vis du modèle de composition initial est constituée par le passage d'un modèle à plat à un modèle hiérarchique gouverné par une séparation contrôleur / contenu.

La hiérarchisation du modèle a permis en premier lieu de factoriser l'implémentation de certaines propriétés pour un groupe de composants qui les partage. On voit apparaître ici la notion de domaine : un domaine est une zone logicielle spécifique garantissant un certain nombre de propriétés à toute entité se situant dans cette zone. Un composant composite peut être assimilé à un domaine garantissant un certain nombre de propriétés à ses sous-composants.

Le deuxième intérêt direct de la hiérarchisation concerne la possibilité laissée au développeur d'un système d'intercepter toute interaction entre un com-

posant et son environnement. L'interception est rendue possible par l'interposition dynamique de composants spéciaux (appelés intercepteurs) entre l'interface fonctionnelle d'un composant et son environnement. Elle permet d'établir une barrière d'abstraction entre les deux et fournit ainsi le cadre idéal à l'implémentation de politiques de sécurité ou de fonctions d'observation.

Enfin, l'introduction d'une séparation explicite entre un contrôleur (un composite) et un contenu (un ou des composant(s)) permet d'implémenter des fonctions de contrôle avancées. L'exécution d'un composant est ainsi totalement contrôlée par son composite englobant. Le composite peut par exemple contrôler les ressources allouées à ses sous-composants. Il est capable d'organiser et de contrôler à sa guise l'exécution de ses sous-composants.

Ces trois points, intimement liés, se sont avérés indispensables à l'implémentation des fonctions de reconfiguration dynamiques dont on établit le bilan dans la section suivante.

### 7.6.2 Outils de reconfiguration

La hiérarchisation du modèle de composition de THINK fournissait le support idéal à l'intégration de fonctions de reconfiguration dynamiques : nous l'avons exploitée à cette fin. Les reconfigurations que nous envisagions étaient à la fois structurelles, c'est à dire modifiant le schéma organisationnel des composants d'un système, et comportementales, c'est à dire modifiant la façon dont étaient implémentées certaines fonctions du système.

Il existait déjà, dans le domaine des intergiciels, un modèle offrant des outils de reconfiguration dynamique s'appuyant sur un modèle de composition hiérarchique : le modèle Fractal. Nous avons importé ce modèle au sein de THINK en l'adaptant aux contraintes d'un système d'exploitation.

Le modèle Fractal est basée sur l'idée originale suivante : chaque composant est libre de définir sa propre politique de reconfiguration vis à vis de son contenu. Ainsi, tout le système n'est pas forcément reconfigurable, et tout ce qui est reconfigurable dans un système ne l'est pas forcément de manière identique. Cette caractéristique est intéressante, car elle permet (à l'inverse de nombreux systèmes reconfigurables) de ne faire payer le coût des mécanismes de reconfiguration qu'aux composants le nécessitant strictement, et d'offrir des mécanismes de reconfiguration adaptés à chaque configuration logicielle.

Si chaque composant maîtrise l'implémentation de ses fonctions de reconfiguration, les interfaces de reconfiguration sont uniformes sur l'ensemble du système, permettant ainsi à chaque composant d'agir sur n'importe quel autre composant pour peu que ce dernier l'y autorise (c'est à dire pour peu qu'il ex-

porte une interface de reconfiguration dans un contexte de désignation auquel le composant a accès). Les interfaces de configuration proposées par le modèle ont été pensées pour offrir l'ensemble des opérations de reconfiguration envisageables sur un composant, mais chaque composant est toutefois libre, s'il en a le besoin spécifique, d'offrir une interface de reconfiguration particulière, adaptée à un usage précis.

Ces mécanismes, utilisés dans l'expérimentation présentée dans ce chapitre, ont prouvé leur efficacité : ils nous ont permis de reconfigurer un noyau de système dans un environnement pourtant très contraint : seuls les composants nécessitant une reconfiguration étaient pourvus des mécanismes permettant cette reconfiguration, allégeant ainsi tout le reste du système qui n'aurait pas pu supporter leur coût s'ils avaient été appliqués à son ensemble.

### 7.6.3 Bibliothèque Embedded-THINK

Les travaux liés au développement de la bibliothèque Embedded-THINK avaient pour premier objectif de prouver la souplesse de l'architecture de THINK, même dans des environnements fortement contraints. L'expérience fut concluante; l'architecture générale de THINK nous a permis de développer nos propres techniques d'interaction (sous forme de fabriques de liaisons) et d'aboutir à un nouveau modèle de composition qui reste compatible avec le modèle original mais en projette les fonctionnalités sur une zone précise du spectre de développement de systèmes, les systèmes embarqués fortement contraints.

