Conception et Implantation d'un Environnement de Développement de Logiciels à Base de Composants, Applications aux Systèmes Multiprocesseurs sur Puce

Ali Erdem Özcan

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N˚ attribué par la bibliothèque

THÈSE

pour obtenir le grade de

DOCTEUR DE l'INPG

Spécialité : « Informatique : Systèmes et Logiciels »

préparée au laboratoire LSR-IMAG, projet SARDES,
dans le cadre de l'Ecole Doctorale
« Mathématiques Sciences et Technologies de l'Information »

présentée et soutenue publiquement par

Ali Erdem ÖZCAN

le 28 Mars 2007

Conception et Implantation d'un Environnement de Développement de Logiciels à Base de Composants

Applications aux Systèmes Multiprocesseurs sur Puce

Directeur de thèse :

Jean-Bernard STEFANI

JURY

M.	Charles	CONSEL	Président
M.	Bertil	FOLLIOT	Rapporteur
M.	Lionel	SEINTURIER	Rapporteur
M.	Philippe	GUILLAUME	Examinateur
M.	Jacques	MOSSIÈRE	Examinateur
M.	Jean-Bernard	STEFANI	Directeur de thèse

Résumé

Ces travaux de thèse définissent un environnement de développement ouvert et extensible pour la conception de logiciels à base de composants. L'environnement se présente comme une chaîne de compilation d'architectures logicielles, acceptant des architectures écrites dans des langages différents et fournissant des fonctionnalités comme la génération de code ou le déploiement. L'extensibilité de l'outil est assurée par une architecture à base de composants implantant des patrons de programmation extensibles et supportant un mécanisme de plug-in pour intégrer des extensions de tierces parties. L'utilisation de l'outil est illustrée au travers deux cadres applicatifs ayant pour trame les systèmes sur puces. La première illustre le développement de systèmes d'exploitation pour ceux-ci. La deuxième illustre la définition d'un nouveau langage facilitant l'expression de la synchronisation au sein d'applications de traitement de flux multimédia réparties.

Abstract

Our work aims at the definition of an open and extensible development environment for supporting the design and implementation of component based software. Our proposition takes the form of a component based software architecture compilation toolset, accepting architectures described in different languages, and providing different functionalities such as code generation and deployment. The extensibility of the toolset is achieved thanks to a fine-grained component-based architecture, implementing a set of extensible programming patterns, and to a plug-in support for loading third party extensions. Two evaluation use-cases set up in the context of system-on-chips are presented in order to illustrate the effectiveness of our approach. The first use-case presents the development of operating system kernels. The second one describes the extension of the toolset with a synchronization pattern description language for easing the development of distributed streaming applications.

Remerciements

Je tiens tout d'abord à remercier Charles Consel, professeur à l'Ecole Nationale Supérieure d'Electronique, Informatique et Radiocommunications de Bordeaux, de me faire l'honneur de présider ce jury.

Je remercie Bertil Folliot, professeur à l'Université Pierre et Marie Curie, Paris VI, et Lionel Seinturier, professeur à l'Université de Lille, d'avoir accepté d'être rapporteurs de cette thèse et d'avoir effectué une évaluation approfondie de mes travaux.

Je remercie également Jacques Mossière, professeur à l'Institut National Polytechnique de Grenoble et Philippe Guillaume, responsable des activités de recherche et développement des modèles de programmation et des systèmes d'exploitation dans la société STMicroelectronics, d'avoir accepté d'être examinateurs de mes travaux.

Je ne saurais assez remercier Jean-Bernard Stefani, mon directeur de thèse. Sa vision scientifique, et ses précieux conseils m'ont guidé tout au long de ces années de thèse. Il a toujours été très disponible et curieux pour discuter de mes travaux, et il m'a toujours encouragé et motivé. Enfin, je tiens à le remercier également sur le plan personnel, pour les nombreuses discussions que nous avons eues.

Je remercie très chaleureusement Philippe Guillaume, mon responsable industriel, pour sa motivation, son ouverture et son encouragement. Il a créé pour moi un environnement de travail très confortable, avec un bon équilibre entre la recherche académique et industrielle. C'est une grande chance que j'ai eue de pouvoir travailler sous sa direction, aussi bien sur le plan technique que sur le plan humain. Je voudrais également remercier Marco Cornero, directeur du Groupe COSA de STMicroelectronics, pour sa confiance, et pour l'intérêt qu'il a manifesté pour mes travaux.

Bien qu'elle soit un travail personnel, une thèse est néanmoins le fruit de trois années pendant lesquelles le thésard collabore avec de nombreuses personnes. À défaut de pouvoir faire une liste exhaustive, je voudrais remercier :

- Matthieu Leclercq qui a joint notre équipe il y a un an en tant qu'ingénieur de développement. Matthieu a joué un rôle très important dans la finalisation du canevas logiciel présenté dans ce document. Il a travaillé avec un enthousiasme remarquable pour faire d'un prototype, un logiciel de qualité industrielle. C'est une chance immense de travailler avec lui. Je n'oublierai jamais nos discussions techniques où l'on a quelques fois eut l'impression de redécouvrir ce qu'est l'informatique.
- Les membres de l'équipe Sardes pour leur accueil chaleureux et pour les interactions que nous avons eues. Oussama Layaïda avec qui j'ai commencé à travailler sur le développement d'applications multimédia, Vivien Quéma avec qui j'ai eu de nombreuses discussions techniques, Alan Schmitt et Frédéric Mayot avec qui j'ai initié les travaux sur JoinDSL, et Juraj Polakovic avec qui j'ai travaillé sur la reconfiguration dynamique de systèmes d'exploitation ; Renaud Lachaize avec qui j'ai beaucoup discuté sur les aspects systèmes. Jacques Mossière qui m'a toujours aidé, non seulement sur le plan professionnel, mais aussi sur le plan personnel. Sacha Krakowiak que j'ai eu la grande chance de côtoyer. Enfin, Sébastien Jean, Fabienne Boyer, Sara Bouchenak, Noël de Palma, Jakub Kornas, Christophe Taton, Michael Lienhardt, Didier Donsez pour leur contact chaleureux.
- Les membres du laboratoire AST pour leur accueil et leur amitié ; Germain Haugou avec qui j'ai travaillé sur Think4L ; Erven Rohou, Roberto Costa et François Naçabal avec qui j'ai collaboré sur les composants pour .Net ; Thierry Lepley, Stéphane Curaba, Jean-Marc Zins avec qui j'ai eu de nombreuses discussions enrichissantes. Thierry Strudel avec qui j'ai eu la chance de travailler et qui a certainement eu un impact important sur ma vision de l'informatique. Enfin, Marcello

Coppola qui m'a accueilli chaleureusement dans le laboratoire durant ces années.
– Les membres des projets Think et Fractal avec qui j'ai eu de nombreux échanges très constructifs.
– Mario Diaz-Nava qui m'a motivé pour entamer une thèse après m'avoir encadré en DEA, et qui a joué un rôle important dans la mise en place de cette thèse CIFRE avec la société STMicroelectronics.

Enfin, sur le plan personnel, je voudrais exprimer mes amitiés à :
– Christophe Le Gal qui a sans doute été un tournant important de ma vie. J'ai eu la chance de le côtoyer au cours d'un stage que j'ai effectué tout au début de mes études en France. Il est la personne qui m'a réellement introduit à l'informatique et a été pour moi un ami très précieux.
– Levent Topaç, Zeynep Eraydın, Taylan Genç, Pınar Özdemir et Yücel Balım pour leur amitié. Nous étions tout petits quand nous nous sommes rencontrés, et avons grandi ensemble. Bien que nous vivions dans des villes lointaines depuis plusieurs années, je garde en moi un souvenir mémorable des années que nous avons passées ensemble.
– Mes amis Onur Marşan, Burçak Kursan, Kerim Nadir, Onur İnanç, Gönenç Onay, Olivier Ptak et Vincent Mazel pour leur gentillesse.
– Mes amis thésards Oussama Layaïda et Vivien Quéma avec qui j'ai passé trois années formidables.
– And finally, my deepest thanks go to my family. To my mother, who changed my life by forcing me to study in France : she was absolutely right, even if we had some difficult moments. To my father, who has always been a role model for me. I don't know how to thank him for all that he did for me, every day of my life. And to my uncle, Yi*ğ*it Gündüç who is the first to motivate me to follow my studies in computer sciences. Mon frère qui a toujours été avec moi et qui m'a supporté sans cesse durant ces trois années de travail intensif. Enfin, mon *yavmur*, Ayşegül, pour son amour et son support inestimable. La vie est plus belle avec elle, et le restera toujours.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Sommaire

La conception des systèmes sur puce ou "SoC"[1] tend vers une complexité toujours croisante ; il en est ainsi depuis les débuts de l'industrie du semi-conducteur. Conçus sur mesure en intégrant de nombreux cœurs de processeurs et des accélérateurs matériels aux architectures dédiées, ces plates-formes ont satisfait depuis une dizaine d'années des besoins applicatifs hautement spécifiques manifestés par des artefacts tels les téléphones portables ou les décodeurs (set-top-box). Cependant, l'évolution rapide des normes applicatives, la compétitivité féroce et l'étroitesse des fenêtres temporelles d'accès au marché, les coûts prohibitifs de fabrication dus à la complexité de conception comme au prix des masques de photolithogravure s'associent aujourd'hui en une réalité difficile à appréhender. Ces réalités obligent les concepteurs à opter pour des circuits plus souples et adaptables afin de produire des familles de produits qui peuvent êtres déclinés vers plusieurs types d'applications, plusieurs types de marchés. Au sein de cette "petite révolution", la programmabilité, et par conséquent la réutilisabilité des circuits, en privilégiant l'emploie de processeurs de calcul intensif plutôt que des accélérateurs matériels, apparaît comme une solution naturelle : ce sont les systèmes multiprocesseurs sur puce (MPSoC[2]). Cette mouvance a considérablement augmenté la quantité de logiciel déployé sur les puces et conduit la conception de logiciel pour ces circuits au premier plan des préoccupations associées au flot de conception global de ces circuits qui de puces sont passés au rang de "mastodontes" miniaturisés.

Le développement de logiciel pour les MPSoC n'a rien de trivial, ceci pour deux raisons essentielles. La première découle de la nature des ressources de calcul en présence. La complexe hétérogénéité des

[1] SoC pour System-on-Chip.
[2] MPSoC pour Multi-Processor System-on-Chip.

plates-formes (e.g. processeurs dédiés de multiple nature, architecture d'interconnexion dédiée, etc.) rend l'ingénierie du logiciel embarqué elle même éminemment complexe, nécessitant, outre l'adaptation souvent manuelle du logiciel à des accélérateurs matériels dédiés, l'usage de chaînes d'outils de développement aussi hétérogènes dans leurs capacités et leur emploi que les architectures qu'elles ciblent. Les contraintes énergétiques ont pour conséquence d'obliger les concepteurs à répartir les ressources de calcul en les associant à des stratégies complexes de gestion de la consommation, à l'opposé de la stratégie considérant un mono-processeur ultra puissant et ultra consommant, stratégie commune dans les ordinateurs à usage général. La seconde raison découle des fonctionnalités applicatives assumées par ces circuits. En effet, le nombre d'applications supportées (codecs multimédia, algorihtmes d'encryptage, ...) ainsi que leur évolution rapide pose d'importants problèmes d'intégration, de maintenance et d'évolution. Dans ce contexte, les techniques de développement traditionnellement employées ne suffisent plus à maîtriser les portefeuilles de logiciel croissant et la pression du marché vers toujours plus de fonctionnalités, toujours plus de complexité.

Cette thèse s'inscrit dans le cadre de cette complexité de conception de logiciels à destination des MPSoC. Le constat de cette complexité motive la recherche de solutions originales tendant à une maîtrise plus grande de celle-ci et à la définition de méthodologies de conceptions permettant d'appréhender l'intégralité du processus de développement de logiciel à destination des MPSoC. Après analyse de la réalité des architectures aujourd'hui en vigueur dans les MPSoC et leur projection à moyen/long termes, enrichie de l'étude de pratiques logicielles associées, nous faisons le constat que les problèmes peuvent-être abordés sous un angle similaire à celui adopté dans le domaine des systèmes informatiques répartis, et que les solutions élaborées dans ce domaine sont sans doute pertinentes transposées au domaine plus contraint des MPSoC. Par conséquent, nous optons pour une démarche qui consiste à analyser les propositions existantes afin de les appliquer à l'échelle des systèmes sur puce. Plus particulièrement, nous nous intéressons à la programmation à base de composants suivant l'intuition que cette approche dispose de bases très prometteuses pour la maîtrise de l'assemblage, de la configuration, de la maintenance et de l'évolution des systèmes logiciels complexes tels que manisfestés dans les MPSoC, permettant de surcroi un raisonnement au niveau de l'architecture du logiciel. L'objectif de cette thèse est donc plus précisément de contribuer au processus de développement de logiciels pour systèmes sur puce en proposant l'utilisation d'un modèle de composants ouvert. Nous défendons l'idée que l'adoption d'un même modèle de composants par l'ensemble de la chaîne de production de logiciel est possible et que celle-ci peut considérablement faciliter le processus d'intégration. Or, une grande partie des apports de la programmation à base de composants repose sur les outils d'aide à la conception, ce qui nécessite la définition d'un canevas logiciel pour faciliter la mise en œuvre de tels outils de programmation.

Le travail présenté dans cette thèse a été effectué à cheval entre le groupe *Compilers, Operating Systems and Applications* (COSA) du laboratoire *Advanced System Technology* (AST) de la société ST-Microelectronics et le projet *System Architecture for Reflective Distributed Computing Environments* (SARDES) de l'Institut National de Recherche en Informatique et Automatique. Le groupe COSA qui apporte la problématique industrielle située au centre de nos travaux est en charge de définir les technologies de développement de logiciels destinées à être employées dans la production des systèmes sur puce de prochaines générations. SARDES est un projet de recherche spécialisé dans la construction d'infrastructures logicielles réparties à grande échelle, caractérisées par une très grande taille et une très grande hétérogénéité. Son domaine de recherche couvre à la fois le développement de technologies de programmation à base de composants réflexifs, et son utilisation pour la construction de serveurs autonomes dans le contexte suscité. Ce travail de thèse effectue la synthèse entre ces deux activités, d'une part en apportant une réflexion alternative aux technologies maîtrisées par le groupe COSA dans une logique de transfert de connaissance entre l'académie et l'industrie, et d'autre part en amenant un cadre applicatif nouveau à l'équipe SARDES.

Cette introduction est organisée comme suit. La section 1.1 décrit les caractéristiques des systèmes multiprocesseurs sur puce et présente la feuille de route émergente pour les systèmes des prochaines générations. La section 1.2 présente certaines pratiques de programmation de tels systèmes et tente d'analyser leurs limitations. Sur la base de cette analyse, la section 1.3 présente les motivations et les objectifs de nos travaux. Enfin, la section 1.4 présente l'organisation de ce manuscrit.

1.1 Caractéristiques des systèmes multiprocesseurs sur puce

Depuis leur émergence il y a une dizaine d'années, les systèmes-sur-puce ont été largement utilisés dans un grand nombre d'applications allant des téléphones portables aux dispositifs multimédia de grand public tels que des récepteurs et décodeurs de télévision numérique. Nous consacrons cette section à la présentation des caractéristiques de ces systèmes et les défis auxquels sont confrontés leurs concepteurs. La section se conclut par une projection de ces caractéristiques telles que l'on peut les envisager dans les produits de prochaines générations afin de mieux comprendre les enjeux concernant la programmation de ces plates-formes multiprocesseurs.

1.1.1 Constituants d'un système sur puce

Les systèmes sur puce comme leur nom l'indique sont de véritables systèmes intégrant de nombreux composants à l'échelle d'une puce. Ils intègrent aujourd'hui plusieurs millions de transistors sur une surface de silicium de quelques millimètres carré [MP03].

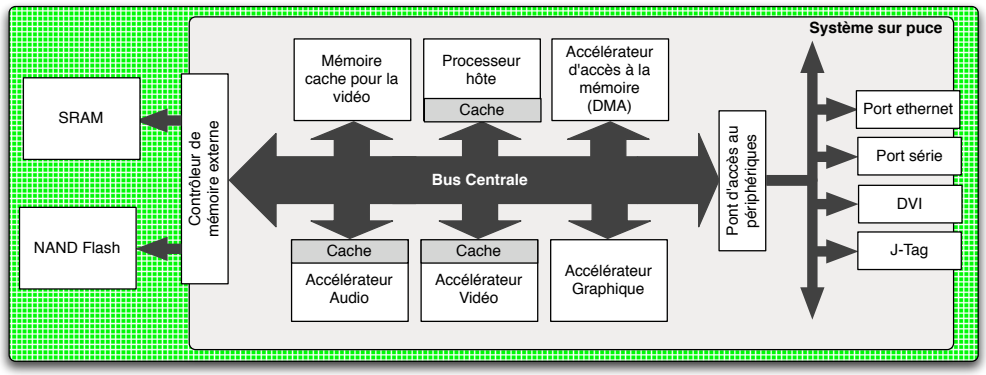


FIG. 1.1 – Un exemple d'architecture classique de système multiprocesseurs sur puce.

La figure 1.1 illustre l'architecture d'un système multiprocesseur sur puce classique, telle que l'on peut la trouver par exemple au sein d'un téléphone portable moderne dans la série "Nomadik" STi88XX de STMicroelectronics. Les composants constituant un tel circuit peuvent êtres distingués comme suit :

Ressources de calcul : Les systèmes sur puce intègrent en général plusieurs ressources de calcul, d'où leur désignation de système multiprocesseurs sur puce. Nous faisons une distinction entre les ressources de calcul programmables au sens classique du terme et celles qui ne le sont pas. Les ressources de calcul programmables sont autrement appelées coeurs de processeurs dans le sens où ils se limitent le plus souvent au strict nécessaire c'est à dire le coeur programmable auquel est adjoint un nombre très limité de périphériques (contrôleur d'interruption, timer, ...). Deux grandes classes de processeurs sont par ailleurs identifiables : les processeurs dits "hôtes" et processeurs dits "médias", suivant une architecture de type maître-esclave. Un système sur puce intègre en général un processeur hôte qui est en charge d'exécuter les programmes d'interface utilisateur et d'assurer le contrôle du reste du système. Sont couplés au processeur hôte un ou plusieurs processeurs de traitement de signal : les fameux "processeurs

médias". Ces derniers ont des architectures plus sophistiquées ou plus dédiées permettant d'obtenir des puissances de calcul plus importantes. Ils présentent le plus souvent des caractéristiques de conception qui les rangent dans la famille des processeurs de traitement du signal ou DSP (*Digital Signal Processing*). On y trouve aussi des architectures de type *Very Long Instruction Word* (VLIW) qui permettent d'exécuter plusieurs instructions en parallèle à chaque cycle afin d'accélérer des opérations de calcul arithmétique complexe. Les ressources de calcul non-programmables sont le plus souvent couplées fortement aux processeurs programmables média, constituant un ensemble de satellites de traitement en charge d'opérations de traitement critiques. Ces ressources de calcul dédiées, conçues comme des ASIC traditionnels, sont encore aujourd'hui plus que jamais nécessaires pour assurer les performances requises et les pics de performances que l'on demande à de tels circuits. Tous les signes indiquent que cette réalité sera persistante.

Ressources de mémorisation : La performance de calcul dépend en grande partie de l'efficacité des ressources de mémorisation. Bien qu'une large part de ces ressources reste encore à l'extérieur de la puce (e.g. SRAM, Flash, etc.) à cause des limites des technologies d'intégration, on retrouve de plus en plus d'unités de mémorisation à l'intérieur même des puces. Les mémoires caches standard ou les mémoires locales constituent un exemple classique. On en trouve aussi au niveau central pour accélérer l'accès à certaines données critiques, comme c'est le cas de la mémoire tampon pour la vidéo que l'on retrouve sur la figure 1.1. Dans certains cas, de petites mémoires tampons sont intégrées entre des blocs de calculs critiques afin de maîtriser les flux de données échangées. Enfin, des accélérateurs d'accès aux différentes mémoires (DMA[3]), organisés en systèmes hiérarchiques, sont employés pour rendre les transferts de données plus efficaces.

Périphériques d'entrée/sortie : Les systèmes sur puce intègrent de nombreux types de périphériques qui servent d'interface avec les composants se trouvant à l'extérieur de la puce. Parmi ces périphériques, citons des interfaces d'entrée/sortie standard (e.g. ethernet, USB, modem de communication sans fil, etc.) et les composants ou capteurs sophistiqués (e.g. capteurs de température, de mouvement, etc.).

Eléments d'interconnexion : Enfin, l'interconnexion de l'ensemble des composants sus cités est assurée par un ou plusieurs réseaux de communication. Les exemples classiques de moyens d'interconnexion employés majoritairement aujourd'hui sont les *bus* et les *cross-bar*. Alors que le premier type constitue une ressource de communication partagée entre plusieurs blocs, le deuxième est le nom générique pour des interconnexions point-à-point qui permettent d'obtenir des bandes passantes plus importantes entre des blocs critiques. Le nombre de composants intégrés dans les puces augmentant, le domaine de recherche connu sous le nom de "Réseaux sur puce" (NoC[4]) [MB02, JT03, BM06] est sorti de l'anonymat au cours des cinq dernières années. L'objet de ces recherches porte sur la mise en place de moyens de communication sur puce plus efficaces en termes de bande passante et de consommation énergétique, répondant par ailleurs plus pertinemment aux contraintes d'asynchronisme global accompagnant la forte intégration actuelle. Ces solutions sont similaires dans leurs principes aux réseaux d'ordinateurs à plus large échelle qu'elles transposent à l'échelle de la puce où les distances relatives entre blocs de traitement augmentent.

1.1.2 Problématiques des concepteurs de systèmes sur puce

Gordon Moore a prédit dans les années 1960 que la croissance de la complexité des circuits intégrés suivrait une courbe exponentielle alors que la taille des transistors diminuerait dans les mêmes proportions [Moo00]. L'évolution de la technologie des semi-conducteurs a confirmé cette prévision pendant de longues années. Or, un autre paramètre crucial, la productivité, n'a pas suivi la même croissance :

[3]DMA pour *Direct Memory Access*.
[4]NoC pour *Network-on-Chip*.

l'efficacité de la production a baissé au cours des années, comme illustré dans la figure 1.2. Ainsi, l'écart entre la capacité d'intégration des circuits et l'usage effectif de cette capacité d'intégration s'est accru. Il est lié en partie à la difficulié de gestion de la complexité grandissante de conception des MPSoC déjà évoquée : agréger des fonctionnalités ne suffit pas ; encore faut-il le faire méthodiquement sous peine d'atteindre des points de complexité impossibles à gérer. Il est aussi lié au coût prohibitif de conception, par ailleurs corrélé au point précédent. Aujourd'hui, un projet de développement de système sur puce demande un effort de près de mille hommes année, et coûte des dizaines de millions d'euros.