Un des points clés de l'architecture THINK est en effet sa capacité à s'adapter à des contextes ultra-spécialisés. Plutôt que d'opter pour un système générique, qui se serait sûrement avéré trop général dans la plupart des utilisations, nous avons fait le choix d'offrir aux développeur les structures logicielles de base lui permettant de développer son propre système, répondant exactement à ses besoins (que nous ne pouvons connaître à l'avance).

Pour développer Embedded-THINK, nous nous sommes mis dans la peau d'un développeur de systèmes embarqués fortement contraints. Nous avons identifié un certain nombre de besoins en terme de fonctionnalités logicielles que nous avons traduit en concepts (et donc composants) THINK. Cette bibliothèque fut ensuite utilisée pour le développement d'un système spécifique pour le Lego RCX, et son utilisation a largement facilité le travail de conception et d'implémentation.

# Chapitre 8

# Conclusion

L'omniprésence future de l'informatique lance de nouveaux défis à la recherche en matière de systèmes d'exploitation. Les systèmes traditionnels monolithiques, lourds et peu flexibles, doivent faire place à de nouvelles architectures modulaires, légères et adaptables. Cela ne peut être fait que par la conception de nouvelles techniques de structuration logicielle.

Une des techniques de structuration les plus efficaces en matière de flexibilité est représenté par l'approche à composants, qui consiste à décomposer en modules indépendants les différentes parties d'un logiciel. Cette technique, quand elle est couplée à des outils de composition efficaces, peut s'avérer extrêmement intéressante. Pourtant, dans l'esprit des chercheurs, elle est trop longtemps restée irrémédiablement liée aux intergiciels, car trop peu performante pour être intégrée dans les couches basses d'un système.

Les travaux présentés dans cette thèse prennent le contre-pied de cette position en proposant l'intégration d'un canevas de structuration à composants offrant des fonctions de reconfiguration dynamique au sein même des systèmes d'exploitation. L'architecture proposée exploite au maximum les techniques de structuration à composants pour offrir une meilleure flexibilité lors de la construction d'un système. La philosophie suivie, qui consiste à limiter au maximum les abstractions définies par le noyau, la rend en outre performante.

Pour prouver l'efficacité du modèle imaginé, celui-ci a ensuite donné lieu à l'implémentation d'un prototype et à son utilisation pour le développement d'une bibliothèque de composants permettant le développement de systèmes d'exploitation à destination d'architectures matérielles fortement contraintes. Le résultat s'est avéré payant et a été vérifié par la génération d'un système d'exploitation pour le Lego RCX, une mini-brique Lego programmable permettant la construction de robots autonomes.

## 8.1 Bilan général

Les résultats auxquels nous sommes parvenus aujourd'hui nous permettent de dresser un premier bilan sur l'opportunité d'utiliser des techniques de structuration à composants dans le cadre de la construction de systèmes d'exploitation.

Les expérimentations menées au cours de cette thèse ont non seulement démontré qu'il était possible d'intégrer un modèle de composition au sein même d'un système d'exploitation, mais elles ont surtout prouvé que l'introduction de tels concepts était largement bénéfique au système et pouvait s'effectuer à moindre coût.

Les bénéfices de l'approche à composants s'évaluent d'abord en terme de flexibilité. THINK permet la décomposition d'un système en unités indépendantes selon des règles systématiques de construction. Il est ainsi possible de construire un système à la carte, en choisissant et en assemblant les composants offrant les fonctionnalités requises par le système. Le système obtenu est ultra-spécialisé : il répond exactement aux besoins de l'utilisateur sans pour autant s'encombrer de fonctionnalités superflues qui auraient pour effet d'alourdir l'ensemble. Grâce à THINK, il nous a été ainsi possible de construire une bibliothèque de composants dédiée à la construction de systèmes d'exploitations pour architectures embarquées fortement contraintes, et de l'utiliser pour développer un système d'exploitation complet pour un environnement matériel très fortement contraint, le Lego RCX.