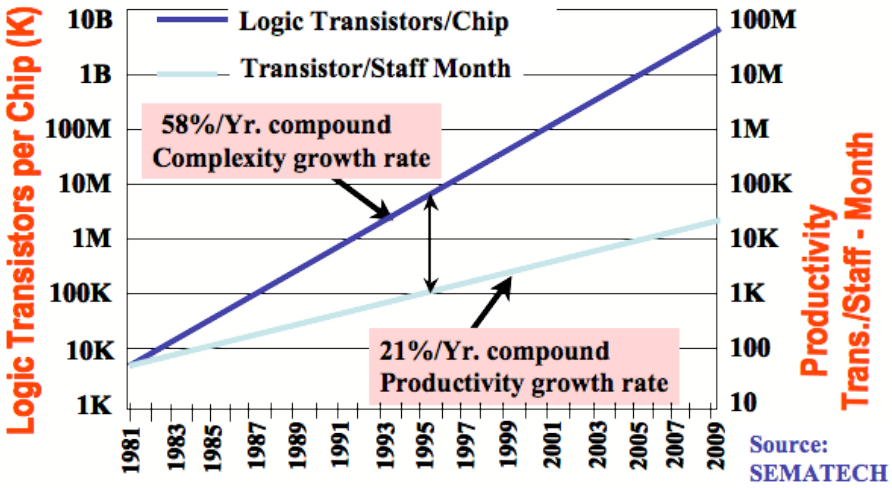


FIG. 1.2 – Accroissement de l'écart entre les technologies d'intégration des composants semi-conducteurs et la productivité exprimée en termes de nombre de transistors que les concepteurs sont capables d'intégrer [Sem].

À ces aspects technologiques et financiers se rajoutent les contraintes du marché des circuits électroniques intégrés à forte valeur ajoutée. En effet, l'évolution des normes applicatives oblige les concepteurs à renouveler sans cesse les produits proposés. Cela implique des temps d'accès au marché de plus en plus courts. Pour se rendre compte de la vitesse de cette évolution, il suffit d'observer le cas familier des téléphones portables qui sont devenus de véritables terminaux multimédia très performants en l'espace de quelques années. De plus, la démocratisation de tels produits et la compétitivité entre les producteurs de systèmes sur puce imposent à ces derniers des réductions de prix de vente. Typiquement, un produit dont le développement coûte près de 100 millions de dollars doit être vendu à moins de 10 dollars à l'unité. Dans ce contexte, le producteur doit en vendre plus de 10 millions d'unités pour rentabiliser son investissement et chercher donc de "gros volumes".

1.1.3 Feuille de route pour les architectures du futur

Un problème intéressant se pose donc aux architectes matériel et logiciel de systèmes sur puce : augmenter la productivité tout en gardant les prix de production dans des intervalles raisonnables [MP03]. La **régularité** et la **réutilisation** sont les maîtres mots de la feuille de route qui commence à se dessiner. La régularité vise à réduire la complexité de production. Il s'agit d'employer des patrons de conception réguliers à tous les niveaux de la construction des systèmes sur puce, depuis le niveau de l'intégration des transistors jusqu'au niveau architecture de système. Orthogonalement à celle-ci, la réutilisation vise à capitaliser sur les coûts de production en proposant la même puce pour plusieurs domaines d'application en modifiant la couche logicielle qui accompagne le circuit.

Cet objectif de conception favorisant des architectures plus régulières et réutilisables des systèmes

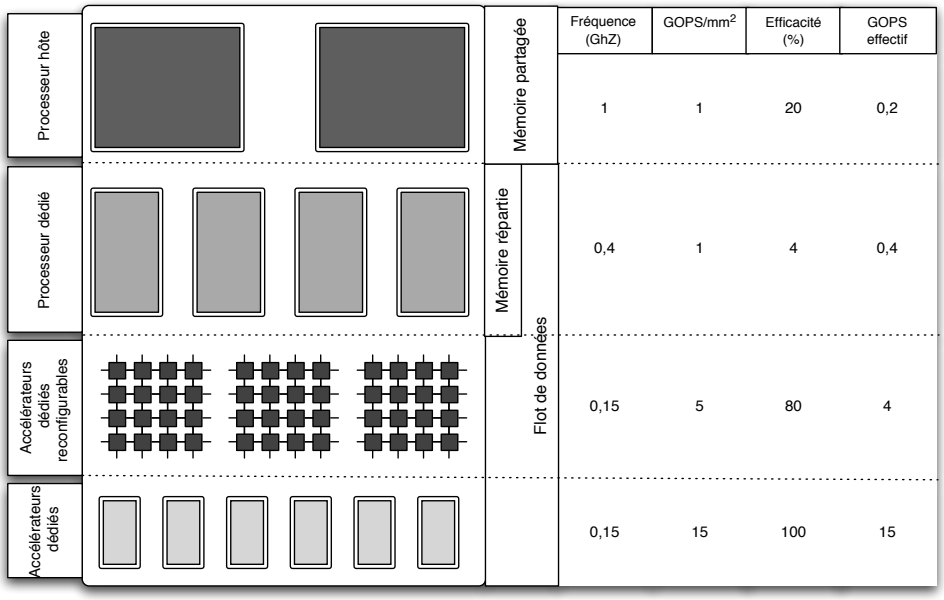


FIG. 1.3 – Un aperçu des ressources de calcul et des architectures de mémoires qui seront intégrées dans les systèmes sur puce de prochaine génération.

sur puce n'est pas forcément incompatible avec la nécessaire persistance de l'hétérogénéité que l'on retrouvera dans les puces des prochaines générations. Il s'agit essentiellement de *structurer* cette hétérogéneité : c'est cette clé, autrement formulée plus haut sous le vocable de "patron de conception" qui permettra à termes d'appréhender correctement cette complexité. La figure 1.3 présente un aperçu ordonné des technologies qui pourront être intégrées avec méthode dans un futur proche. Cela comprend l'intégration de ressources de calcul hétérogènes couplées à des architectures de mémoire appropriées, ainsi que la mise en place de moyens de communication évolués. Nous détaillons ci-dessous chacun de ces points.

Persistance de l'hétérogénéité des ressources de calcul En raison du compromis persistant programmabilité versus efficacité, les concepteurs sont contraints de maintenir l'hétérogénéité des ressources de calcul employées dans les puces. La figure 1.3 illustre donc cette hétérogénéité en distinguant quatre grands domaines de calculs : le domaine formé par les ressources de calculs consituant la partie "hôte" de la puce, celui constitué des processeurs programmables dédiés, le domaine des accélérateurs matériels lui-même subdivisé en accélérateurs reconfigurables[5] et accélérateurs dédiés de type ASIC. Associées à ces domaines sont données sur cette figure des données empiriques permettant de préciser les ordres de grandeurs considérées par domaines. Ainsi, le rapport entre les performances d'exécution démontrées par les accélérateurs matériels dédiés et les processeurs à usage général s'élève à presque deux ordres de grandeurs, pour une puissance dissipée sans commune mesure. Entre ces deux extrémités se trouvent deux solutions intermédiaires. Il s'agit des processeurs dédiés (e.g. DSP, VLIW) qui fournissent des facilités de calcul pour des opérations intensives et des circuits accélérateurs reconfigurables (e.g. technologie FPGA ou assimilée). Afin d'améliorer l'intégration de ces différentes technologies définissant autant de domaines de calcul, fournir des facilités de programmations pour répartir le calcul sur ces différents domaines tout en exploitant au mieux leurs spécificités représente un verrou technologique majeur.

[5]Certains emploient ici la notion un peu détournée d'"accélérateur matériel programmable".

Utilisation de différentes architectures de mémoire À ces domaines de calculs séparés, la tendance est de coupler des architectures mémoire appropriées afin de répondre au mieux aux besoins spécifiques de chacun de ces domaines, avec pour conséquence une efficacité globale accrue. Par exemple, alors que les logiciels de type interface graphique s'exécutant sur le domaine hôte sont plutôt écrits à l'aide de plusieurs *threads* qui ont besoin de partager une quantité d'information importante, les calculs de type multimédia effectués typiquement sur les accélérateurs dédiés sont a contrario organisés sous forme de *pipelines* au sein desquels les données sont propagées de place en place au gré de leur transformation. Dans ce cadre tendent à coexister trois grands types d'architectures mémoire (voir la figure 1.3) : (i) architecture de type mémoire partagée dans la partie hôte (ex : noeud SMP ou *Symmetrical Multi Processor*), (ii) architecture de type mémoire répartie présente au niveau des processeurs dédiés pour permettre un modèle de communication de type échange de messages, et (iii) un ensemble de mémoires tampons ou *buffers* insérées entre les différents composants de la partie média pour assurer une communication de type flot de données [Cel].

Utilisation de technologies d'interconnexion avancées Enfin, les technologies de réseau sur puce devraient remplacer les moyens d'interconnexion classiques pour assurer une meilleure bande passante entre les ressources de calcul et de mémorisation, tout en facilitant le mise en place d'architectures globalement asynchrones et localement synchrones ou GALS (Globaly Asynchronous Locally Synchronous). Ces technologies permettront par ailleurs de fournir des modèles de programmation et des garanties en termes de qualités de service, similaires à celles fournies dans le domaine de réseaux d'ordinateurs à plus grande échelle. De cette façon, les programmeurs devraient disposer d'abstractions de plus haut niveau pour la communication inter-processeurs, ce qui devrait contribuer à la simplification de la mise au point des applications réparties.

1.2 Programmation des systèmes multiprocesseurs sur puce

Nous consacrons cette section à l'analyse du flot de développement de logiciel pour les systèmes sur puce. Elle nous permettra de mettre en lumière les techniques de programmation utilisées et les besoins en termes d'ingénierie logicielle et technologies que ces travaux de thèse tentent de contribuer à combler.

1.2.1 Flot de conception de logiciel pour les systèmes sur puce

Encore aujourd'hui, l'une des différences principale concernant le flot de conception de logiciel pour les systèmes sur puce comparativement au flot classique de développement est que la plate-forme matérielle est produite en même temps que le logiciel [JM06, JW05]. Cela s'explique historiquement : les systèmes sur puce jusqu'à une date encore récente étaient de purs ASIC (Application Specific Integrated Circuits). L'apparition de processeurs programmables s'est faite douloureusement par la pression d'un marché enclin à favoriser les solutions plus flexibles en raison en particulier de la nature fortement impermanente des standards multimedia. Cependant, la logique de conception des systèmes reste encore profondément marquée par la logique ASIC : un système est "taillé" en fonction de contraintes applicatives figées (performances, services d'accélération, ...) dans une logique "pire-cas" et en adoptant une spécialisation systématique des diverses fonctions de la puce qui sont conçues et programmées indépendamment (ex : partie audio, partie video, etc.) et ce même si en pratique les même cœurs de processeurs entrent souvent en jeu. La figure 1.4 illustre un flot de conception classique quelque peu idéalisé des systèmes sur puce, aussi appelé conception conjointe matérielle/logicielle. Ce flot débute par la mise au point d'une spécification du système en partant d'une spécification fonctionnelle de la plate-forme et des contraintes à prendre en considérations (e.g. technologie de fabrication, surface, budget énergétique, puissance de calcul et bande passante à supporter, etc.). Une fois la spécification établie, les fonctions à implanter sont partitionnées pour être implantées en matériel ou en logiciel. C'est en particulier dans cette partie que l'on décide combien et quels types de processeurs seront intégrés, et quels types de

fonctions seront implantées par des accélérateurs dédiés.

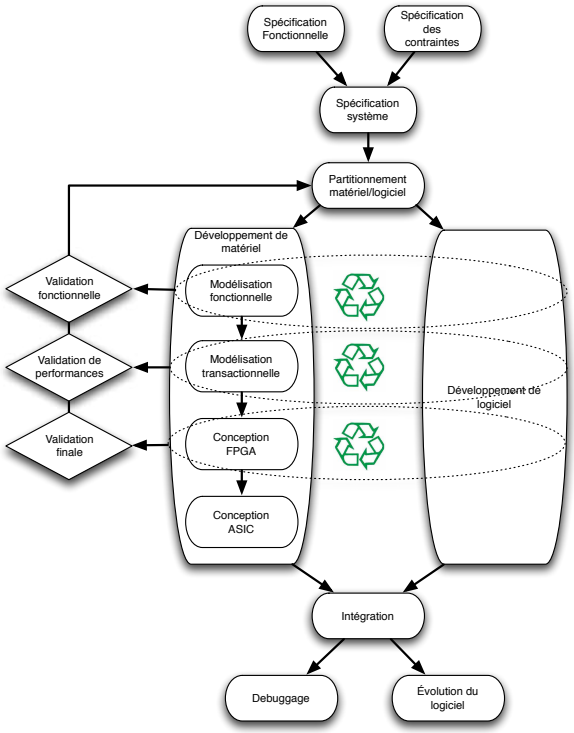


FIG. 1.4 – Flot de conception idealisé d'un système sur puce.

À partir de ce partitionnement commence un travail conjoint de conception du matériel et du logiciel. Une tendance émergente de la part des concepteurs de matériel consiste à concevoir en premier lieu un modèle fonctionnel puis transactionel sous forme de simulateur. Ce modèle sert en particulier de plate-forme d'exécution pour le développement de logiciel et peut permettre d'obtenir de premiers chiffres estimés de performance. Si celles-ci ne conviennent pas, le partitionnement matériel/logiciel peut être revu dans un processus itératif facilité par la nature virtuelle de la plateforme considérée. Après avoir validé ce stade, un prototypage sous la forme d'une implantation à l'aide de technologies reconfigurables (FPGA[6]) est souvent mis en place afin d'effectuer une ultime validation. C'est à partir de ce stade que les concepteurs de logiciel peuvent commencer à effectivement intégrer le logiciel à la plate-forme cible dans des conditions réalistes. Enfin, les circuits finaux sont fabriqués. Un travail d'intégration de logiciel commence alors afin de finaliser la production de la puce. Il faut remarquer que la tâche des programmeurs continue même après avoir expédié les puces chez le client. Il s'agit à partir de ce stade de réparer les erreurs et envisager les mise à jours du logiciel fourni.

En pratique, il convient de préciser que le flot de conception ne présente pas forcément cette rigueur pour la totalité du système sur puce, pour la simple raison que rares sont les circuits conçus à partir de zéro. Les circuits sont en effet le plus souvent des versions améliorées ou plus sophistiqués de circuits existants, constituant la génération suivante dans une ligne de produits donnée. Ainsi de grande portions de circuits préexistants sont amenées à être réutilisées, le travail d'intégration globale constituant la partie délicate de la manoeuvre, nécessitant forces adaptations et rustines.

[6]FPGA pour *Field Programmable Gate Array*.

1.2.2 Problématique de programmation

La quantité de logiciel intégré dans ces plates-formes atteint d'ores et déjà des niveaux très importants. La figure 1.5 donne un aperçu des couches logicielles qui peuvent être présentes dans une application de téléphonie mobile moderne. Nous constatons que similairement à la couche matérielle, la couche logicielle intègre elle aussi diverses technologies allant des systèmes d'exploitation dédiés aux machines virtuelles.

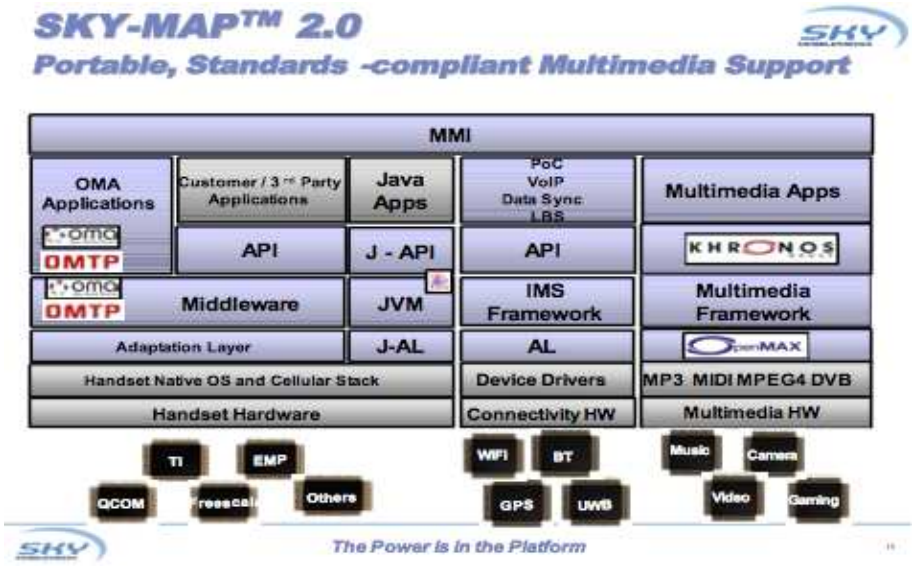


FIG. 1.5 – Exemple d'empilement de couches d'abstractions tel qu'il peut se trouver dans une application de téléphonie mobile.

1.2.2.1 Programmation du processeur hôte

La partie hôte d'un système sur puce est en général composée d'un processeur à usage général. Ce type de processeurs fournissent des services similaires aux processeurs ordinaires que l'on retrouve dans les stations de travail. Cependant, ils tournent à des fréquences plus basses, ce qui résulte en des performances de calcul considérablement plus réduites.

Les logiciels exécutés sur la partie hôte, outre le contrôle et l'orchestration du système global, comportent pour l'essentiel les parties interactives telles les interfaces graphiques utilisateur, les jeux et les applications utilitaires. La plupart de ces logiciels sont écrits par des tierces parties. Par exemple, dans le cas d'un système sur puce dédié à la téléphonie mobile, la société qui intègre le téléphone (e.g. Nokia, Ericsson, etc.) et les fournisseurs d'accès aux services de téléphonie (e.g. Orange, Vodafone, etc.) vont déployer leurs logiciels propres dans la partie hôte. Pour cette raison, la problématique principale de ce domaine est la programmabilité et non la performance. Afin de fournir des services de programmation évolués, la partie hôte accueille des systèmes d'exploitation d'architecture et de fonctionnalité classiques comme Symbian [Sym], Linux [Lin] ou Windows CE [Win]. Au-dessus du système d'exploitation, sont supportées de nombreuses librairies de programmation ou des technologies de virtualisation comme les machines virtuelles Java ou .Net comme illustré sur la figure précédente.

Le calcul intensif nécessaire pour les applications multimédia ou des fonctions de transmission sont délégués à la partie média à l'aide des interfaces de programmation d'application (API). L'implantation de ces interfaces est assurée en partie par des micrologiciels s'exécutant en dessous du système d'exploitation du processeur hôte. De cette façon, les programmeurs de tierces-parties sont abstraits des

complexités architecturales de la partie média. Il existe à ce jour des tentatives considérables de standardisation des interfaces de programmation pour normaliser les primitives d'accès aux services multimédias. Par exemple le standard OpenMax[Ope06] rédigé par le groupe Khronos définit des primitives pour accéder au services de type codage/décodage audio/vidéo ou calculs graphiques. Bien que ce type de standards permettent d'améliorer la réutilisation des programmes utilisateurs, ils ont comme inconvénient de limiter l'exploitation des ressources spécifiques des plates-formes matérielles sous-jacentes à un ensemble figé de primitives.

1.2.2.2 Programmation de la partie média

Le logiciel qui est exécuté dans la partie média constitue la plus grande partie du micrologiciel fourni avec le système sur puce. Ce type de micrologiciel implante en générale l'intégralité des services de type média et transmission de signaux disponibles. Tous les choix de conception concernant cette partie privilégient les performances. Pour cette raison, on retrouve sur les processeurs médias des systèmes d'exploitation maison, i.e. le plus souvent conçus par les producteurs de systèmes sur puce.

Un défi de la programmation de la partie média est la mise au point des applications parallèles de manière à exploiter au mieux les possesseurs et les accélérateurs dédiés [KDH+05]. Or, très peu de méthodologies de développement sont définies à l'heure actuelle pour maîtriser le processus de mise en œuvre d'applications parallèles. La pratique actuelle de parallélisation est basée sur la programmation à base de *threads* et sur l'utilisation des interfaces d'échange de message à la MPI[7] [OIS+06, GBB+04]. Cela résulte en la mise en place de programmes difficilement réutilisables et évaluables. Enfin, une très grande partie du support pour la communication inter-processeurs est implantée de manière ad-hoc. Cela pose d'importants problèmes de portage lors qu'il s'agit d'adapter les programmes à une architecture de plate-forme différente.

Les programmes sont en grande partie écrits en C en se basant sur des librairies parfois implantées en assembleur pour améliorer les performances. Le volume de code global est considérable : il s'agit généralement de quelques millions de lignes de code écrites par des experts en algorithmique et non en ingénierie logicielle. A ces aspects se rajoute la diversité des architectures matérielles et des périphériques. Par conséquent, l'intégration de différents modules développés par des équipes réparties dans le monde constituent une des étapes les plus problématique du processus de développement de micrologiciels. Bien qu'il existe également des tentatives de standardisation des interfaces d'accès aux périphériques et aux accélérateurs matériels [Ope05], il nous semble compliqué que les concepteurs de systèmes sur puce puissent converger sur des spécifications communes [BCP+05, CWJ05]. En effet, l'élément différenciateur entre différents systèmes sur puce consiste précisément en l'ensemble des fonctions spécifiques supportées pour satisfaire un domaine applicatif bien précis.

1.2.3 Analyse critique de la perception des pratiques actuelles

À la lumière de la problématique de programmation des systèmes sur puce exposée préalablement, nous nous permettons quelques remarques sur les limites principales des pratiques actuelles en les organisant en des rubriques exprimants les besoins que nous avons pu identifier.

Abstraction efficace de la plateforme matérielle La méthodologie de conception décrite plus haut, issue d'une tradition ASIC, reste très orientée par les considérations matérielles. Le circuit, bien que programmable pour de larges part, n'est pas conçu pour être programmable en tant qu'entité globale. La seule partie réellement programmable et accessible à des tierces parties reste le plus souvent exclusivement la partie hôte de contrôle du système, partie émergée de l'iceberg. Les parties programmables medias sont conçues le plus souvent indépendemment les unes des autres, et sont programmées par les

[7] MPI pour *Message Passing Iterface*.

concepteurs du logiciel propriétaire de la puce. Ces programmeurs sont des spécialistes de la plate-forme, en butte directe à l'extrême complexité de celle-ci. Le code est programmé de façon extrêmement dédiée, faisant appel presque directement aux accélérateurs matériels fortement couplés aux processeurs média considérés. Ces programmeurs spécialisés utilisent des chaînes d'outils de conceptions pareillement dédiées, nécessitant une très forte expertise pour être exploitées au mieux de leur capacité.

Il apparaît donc comme un défi intéressant que d'étudier et développer des concepts et techniques permettant d'une part d'abstraire la complexité trop importante et croissante de la plateforme programmable MPSoC, et d'autre part de considérer la programmation de cette plateforme comme un tout cohérent en s'appuyant sur ces nouvelles capacités d'abstraction. À termes, ces techniques devraient s'appuyer sur des évolutions des architectures des MPSoC facilitatrices. Le principal défi réside à vrai dire dans la recherche de la préservation des performances en dépit de l'abstraction, ce qui apparaît comme un défi relevable pour peu que l'on adopte des principes de conceptions adéquats. De ce point de vue, la philosophie de conception dite "exo" apparaît pertinente. Elle est une généralisation de la mouvance "exo-kernel" au sens de ne rajouter que le strict nécessaire, par opposition à des approches génériques souvent coûteuse (taille, performance). "exo" n'est pas équivalent à l'"ad-hoc"pointé du doigt dans la mesure où l'approche est systématisée à l'aide d'un outillage dédié, notamment dans une approche à base de composants logiciels s'appuyant sur un outillage d'assemblage exploitant la description expicite de l'architecture logicielle ciblée.