L'approche à composants offre en outre le support d'exécution idéal pour l'implantation de fonctions d'administration avancées. Parmi elles, les fonctions de reconfiguration dynamique du système sont particulièrement intéressantes. Nous nous sommes ainsi appuyés sur le modèle de composition de THINK pour offrir un canevas de reconfiguration dynamique permettant des évolutions à la fois structurelles et comportementales du système. Ces fonctions de reconfiguration ont été ensuite utilisées pour faire évoluer dynamiquement le noyau de système que nous avions développé pour le Lego RCX. L'introduction de fonctions d'administration avancées dans un cadre aussi contraint matériellement que le Lego RCX ne fut possible que par la flexibilité du modèle Think, qui permet d'un part de ne faire payer le coût des fonctions de reconfiguration qu'aux composants les utilisant, et d'autre part d'adapter l'implémentation des fonctions de reconfiguration pour ne pas trop pénaliser le système.

Le paradigme de structuration à composants, lorsqu'il est appliqué aux noyaux de systèmes d'exploitation, nous apparaît ainsi offrir de larges perspectives et semble devoir s'imposer, dans l'avenir, comme l'une des techniques majeures de structuration dans ce domaine.

## 8.2 Perspectives

Les travaux présentés dans cette thèse ne sont toutefois que l'amorce de recherches plus vastes sur les nouvelles architectures logicielles pour systèmes d'exploitation. Celles-ci pourront s'effectuer selon deux axes différents : l'affinage du modèle de composition d'une part, et son exploitation dans des cadres applicatifs précis d'autre part.

Dans le premier cas, les recherches pourront par exemple porter sur la définition d'un modèle de gestion unifié des ressources du système. Par un tel modèle, on entend gérer de manière identique ressources matérielles (processeur, mémoire, disque dur, etc.) et logicielles (services systèmes, applications, etc.). Chaque ressource serait directement représentée par un composant et proposerait une interface de gestion uniforme permettant son allocation, son utilisation et sa destruction.

Une autre piste, toujours dans le premier axe de recherche, concerne la définition d'architectures logicielles sécurisées. L'encapsulation par composants permet en effet de créer des barrières artificielles entre l'intérieur d'un composant et son environnement. C'est donc un endroit idéal pour implémenter des politiques de sécurisation. Le composant pourrait alors être imaginé comme un domaine de protection explicitant une politique particulière de sécurisation.

Le second axe de recherche est plus applicatif. En effet, l'architecture THINK permet potentiellement l'implémentation de systèmes ultra-flexibles, même dans le cadre d'environnements matériels très spécialisés. Ainsi, il serait par exemple intéressant d'évaluer l'intérêt de l'utilisation de THINK, et notamment de son extension reconfigurable présentée dans ce document, dans le cadre de l'implémentation de routeurs actifs. En effet, un système pour routeur actif exige, pour des raisons de performance, que les techniques de routage soient implémentées au sein même du système. Toutefois, le système doit faire preuve de flexibilité pour prendre en compte de nouvelles politiques de routage de façon dynamique. Les primitives de reconfiguration dynamiques de THINK paraissent ainsi parfaitement adaptées à la gestion de systèmes de ce type.

Une autre application envisageable consiste en l'implémentation de machines virtuelles Java nues reconfigurables directement au-dessus de plate-formes matérielles embarquées. De tels environnements faciliteraient énormément le développement d'applications embarquées de qualité en proposant un langage sûr et à haut niveau d'abstraction dès les couches les plus basses d'un logiciel. Les constructeurs d'appareils de communication grand public (smartphones, assistants personnels, etc.), par exemple, marquent un réel intérêt pour de telles architectures, et il semble que THINK soit une plate-forme idéale pour une pre-

mière expérimentation dans ce sens.

# Bibliographie

[ABB+86] M. Acetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for unix development. In *Summer 1986 USENIX Conference*, pages 93–112, July 1986.

[ADE+03] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivan, C. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical modeling and analysis of embedded systems. In *IEEE*, volume 91, pages 11–28, January 2003.

[AWB+94] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object-interactions using composition- filters. In Lecture Notes in Computers Science, editor, *Object-based Distributed Processing*, pages 152–184. Springer-Verlag, 1994.

[BCD+97] G. Blair, G. Coulson, N. Davies, P. Robin, and T. Fitzpatrick. Adaptive middleware for mobile multimedia applications. In *8th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV '97)*, pages 259–273, St. Louis, Missouri, May 1997.

[BCD+98] G. Blair, G. Coulson, N. Davies, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing : Middleware'98*, pages 191–206, The Lake District, U.K., September 1998.