Méthodologies d'ingénierie logicielle Il ressort des constatations plus haut que le logiciel reste traité en quelque sorte comme un "mal nécessaire" même si ce qualificatif est quelque peu caricatural. En particulier, si le développement de la partie matérielle de la puce dispose d'une méthodologie bien établie, ceci n'est pas vraiment le cas, ou du moins pas encore, de la partie logicielle. Or, les systèmes sur puce sont aujourd'hui de véritables systèmes répartis ayant une complexité comparable à celle des réseaux d'ordinateurs. Ils intègrent une quantité de logiciel considérable, évaluée à plusieurs millions de lignes de code développées par quelques centaines de programmeurs organisés en équipes souvent réparties dans le monde. C'est ainsi que les rapports de coûts de conception entre matériel et logiciel se sont récemment inversés. Le manque d'un processus de développement et d'intégration permettant de faire face à cette complexité constitue la raison de la plupart des retards enregistrés au niveau de la production des puces. Cette absence de méthodologie de développement se traduit en des problèmes de conception, d'intégration, de maintenance et d'évolution de logiciel et réduit la productivité. Un souffle nouveau est indispensable.

Ouverture à la programmation par des tierces parties de la partie média Actuellement, la partie média des systèmes sur puce est uniquement accessible au travers d'interfaces de programmation standards ou définies par les producteurs. Cela limite d'une part l'exploitation des ressources disponibles à des fins initialement non prévues. D'autre part, cela augmente le nombre de couches logicielles ce qui a comme effet de réduire les performances. Enfin, les "standards" étant en pratique mouvants, une perpétuelle danse d'adaptation s'offre aux concepteurs de microgiciel. Dans ce cadre, une des perspectives intéressante pour les concepteurs de systèmes sur puce est d'ouvrir la partie média à la programmation par les tierces parties, permettant de rentrer dans une logique de "standard d'usage" plutôt qu'une logique de standard formel exclusive. Or, la complexité intrinsèque des architectures de système sur puce constitue un obstacle devant une telle démarche. Pour adresser ce problème, les concepteurs ont besoin de disposer de modèles de programmation ouverts facilitant l'accès des programmes utilisateurs aux composants de la partie média tout en permettant la définition de niveaux d'abstraction arbitraires.

Meilleure exploitation des ressources matérielles distribuées Les plateformes MPSoC restent conçues pour une classe d'applications donnée, suivant une logique "pire cas". Cette dernière notion est intéressante à développer. Elle signifie que le circuit est conçu pour pouvoir résister aux conditions

d'exécution les pires qui puisse advenir compte tenu des services applicatifs qu'il doit fournir. En d'autres termes, le circuit est globalement et largemement "sur-taillé" pour un fonctionnement moyen. Comme précisé précédemment, ces circuits sont par ailleurs conçus par parties spécialisées indépendantes (ex : partie media audio, partie média video, partie média graphique, etc ...), ces sous-domaines de calculs n'étant pas ou peu interopérables. La combinaison de ces deux facteurs résulte en une sous-exploitation des ressources de calcul dans le cas ou les sous domaines applicatifs ne sont pas en fonction. Des techniques de parallélisation des logiciel déployés sur la partie média, permettant de simplifier la programmation, la décomposition et le déploiement des applications parallèles sur l'architecture répartie, pourraient ainsi s'avérer extrèmement utiles. L'objectif serait d'une part d'exploiter de façon plus efficace les ressources de calculs très importantes disponibles et sous exploitées en cas de fonctionnement en dehors des cas pour lesquels elles ont été conçues. L'objectif, ou plutôt le défi, serait d'autre part de gérer plus efficacement l'hétérogénéité des noeuds de calcul en présence. Aujourd'hui, les moyens disponibles, si employés, se basent sur des modèles de programmation de très bas niveau, usant de moyens matériels ad-hoc pour les communications inter processeurs. De premières solutions ont le mérite d'exister pour pallier ce manque, telle la solution Multicom interne à STMicroelectronics, ou relevant de la recherche telle la solution MultiFlex/DSOC [Pau04] transposant sur puce des techniques provenant du domaines de la programmation par objects distribués.

1.3 Motivations et objectifs de notre proposition

1.3.1 Défis

Ce travail a pour objectif d'améliorer le processus de développement de logiciel pour la programmation des systèmes multi-processeurs de manière à adresser les limitations énumérées ci-dessus. Dans ce cadre, nous visons la définition d'une méthodologie de développement qui accompagne les programmeurs tout au long du processus de développement, de la conception au déploiement, en leur offrant diverses facilités comme la vérification, la génération de code, et le déploiement.

Nous basons nos travaux sur l'approche de programmation à base de composants. Cette approche d'ingénierie logicielle, largement acceptée par la communauté des systèmes répartis, propose essentiellement de structurer les programmes à l'aide des composants qui réifient des unités d'encapsulation de comportement et de données. Suivant cette approche, les composants interagissent via des connexions explicites au travers d'interfaces bien définies. Ces considérations de base permettent de découpler la manière dont les composants d'un logiciel sont implantés de la manière dont ils sont interconnectés et composés. L'identification explicite des composants d'un système et de leurs protocoles d'interaction permet d'adopter un raisonnement architectural et d'ainsi maîtriser la structure d'intégration des logiciels. De plus, ce raisonnement crucial pour l'assemblage, la configuration, la maintenance et l'évolution des systèmes complexes peut être supporté par des outils implantant des services d'aide à la conception à partir de descriptions d'architecture logicielle écrites dans des langages dédiés de type ADL[8].

La diversité des préoccupations et des contraintes associées au développement de logiciel pour les systèmes sur puce est incompatible avec la définition d'un modèle de programmation universelle. En effet, il serait très difficile de définir un modèle qui soit entièrement adapté à la fois au développement de système d'exploitation, d'application multimédias temps-réel et d'application utilisateurs dans un langage de type Java. De plus, nul ne peut prévoir les nouvelles contraintes à prendre en compte dans le futur. Pour ces raisons, les concepteurs sur puce doivent pouvoir bénéficier d'un modèle de composants et d'un outillage associé qui soit ouvert, c'est à dire extensible, afin de pouvoir intégrer de nouvelles propriétés émergenates. Une telle infrastructure doit être compatible avec des langages de description hétérogènes comprenant des éléments structurels (e.g. description d'architecture, d'interface, etc.) et

[8] ADL pour *Architecture Description Language*.

des éléments comportementaux (e.g. description de protocoles d'interaction, de synchronisation, etc.). De plus, compte tenu de la diversité des cultures logicielles et des problématiques programmatiques à résoudre, un tel environnement il doit permettre l'intégration de composants implantés dans des langages de programmation différents. Enfin, cette infrastructure doit permettre de tisser à la périphérie des composants logiciels et de la fonctionalité qu'ils encapslent divers aspects non-fonctionnels (contrôle, administration, réflexivité) pour permettre leur réutilisation dans des contextes d'exécution variés.

1.3.2 Propositions

Nous proposons de relever ces défis en fournissant une infrastructure de programmation à base de composants. Ci-dessous sont dressés les trois principaux éléments constituant notre proposition.

1. La base de notre proposition repose sur l'utilisation d'un modèle de composants ouvert qui peut être adapté aux besoins spécifiques issus de cadres applicatifs différents. Par le biais de ce modèle de composants extensible, nous visons la définition d'un modèle de programmation de base employé pour la programmation de toutes les couches logicielles considérées dans le contexte des systèmes sur puce. Nous croyons que l'adoption d'un tel modèle de base utilisé par différentes équipes de développements peut grandement contribuer à la simplification du processus d'intégration et à l'amélioration de taux de réutilisation des composants logiciels simplement par l'usage d'un même formalisme partagé.

2. Comme nous avons souligné précédemment, la programmation à base de composants va de pair avec des outils d'aide à la conception. Compte tenu du caractère ouvert du modèle de composants que nous proposons d'adopter, son outillage associé peut prendre des formes différentes, et peut être sujet à des évolutions continues. Afin de permettre l'extension systématique d'un tel outillage, nous fournissons un canevas logiciel pour la construction d'outils d'aide à la programmation à base de composant. Ce canevas définit essentiellement des patrons de programmation pour implanter différents types de services que peut implanter une chaîne de traitement basés sur des descriptions ADL. De plus, ce canevas définit des guides de programmation et fournit des composants utilitaires pour prendre en compte de nouveaux langages d'entrée et de sortie ainsi que pour intégrer de nouvelles fonctionnalités. Enfin, il définit une architecture de *plugin* pour permettre à des tierces-parties d'étendre facilement une chaîne existante.

3. Enfin nous décrivons l'architecture d'une personnalité de chaîne de traitement ADL que nous avons mis en place pour fournir des services de génération de code dans l'objectif d'automatiser l'assemblage d'un système à base de composants. Nous démontrons au travers de cette personnalité la capacité de notre canevas à supporter des langages de descriptions de natures différentes, à implanter des fonctions d'analyses variées et à intégrer les composants écrits dans des langages de programmation différents.

Dans l'objectif de démontrer les apports des propositions décrites ci-dessus au processus de développement de logiciel à destination de systèmes sur puce, nous décrivons dans ce manuscrit deux des applications que nous avons mises au point dans le cadre de nos travaux. La première application démontre la compatibilité du modèle de composants proposé avec la construction de systèmes d'exploitation spécialisés et discute des apports d'une telle approche. La deuxième application est dédiée à la programmation des applications multimédias. Nous démontrons au travers de cette application la façon dont le modèle de composants de base peut être étendu pour être adapté à des modèles d'exécution variés, et la manière dont la chaîne de traitement ADL peut être améliorée pour prendre en compte une telle extension.

1.4 Organisation du document

Ce document est organisé en trois parties. La première partie présente un état de l'art des technologies de programmation à base de composants et des outils de configuration de système basés sur des langages de descriptions d'architecture. Cette partie se conclue par la présentation du modèle de composants FRACTAL sur lequel repose notre proposition. La deuxième partie présente notre contribution centrale, c'est-à-dire le canevas logiciel pour la construction de chaînes de traitement venant en soutien à la programmation à base de composants. Enfin, la troisième partie présente deux applications qui servent de preuve de concept en démontrant les bénéfices de l'approche et du canevas logiciel proposés.

Partie I : Etat de l'art

Le Chapitre 2 « Programmation système à base de composants » définit ce que l'on entend par composant logiciel et présente différentes approches à la programmation à base de composant. Sur la base des caractéristiques des différents modèles présentés, ce chapitre dresse un panorama synthétique des préoccupations et des propriétés des modèles de composants pour permettre d'en identifier un propre à satisfaire les besoins du processus de développement de logiciel pour système sur puce.

Le Chapitre 3 « Configuration de systèmes à l'aide de langages de description d'architecture » présente différents langages de description d'architecture ainsi que les chaînes d'outils associées. Après avoir analysé les caractéristiques d'un certain nombre de langages de description d'architecture conçus pour des cadres applicatifs variés, ce chapitre identifie les diverses préoccupations considérées et ainsi contribue à la définition des propriétés que doit supporter un canevas logiciel qui permettrait de construire divers outils d'aide à la programmation à base de composants.

Le Chapitre 4 « Le modèle de composants FRACTAL et les outils associés » présente le modèle de composants qui constitue la base de notre proposition. Ce chapitre décrit les caractéristiques du modèle FRACTAL ainsi que celles de ses implantations principales. Enfin, ce chapitre présente l'outil de déploiement FRACTAL ADL qui constitue la source d'inspiration majeure du canevas logiciel proposé.

Partie II : Une chaîne d'outils extensible et multi-cibles pour le traitement de descriptions d'architectures

Le Chapitre 5 « Présentation générale » synthétise les limitations des infrastructures existantes et précise les motivations et les objectifs du canevas logiciel proposé.

Le Chapitre 6 « Un canevas logiciel extensible pour le traitement d'ADL hétérogènes » décrit le cœur de notre proposition. Sont présentés dans ce chapitre les patrons de programmation et les composants utilitaires faisant partie du canevas logiciel proposé.

Le Chapitre 7 « Construction d'un outils de traitement d'ADL pour la génération de code » présente l'architecture d'une personnalité mise au point pour servir d'aide à l'assemblage de systèmes logiciels écrits en FRACTAL. Ainsi, ce chapitre illustre comment le canevas proposé a pu être utilisé avec succès pour mettre en place une chaîne d'outils qui effectue des opérations de vérification, qui génère le code *glue* pour assembler les composants et qui compile l'ensemble des fichiers générés en intégrant ceux écrits par les programmeurs afin de créer une forme exécutable à partir d'une description d'architecture donnée.

Partie III : Applications

Le Chapitre 8 « Construction de noyau de systèmes d'exploitation » présente le canevas THINK, une incarnation du modèle FRACTAL en C originellement conçu pour la construction de systèmes d'exploitations spécialisés. Ce chapitre présente en particulier l'architecture d'un micro-noyau L4 que

nous avons construit en utilisant le canevas THINK et discute des bénéfices et des inconvénients de cette approche par rapport à une implantation industrielle monolithique.

Le Chapitre 9 « Construction d'applications de streaming réparties » présente le positionnement du canevas THINK dans le cadre de la programmation d'applications parallèles de type multimédia. Le chapitre débute par la présentation d'une expérience de *composantisation* d'un décodeur vidéo monolithique et discute des caractéristiques du résultat obtenu. A la lumière des limitations identifiées par le biais de cette première expérimentation, le chapitre propose une extension du modèle de composants FRACTAL et décrit comment cette extension a pu être facilement mise en oeuvre au sein de la chaîne d'outils de traitement d'ADL proposée. Enfin, une discussion sur les apports de cette extension est proposée au travers d'un autre décodeur vidéo implanté en utilisant le modèle proposé.

Première partie

Etat de l'Art

Chapitre 2

Programmation système à base de composants

Sommaire

2.1 Composants logiciels

Bien qu'assez jeune par rapport aux autres domaines de l'ingénierie, l'informatique a connu une évolution rapide où les méthodologies de programmation se sont développées très vite, en grande partie grâce aux recherches sur les langages de programmation. Nous essayons, dans cette section, de survoler brièvement cette histoire proche pour comprendre comment on a évolué jusqu'aux composants logiciels tels qu'on les comprend aujourd'hui et, par la suite, nous présentons une définition abstraite des composants afin de pouvoir situer les modèles et les environnements que l'on retrouve dans les propositions industrielles et académiques.

2.1.1 Un peu d'histoire

Préhistoire Il fut un temps où les machines disposaient de ressources de calcul et de mémorisation très restreintes. À cette époque ont été inventés les sous-programmes qui permettaient d'exécuter un même segment de code à plusieurs reprises avec des paramètres différents à l'aide des instructions de branchement ; l'objectif était alors de diminuer l'espace mémoire physique nécessaire pour stocker le code d'un programme. On peut considérer que la réutilisation de logiciel a été inventée à cette occasion, mais pour une raison tout à fait différente de celle qui nous préoccupe actuellement ; la réutilisation pour aider l'être humain s'imposera plus tard lorsque les machines deviendront assez efficaces pour que les programmes qu'elles seront capables d'exécuter se révèlent trop gros pour la maîtrise des programmeurs.

Evolution Très vite, les programmeurs ont compris que les sous-programmes pouvaient êtres réutilisés dans plusieurs programmes afin de faciliter le processus de développement. L'utilité de cette approche étant évidente, ces mêmes programmeurs ont commencé à essayer d'écrire des sous-programmes génériques de telle sorte qu'ils puissent être rassemblés dans des librairies et réutilisés de manière systématique. En 1968, Dijkstra a souligné l'importance de la structuration du code pour l'écriture de programmes corrects, et a introduit la notion de séparation des préoccupations en démontrant qu'un programme pouvait être composé de plusieurs segments de code implantés de manière indépendante [Dij68]. À la même date, McIlroy présentait, dans une réunion de l'OTAN, les composants logiciels comme étant les briques de base pour la production de masse dans le cadre de l'industrialisation et prononçait déjà l'idée de l'assemblage à partir de boîtes noire bien spécifiées sans citer de solution technique [McI68]. Dans les années 1970, [Par72] a présenté le concept de programmation face aux interfaces pour concevoir des logiciels avec des modules interchangeables en ignorant leurs détails d'implémentation, et [DK75] a introduit le concept de "programmation à gros grain" (*programming in the large*) qui consiste en la programmation des composants logiciels fournissant des abstractions bien définies, par des groupes de travail différents.

On peut considérer que la première tentative qui semble être l'ancêtre des composants logiciels tels qu'on les définit aujourd'hui correspond à la mise en œuvre d'*ObjectiveC* par Brad Cox [Cox86]. Les concepts fondamentaux de la programmation orientée objets comme l'encapsulation de comportements et de données, les notions d'instance, d'héritage et de polymorphisme étaient alors définis. La programmation orientée objet a apporté une nouvelle vision à la programmation et est devenue rapidement très populaire grâce au gain considérable en termes d'organisation, de réutilisation et de maintenabilité qu'elle apportait. Bien que la vision objet ait été un grand pas en avant, elle n'a jamais réussi à apporter une solution bien établie à la question de *comment assembler le tout à partir des briques de base* ; question que l'on se pose aussi dans d'autres domaines de l'ingénierie tels que l'ingénierie civile et électronique où la réponse est fondée sur la notion d'*architecture*.

Composants Les composants logiciels sont apparus au cours de la dernière décennie pour aborder cette question en mettant l'architecture au cœur du processus de conception de logiciel. Alors qu'il n'existe pas à l'heure actuelle une définition du terme *composant logiciel* acceptée par l'ensemble de la communauté, nous pouvons considérer qu'il s'agit d'une tentative d'aller au-delà des *objets* [Szy98] en fournissant les modèles et les outils nécessaires pour capturer, maîtriser et manipuler l'architecture d'un système logiciel.

2.1.2 Caractérisation des composants logiciels

Il existe de multiples tentatives de définition des composants logiciels dans la littérature. Cette multiplicité vient du fait qu'il existe des points de vue variés allant d'une vision de morceaux de codes encapsulés via des interfaces avec un format binaire standard (Microsoft COM), jusqu'à une vision d'entités logicielles coopérantes indépendamment déployables au sein d'un canevas d'exécution commun (Corba Component Model - OMG). Néanmoins, la définition ci-dessous faite par Szyperski lors du *Workshop on Component Oriented Programming* en 1998 [BW98] décrit une vue abstraite des composant logiciels en soulignant les propriétés de base qui font d'eux les briques de bases d'un système logiciel conçu par assemblage.

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition. (Clemens Szyperski - 1996)*

Tout en restant conforme à cette définition, nous voyons les composants comme des entités logicielles présents à la conception pour modéliser des modules arbitrairement complexes avec des services propo-

sés et requis pour le bon fonctionnement, mais aussi présents à l'exécution en tant qu'unités d'encapsulation et de configuration. Nous précisons par la suite, notre conception des composants en essayant de décrire un modèle abstrait.

2.1.2.1 Type

Le *type* d'un composant est une définition abstraite d'une entité logicielle. Il présente la vue extérieure d'un composant et donne suffisamment d'informations pour présenter ses fonctions, ses dépendances et ses propriétés configurables. Plus précisément, le type d'un composant est constitué de l'union de ses *interfaces serveur*, de ses *interfaces client* et de ses *attributs*.

Les interfaces serveur donnent accès aux fonctions implantées par le composant en réagissant aux invocations reçues. Dans ce sens, elles constituent l'équivalent de la notion d'interface dans la programmation orientée objets. Un composant peut implanter plusieurs interfaces. Pour cette raison, elles sont en général définies par une paire (*nom*, *type*). Comme dans les objets, le type d'une interface est défini par l'ensemble des signatures des opérations qu'il contient.

Les interfaces client d'un composant lui donnent accès à son environnement pour invoquer des opérations sur d'autres composants. La caractérisation du type d'une interface client est identique à celle d'une interface serveur. En termes d'architecture, l'introduction des interfaces client présente une avancée considérable par rapport aux objets en explicitant les références extérieures d'un module logiciel qui étaient cachées dans le code des objets. Par ce biais, les composants forcent la programmation face aux interfaces, ce qui permet (i) de connaître les dépendances d'un composant à son environnement en regardant son type et (ii) de modifier le fournisseur du service requis sans modifier le code d'implantation.

Les attributs constituent les paramètres d'un composant accessibles pour permettre à son environnement de configurer son comportement. Ils permettent en général la réutilisation d'un même type de composant dans des contextes différents en permettant d'ajuster les propriétés configurables au début ou au cours du cycle de vie d'une instance de composant. Les attributs sont en général définis par des doublets *(nom, type)* et sont accessibles via des opérations de mise à jour (*set*) et de lecture (*get*) fournies par le composant.

2.1.2.2 Implantation

L'implantation d'un composant correspond à l'association d'un comportement à un type de composant. Si l'on fait une analogie avec la programmation orientée objets, l'implantation d'un composant correspond à une classe, dans un langage à base de classes ou à un prototype dans un langage à base de prototypes. Plus précisément, il s'agit de l'implantation de l'ensemble des interfaces serveur d'un composant en utilisant ses interfaces client et en prenant en compte ses attributs.

À ce point, nous trouvons utile de préciser que les interfaces serveur d'un composant sont souvent réparties en deux classes, du point de vue du contexte d'utilisation et éventuellement de moyens d'implantation. Il s'agit des interfaces fonctionnelles et des interfaces non-fonctionnelles. Les interfaces fonctionnelles d'un composant représentent son métier en étant abstraites de tout aspect relatif à l'environnement d'exécution du composant. L'implantation des interfaces fonctionnelles est souvent écrite dans des langages de programmation ou parfois obtenue par l'assemblage d'autres composants dans des modèles hiérarchiques. Les interfaces non-fonctionnelles sont en charge de fournir l'implantation des autres aspects liés au contrôle du composant dans son environnement d'exécution. Leurs implantations sont souvent générées en fonction des propriétés architecturales et des propriétés environnementales du composant. Parmi les aspects de contrôle, nous pouvons citer la manipulation des attributs, l'implantation des propriétés réflexives des composants, l'établissement des services requis à travers des liaisons, la gestion du cycle de vie des composants, etc.

2.1.2.3 Paquetage

Un paquetage de composants est une unité diffusable et déployable [Mar02]. Un paquetage contient suffisamment d'informations pour permettre à une usine à composants d'instancier le ou les composants qu'il inclut. Ces informations sont en général constituées au moins d'un type de composant et de son implantation sous une forme compilée en binaire natif pour une plate-forme matérielle donnée, ou interprétable et/ou compilable à la volée dans le cas des binaires Java ou .Net.