[Ber94] L. Bergmans. The composition filters object model. In *RICOT Symposium Enabling Objects for Industry*, June 1994.

[Blo83] T. Bloom. *Dynamic Module Replacement in a Distributed Programming System.* PhD thesis, Laboratory for Computer Science, The Massachussets Institute of Technology, March 1983.

[BSP+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.

[BSS+94] B. N. Bershad, E. G. Sirer, S. Savage, P. Pardyak, C. Chambers, and S. Eggers. SPIN an extensible microkernel for appplication-specific operating systems services. Technical Report TR-94-03-03, University of Washington, Seatle, Washington, February 1994.

[CC98] M. Clarke and G. Coulson. An architecture for dynamically extensible operating systems. In *4th International Conference on Configurable Distributed Systems*, Annapolis, Maryland, USA, 1998.

[CC99] M. Clarke and G. Coulson. Dynamic memory model reconfiguration in deimos. In *CFSE*, Rennes, France, June 1999.

[CCM02] CORBA Component Model, v3.0. Full specification, June 2002. http://www.omg.org/technology/documents/formal/components.htm.

[CM01] P. G. Cadence and G. Martin. Reliable estimation of execution time of embedded software. In *Design Automation and Test in Europe*, pages 580 – 589, Munich, Germany, 2001.

[Con03] ObjectWeb Consortium. Fractal specification 1.0-0, 2003. http://www.objectweb.org/fractal.

[COR01] The Common Object Request Broker: architecture and specification. OMG Document, December 2001. 2.6 edition.

[CT95] R. H. Campbell and S. M. Tan. uchoices: An object-oriented multimedia operating system. In *Fifth Workshop on Hot Topics in Operating Systems*, Orcas Island, WA, May 1995. IEEE Computer Society Press.

[EJB02] Enterprise JavaBeans(TM) Specification 2.1. Proposed Final Draft, August 2002.

[EK95] D. R. Engler and M. F. Kaashoek. Exterminate all operating system abstractions. In *5th workshop on hot topics in operating systems*, pages 78–83, Orcas Island, Washington, May 1995.

[EKO95] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[Fas01] Jean-Philippe Fassino. Think : Vers une Architecture de Systèmes Flexibles. Master's thesis, École Nationale Supérieure des Télécommunications, December 2001.

[FBH+97] B. Ford, G. Black, M. Hibler, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: a substrate for kernel and language

research. In *16th ACM Symposium on Operating System Principles*, St Malo, France, October 1997.

[FOA02] Boyer F., Charra O., and Senart A. *Les Intergiciels*, chapter Réflexivité pour les environnements adaptables, pages 73–92. Lavoisier, 2002.

[Frl96] S. Frlund. *Coordinating distributed objects - an actor-based approach to synchronization.* MIT Press, 1996.

[FS01a] J.-P. Fassino and J.-B. Stefani. Think : un noyau d'infrastructure répartie adaptable. In *Deuxième Conférence française sur les Systèmes d'Exploitations (CFSE-2)*, Paris, France, April 2001.

[FS01b] Jean-Philippe Fassino and Jean-Bernard Stefani. Think : un Noyau d'Infrastructure Répartie Adaptable. In *2ème Conférence française sur les Systèmes d'Exploitation (CFSE-2)*, Paris, France, April 2001.

[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.

[Gri00] G. Grimaud. *Camille : un système d'exploitation ouvert pour carte à microprocesseur.* PhD thesis, Université des Sciences et Technologies de Lille, 2000.

[Han99] S. M. Hand. Self-paging in the nemesis operating system. In *3 rd USENIX Symp. on Operating Systems Design and Implementation*, pages 73–86, New Orleans, LA, February 1999.

[HF98] J. Helander and A. Forin. Mmlite: a highly componentized system architecture. In *8th ACM SIGOPS European workshop on Support for composing distributed applications*, pages 96–103, Sintra, Portugal, 1998.

[HP91] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[ILY95] J. Itoh, R. Lea, and Y. Yokote. Using meta-objects to support optimisation in the apertos operating system. In *USENIX Conference on Object-Oriented Technologies (COOTS '95)*, 1995.

[KC99] F. Kon and R. H. Campbell. Supporting automatic configuration of component-based distributed systems. In *5th USENIX Conference on Object-Oriented Technologies and Systems*, pages 175–187, San Diego, CA, May 1999.