Le type inclus dans un paquetage permet de renseigner l'usine de déploiement sur la spécification architecturale du composant en termes des services qu'il fournit (métiers, contrôle, etc.), des services qu'il requiert (dépendances fonctionnelles, persistance, gestion de cycle de vie, etc.) et de ses propriétés configurables. Dans la plupart des modèles à composants, l'implantation du composant est présentée sous une forme monolithique contenant du code et des données. En revanche, lorsqu'il s'agit de modèles hiérarchiques et réflexifs, l'implantation est souvent présentée de manière récursive, c'est-à-dire que l'implantation de chaque composant contient potentiellement un assemblage de sous-composants. Dans ce cas, le paquetage contient suffisamment d'informations architecturales sur l'intérieur du composant (les sous-composants à instancier, leurs inter-connexions, etc.) pour permettre à l'usine de déployer les composants de manière récursive.

2.1.2.4 Instance

Une instance de composant est une entité vivante à l'exécution. Si, dans la programmation orientée objets, une implantation de composant équivaut à une *classe*, une instance de composant équivaut à un *objet*. Une instance de composant est identifiée par une référence unique dans son contexte d'exécution, et possède un type et une implantation. Au cours de son cycle de vie, une instance maintient un état interne évolutif, potentiellement observable via ses interfaces non-fonctionnelles, répond aux invocations de ses interfaces fournies et émet des opérations sur d'autres composants via ses interfaces requises. Il est important de remarquer qu'une instance de composant n'est pas forcément une entité active, au même titre que les objets. Dans des modèles à composants réflexifs, une instance de composant peut fournir des interfaces d'introspection pour rendre son architecture interne observable et manipulable depuis l'extérieur.

2.2 Modèles de composants standards et industriels

2.2.1 Beans / Enterprise Java Beans

Les propositions de Sun Microsystems dans le domaine des composants logiciels commencent avec les Java Beans [Ham97] qui sont destinés à la mise en place des interfaces graphiques. Les composants *Java Beans* sont en effet des objets Java classiques contenant certaines méta-informations sur les événements auxquels ils répondent, et sont destinés à être assemblés par des outils graphiques. Enterprise Java Beans (EJB) [DK06] est un deuxième modèle proposé par Sun qui est plus riche et qui est destiné à la mise en place des serveurs d'application. Notre intérêt principal étant focalisé sur la programmation des systèmes complexes, nous allons, dans la suite, nous focaliser sur les composants EJB.

2.2.1.1 Modèle

Structure d'un composant EJB Un composant EJB est essentiellement un objet Java qui fournit un certain nombre d'opérations de contrôle bien précises en plus de ses opérations métiers. Pour cette raison, un composant EJB n'a qu'une seule interface serveur, et n'a aucune interface client faisant partie de son type. Il existe trois types de composant EJB : les composants *session*, *message* et *entité*. Les composants *session* sont destinés à être instanciés pour chaque session d'interaction entre un client et le serveur. Leur durée de vie est équivalente à la durée de validité de l'interaction. Les composants *session* sont dédiés à

l'implantation de la logique (du comportement) de métier, et sont distingués par le fait qu'ils ont un état persistant ou non. Les composants *message* ont essentiellement le même rôle que les composants *session* mais sont compatibles avec le mode de communication asynchrone. Enfin, les composants *entité* sont de durée de vie indéfinie et sont dédiés à la réification des données persistantes du serveur.

Conteneurs d'EJB Les composants EJB sont assemblés et exécutés au sein de conteneurs. Les conteneurs implantent la création et la destruction des composants EJB et sont en charge de leur fournir un environnement d'exécution comprenant des services non-fonctionnels de type persistance, gestion des transactions, etc. Un conteneur s'exécute au-dessus d'une machine virtuelle Java sur un serveur physique, et peut gérer plusieurs types d'EJB. L'instanciation de composants se fait par l'interface *Home* fournie par les conteneurs. À partir de la spécification v2, les utilisateurs peuvent étendre cette interface pour supporter des fonctionnalité additionnelles. Les conteneurs sont aussi responsables de gérer les accès aux composants qu'ils accueillent. Alors que les composants se trouvant dans le même conteneur peuvent interagir directement, l'exportation des interfaces par le conteneur est requise pour donner accès aux fonctions métiers depuis l'extérieur. Comme illustrés dans la figure 2.1, les conteneurs peuvent intercepter les interfaces qu'ils exportent pour effectuer des pré- et post-traîtements.

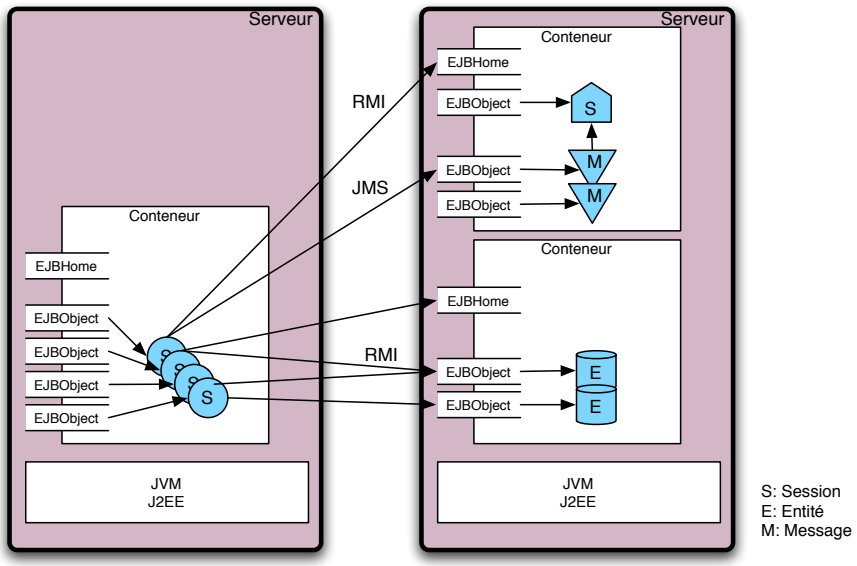


FIG. 2.1 – Architecture d'un système avec des EJB.

2.2.1.2 Implantation

L'approche EJB propose de séparer l'implantation des fonctions métiers des aspects non-fonctionels. Les composants encapsulent des codes métiers et bénéficient des environnements d'exécution enrichis fournis par les conteneurs. Pourtant, cette approche ne propose aucun support de génération de code ; bien qu'ils bénéficient d'un grand nombre de services génériques, les conteneurs sont implantés par les programmeurs en Java. Parmi les services génériques, nous pouvons citer le service de nommage et de répertoire supporté par JNDI et les communications synchrone et asynchrone supportées respectivement par Java-RMI et JMS.

2.2.1.3 Programmation avec les EJB

La mise en œuvre des composants et des conteneurs EJB se fait en Java en étendant deux interfaces prédéfinies. L'interface de base pour un composant EJB s'appelle `EJBObject`. Via cette interface, on peut obtenir une référence unique pour l'instance de composant, et l'on peut accéder au conteneur qui l'accueille. Cette interface de base doit être étendue pour définir des types de composants EJB avec leurs fonctions métiers comme illustré dans la première définition de la figure 2.2.

```
public interface VideoStreamer extends javax.ejb.EJBObject {
  byte[] getContent() throws RemoteException ;
  int getHeight() throws RemoteException ;
  int getWidth() throws RemoteException ;
}

public interface VideoStreamerHome extends javax.ejb.EJBHome {
  VideoStreamer create() throws RemoteException, CreateException ;
  void remove(VideoStreamer instance) throws RemoteException, RemoveException ;
}
```

FIG. 2.2 – Définitions d'un type de composant et d'un conteneur EJB

L'interface de base pour mettre en œuvre un conteneur EJB s'appelle EJBHome. Cette interface fournit une référence unique pour le conteneur, et permet d'effectuer des recherches parmi les composants lui appartenant. Cette interface doit être étendue pour spécifier les opérations spécifiques à un type de conteneur donné. Par exemple, une interface *Home* spécifique qui permet la création et la destruction des composants de type `VideoStreamer` est définie en bas de la figure 2.2.

2.2.1.4 Évaluation

Les *Java Beans* constituent un pas en avant par rapport au langage Java en définissant un patron de conception pour favoriser la réutilisation des composants logiciels. En effet, les *Java Beans* proposent de mettre en place des objets Java avec des interfaces bien définies, et de documenter les paquetages créés avec des informations contenues sous forme de *Bean Infos*.

C'est à partir d'*Enterprise Java Beans* que l'on peut commencer à parler d'un modèle de composants. Les EJB renforcent les concepts de base des JB en définissant trois types de composants qui répondent aux besoins spécifiques des serveurs d'application. Cependant, l'absence de la notion d'interface client illustre que la notion d'isolation forte des composants n'est pas tout à fait adopté. La présence des conteneurs, qui prennent la charge de fournir certains aspects de type contrôle, simplifie la programmation des composants en permettant la focalisation sur les aspects métiers. Les conteneurs améliorent aussi la réutilisation des composants en permettant leur exécution dans des contextes différents. Néanmoins, l'ensemble des services proposés par les conteneurs est fixé, et semble assez restreint au niveau des fonctionnalités d'administration et de configuration. Cela réduit les possibilités d'utilisation des EJB dans un cadre différent des serveurs d'applications. De même, les conteneurs fournissent un modèle de composition à plat, assez statique et limité, et laissent aux programmeurs de composants la gestion des liaisons entre les composants. Ces propriétés nous semblent limitatives dans le cadre de la mise en place de systèmes complexes et à grande échelle.

Bien qu'ils ne représentent pas un modèle très évolué, les EJBs sont largement utilisés dans la mise en place des serveurs de commerce électronique. Le fait que ce modèle soit simple et directement compatible avec les programmes existants (legacy) Java, la présence d'outils de conception graphique et le support de la société Sun Microsystems expliquent les raisons de son succès. Néanmoins, la dépendance de

ce modèle envers l'environnement Java restreint son utilisation dans des contextes d'application avec ressources limitées comme les plates-formes embarquées.

2.2.2 CCM : le modèle de composants de CORBA

CORBA (*Common Object Request Broker Architecture*) [OMG01b] est un standard proposé par l'OMG (*Object Management Group*) pour la programmation des objets répartis. CORBA spécifie des interfaces de programmation et des protocoles de communication de type client/serveur qui permettent de déléguer la mise en œuvre de tous les aspects liés à la communication et à la synchronisation des objets répartis avec leurs environnements d'exécution sous-jacents. Le CCM (*Corba Component Model*)[OMG01a] est apparu dans la version 3 de la spécification CORBA en vue d'ajouter des composants logiciels dans l'environnement existant. Conceptuellement, le CCM peut être considéré comme étant une extension des EJB pour rendre ce modèle indépendant des langages de programmation utilisés pour développer les composants.

2.2.2.1 Modèle

CCM spécifie essentiellement une structure de composant et un environnement d'exécution fournissant des services non-fonctionnels. La suite de cette sous-section est consacrée à un survol synthétique de ce modèle ainsi qu'à la présentation des langages de description qui l'accompagnent.

Structure d'un composant CCM Un composant, d'après le modèle CCM, est une entité logicielle qui peut fournir ou recevoir des opérations à travers des interfaces typées, appelées *ports*. Comme illustré dans la figure 2.3, les composants peuvent avoir de multiples ports, ainsi que des attributs qui donnent accès à leurs paramètres configurables. CCM distingue quatre sortes de ports : les *facettes* et les *réceptacles* sont respectivement les interfaces serveur et client en mode de communication synchrone, alors que les *puits* et les *sources* sont utilisés en mode de communication asynchrone. Il est à noter que la notion de type de composant est présente dans ce modèle et correspond à l'ensemble des ports du composant.