[KC00] F. Kon and R. H. Campbell. Dependence management in component-based distributed systems. *IEEE Concurrency*, 8(1):26–36, January-March 2000.

[KCM+00] F. Kon, R. Campbell, M. D. Mickunas, K. Nahrstedt, and F. J. Ballesteros. 2k: A distributed operating system for dynamic heterogeneous environments. In *9th IEEE International Symposium on High Performance Distributed Computing*, Pittsburgh, August 2000.

[KdRB91] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of The Metaobject Protocol.* The MIT Press, Cambridge, Massachusetts, 1991.

[KLM+97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[Kra02] S. Krakowiak. The Jonathan Tutorial. ObjectWeb, 2002. http://www.objectweb.org/jonathan/doc/tutorial.

[KRL+00] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhães, and R. H. Campbell. Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, New York, April 2000.

[KYH+01] F. Kon, T. Yamane, C. K. Hess, R. H. Campbell, and M. D. Mickunas. Dynamic resource management and automatic configuration of distributed component systems. In *Conference on Object-Oriented Technologies and Systems*, 2001.

[LFS02] Dario Laverde, Giulio Ferrari, and Jurgen Stuber. *Programming Lego Mindstorms with Java.* Paperback, May 2002.

[Lib] Rcx tools. http://graphics.stanford.edu/~kekoa/rcx/tools.html.

[LVM95] D. C. Luckham, J. Vera, and S. Meldal. Three concepts of system architecture. Technical Report CSL–TR–95–67, Computer Systems Lab, Stanford University, July 1995.

[McA93] J. McAffer. The coda mop. In *OOPSLA'93 Workshop on Object-Oriented Reflection and Metalevel Architectures*, Washington, D.C., September 1993.

[MN88] P. Maes and D. Nardi. Meta-level architectures and reflection. In *Worshop on Meta-Level Architectures and Reflection*, Alghero, North-Holland, October 1988.

[MP96] D. Mosberger and L. L. Peterson. Making paths explicit in the Scout operating system. In *Operating Systems Design and Implementation*, pages 153–167, 1996.

[Nie00] Stig Nielsson. Introduction to the legOS Kernel, September 2000.

[PQB01] F. Peschanski, C. Queinnec, and J.-P. Briot. A typeful composition model for dynamic software architectures. Technical report, Univ. of Paris VI - Pierre et Marie Curie, 2001.

[RAA+88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Glen, M. Guillemont, F. Hermann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Chorus distributed operating systems. *Computing Systems Journal*, 1(4):305–370, 1988.

[RAA+92] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. He rrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the chorus distributed operating system. In *USENIX Workshop on Micro-kernels and other Kernel Architectures*, pages 39– 69, Seattle WA, U.S.A., April 1992.

[Rip03] C. Rippert. Protection dans les architectures de systèmes flexibles. Master's thesis, Université Joseph Fourier, 2003.

[RKC99] M. Román, F. Kon, and R. H. Campbell. Design and implementation of runtime reflection in communication middleware: the dynamictao case. In *ICDCS'99 Workshop on Middleware*, Austin, Texas, June 1999.

[SESS96] M. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *2nd Symposium on Operating System Design and Implementation*, pages 213–227, Seattle, WA, October 1996.

[SESS97] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Issues in extensible operating systems. Technical Report TR-18-97, Harvard University, 1997.

[Smi82] B. C. Smith. Reflection and semantics in a procedural language. Technical report, Laboratory of Computer Science, Massachusetts Institute of Technology, 1982.

[SR91] J. A. Stankovic and K. Ramamritham. The spring kernel: A new paradigm for real-time systems. *IEEE Software*, 8(3):62–72, May 1991.

[Sun] Sun. Java Remote Method Invocation - distributed computing for java. White paper.

[Szy98] C. Szyperski. *Component software: beyond object-oriented programming.* Addison-Wesley, 1998.

[TRLC95] S. M. Tan, D. K. Raila, W. S. Liao, and R. H. Campbell. Virtual hardware for operating system development. Technical report, University of Illinois at Urbana-Champaign, Department of Computer Science, September 1995.

[WY88] T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 306–315, San Diego, CA, September 1988.

[X.995] ITU-T Recommendation X.903. Odp reference model : Architecture. ISO/IEC Internation Standard 10746-3, 1995.

[Yok92] Y. Yokote. The apertos reflective operating system: The concept and its implementation. In *Object-Oriented Programming Systems, Languages and Applications*, pages 414–434, 1992.