Bien que la spécification ne fasse aucune distinction entre les interfaces fonctionnelles et les interfaces de contrôle des composants CCM, la manière dont elles sont implantées diffère. En effet, les interfaces fonctionnelles, c'est-à-dire les *facettes* et les *puits* d'un composant, sont implantées dans un langage de programmation à usage général (C++, Java, etc.). Par contre, les interfaces de contrôle des composants sont souvent générées en fonction de la description des composants et de leur environnement d'exécution. Grâce à ce support de génération d'implantations, la mise en œuvre des composants fournit uniquement des fonctions métiers, ce qui la simplifie considérablement. Parmi les interfaces de contrôle générées, nous pouvons citer l'interface référence des composants CCM qui permet d'introspecter les ports d'un composant en cours d'exécution.

Conteneurs de composants Les conteneurs fournissent tout d'abord aux composants un environnement d'execution, c'est-à-dire un emplacement mémoire et un flot d'exécution. Cet environnement fournit aussi des services non-fonctionnels comme des transactions, de la persistance, de la sécurité ou de la tolérance aux fautes ainsi que les moyens de communication basés sur le bus CORBA. Comme illustré dans la figure 2.4, chaque composant est exécuté dans un conteneur, et les conteneurs sont spécifiques à chaque type de composant. Chaque conteneur implante une *maison* de composants. La *maison* est responsable du contrôle des composants lui appartenant. Les services de contrôle implantés par une maison sont rendus accessibles à travers deux interfaces de types *fabrique* et *recherche* qui permettent respectivement de créer ou de détruire des instances de composants et d'effectuer des opérations de connexion et d'introspection des interfaces des composants contrôlés par la *maison*. Les conteneurs peuvent en quelque sorte être considérés comme des adaptateurs des composants CCM à l'environnement CORBA, fournissant les services non-fonctionnels cités précédemment.

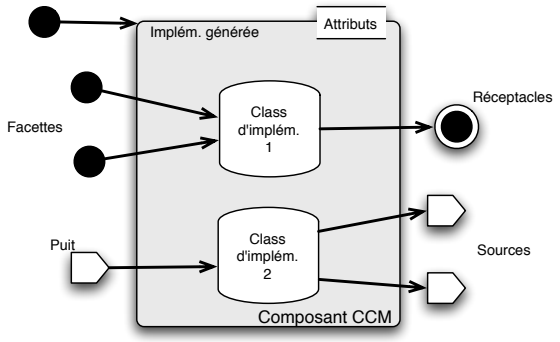


FIG. 2.3 – Structure interne d'un composant CCM.

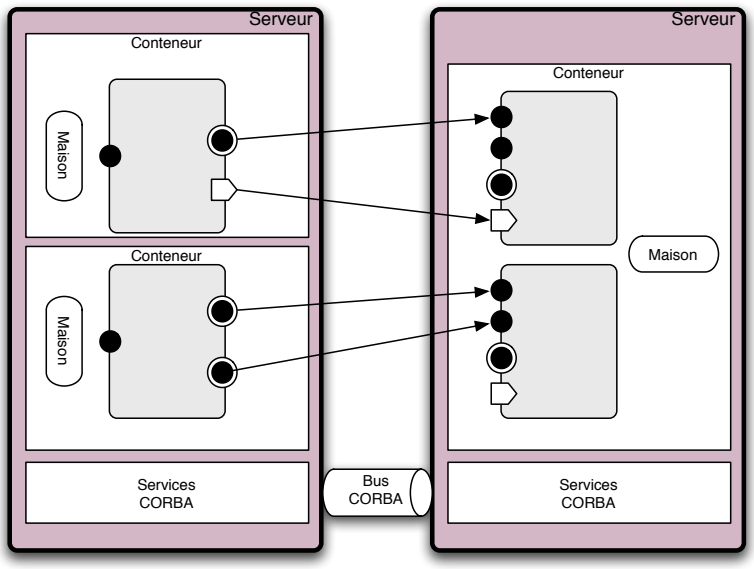


FIG. 2.4 – Représentation d'un système à multiples conteneurs à base de CCM.

Langages de description CORBA spécifie deux langages déclaratifs pour définir les types d'interfaces et pour décrire les types des composants ainsi que la description des services non-fonctionnels qu'ils requièrent de leur environnement d'exécution. Le premier langage en question est le CORBA IDL, le langage originel de description d'interfaces utilisé pour mettre en œuvre la programmation orientée objets dans cet environnement. Cet IDL est utilisé principalement pour la génération des adaptateurs de communication appropriés aux interfaces fournies par chaque objet afin de faciliter la programmation répartie. Ces adaptateurs peuvent comprendre l'encapsulation de protocoles de communication arbitrairement complexes, la conversion de format de données échangées et l'adaptation de la communication entre des objets programmés dans des langages différents. Le langage CORBA IDL est utilisé dans le cadre de CCM uniquement pour décrire le type des interfaces des composants. Afin de compléter ce dernier avec la spécification des types de composants en termes de leurs facettes, réceptacles, puits et sources, l'OMG à défini le CIDL (*Component Implementation Description Language*). À l'aide de ces informations, l'outillage CORBA est capable de générer l'implantation des interfaces *fabrique* et *recherche* proposées par les *maisons* des composants. En plus de la description des types de composants,

CIDL permet de spécifier les maisons des composants en énumérant les services environnementaux qu'elles doivent fournir aux composants qu'elles accueillent.

2.2.2.2 Implantation

Une des caractéristiques importantes de CCM est sa compatibilité avec de multiples langages de programmation pour l'implantation des composants. Deux éléments sont nécessaires pour bâtir un tel environnement de programmation multi-langages : des guides de programmation à respecter dans chacun des langages de programmations supportés, et des environnements d'exécution pour au moins chaque type d'objet binaire[1] supporté afin d'assurer leur intégration avec le reste du système. Dans ce cadre, l'OMG a standardisé des *mappings* d'interfaces pour divers langages de programmation parmi lesquels on peut citer Ada, C, C++, Java, Python et Perl [COR]. Par contre, il y a, à ce jour, un manque considérable d'environnements d'exécution qui implantent la totalité de la spécification CORBA v3 dans l'ensemble des langages ciblés ; il existe en effet soit des environnements complets destinés à la programmation des serveurs d'application en Java [Ope], soit des environnements qui implantent des sous-ensembles (ou des versions spécialisées) de cette dernière [MIC, OMG02].

2.2.2.3 Programmation avec des composants CCM

La programmation des composants CCM demande la prise en main de quatre éléments qui sont les langages IDL et CIDL, le guide de programmation à respecter pour l'implantation des composants dans le langage de programmation choisi et, finalement, l'environnement de génération de code et d'exécution utilisé pour mettre en œuvre les composants CCM.

La figure 2.5 illustre la définition dans le langage IDL d'une interface, appelée `VideoStream`, qui intègre trois opérations pour donner accès à un flux vidéo. Plus bas dans la même figure, la définition d'un type de composant est illustrée. Ce composant a deux interfaces serveur qui fonctionnent en mode synchrone pour donner accès à ses flux vidéo et audio. Grâce au mot clé `emits`, il est aussi dit que ce composant peut envoyer des signaux asynchrones pour signaler qu'il est prêt. La dernière construction qui est utilisée en bas de la figure 2.5 spécifie une *maison* par défaut qui doit être utilisée pour exécuter des composants de type `MPEG4Streamer`.

```
interface VideoStream {
  byte[] getContent();
  int getHeight() ;
  int getWidth() ;
};

component MPEG4Streamer {
  provides AudioStream audioPort;
  provides VideoStream videoPort;
  emits    ReadyEvent  ready ;
};

home MPEG4Home manages MEPG4Streamer {} ;
```

FIG. 2.5 – Mise en œuvre d'un composant dans l'environnement de programmation CCM.

Une fois les types d'interfaces et de composants déclarés, leurs implantations doivent être fournies dans un langage de programmation donné. La prise en main du guide de programmation à suivre peut être plus ou moins complexe en fonction du langage choisi. Alors que ce guide est très simple dans des

[1] Par exemple, des composants écrits en C et C++ peuvent être accueillis par le même environnement d'exécution, mais il en faut un autre pour Java.

langages orientés objets comme C++ et Java, il peut être assez complexe en C où il faut explicitement rajouter des paramètres dans les listes d'arguments des fonctions pour donner accès à l'environnement CORBA, et utiliser un certain nombre de macros pour accéder aux données d'instance des composants CCM.

2.2.2.4 Evaluation

Le CCM est un modèle de composants qui s'insère dans la logique adoptée par CORBA. Tout en restant compatible avec plusieurs langages d'implémentation, le CCM prétend être, et ce à juste titre, une solution pour la programmation répartie. La possibilité d'implantation d'interfaces serveur multiples et la définition d'interfaces client améliorent le niveau d'identification des éléments architecturaux par rapport aux objets CORBA. De plus, des langages de description comme CORBA IDL et CIDL accompagnent le modèle pour permettre la mise en œuvre des descriptions de haut niveau, et facilitent la programmation en générant une partie du code nécessaire à l'implantation des composants.

Néanmoins, le modèle de composition à plat des composants CCM au sein des conteneurs reste à un niveau équivalent à celui des EJBs, et nous semble limitatif pour la mise en place de systèmes complexes. L'obligation de l'implantation d'un nombre fixe de services non-fonctionnels pour tout composant CCM est l'un des inconvénients majeurs de ce modèle : l'implantation de cet ensemble de services semble coûteuse pour des plates-formes à ressources limitées comme des systèmes embarqués, alors que ces fonctionnalités fixées peuvent ne pas répondre au niveau d'administrabilité requis par certains systèmes, comme le type de serveur d'applications autonomes. De même, la structure monolithique des conteneurs qui assurent un nombre fixe de services non-fonctionnels rend plus difficile l'adaptation des applications à base de CCM à des plates-formes réclamant des besoins spécifiques. Pour répondre à ce type de besoin, diverses versions spécialisées de CCM ont été publiées et implantées, mais en posant alors des problèmes de compatibilité avec les applications CCM standard. Dans ce contexte, [HTM+05] propose d'opter pour un langage de configuration de haut niveau combiné avec un mécanisme d'adaptation dynamique pour répondre au problème de spécialisation des conteneurs pour les composant CCM. Enfin, il est nécessaire de remarquer qu'il existe à ce jour un nombre important de projets industriels et académiques qui portent sur des implantations ou utilisations des plates-formes à base de CCM.

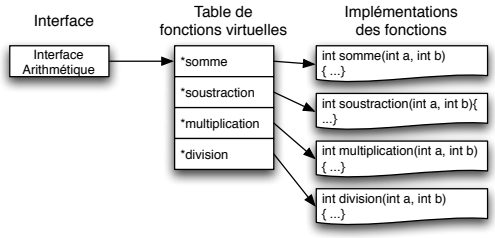
2.2.3 COM/DCOM/COM+

La famille COM (*Component Object Model*) [Box98] constitue la solution de Microsoft dans le domaine des composants. COM est en effet plus un modèle binaire qui résulte d'une réflexion sur « comment implanter des composants logiciels présents au cours de l'exécution » qu'une réflexion conceptuelle comme c'est le cas dans la plupart des modèles à composants. De ce fait, COM met en place un modèle qui permet l'inter-opération des modules logiciels, éventuellement écrits avec des langages différents. DCOM (*Distributed Component Object Model*) représente une version enrichie du modèle de base qui est COM pour permettre la programmation de composants répartis. Quant à COM+, il s'agit non pas d'un modèle mais plutôt d'un environnement d'exécution pour des composants COM/DCOM qui est inclus dans les systèmes MS-Windows depuis la version 2000.

2.2.3.1 Modèle

Structure d'un composant COM Le modèle de composants COM est avant tout un format binaire d'interface qui permet de séparer l'utilisation des opérations implantées dans des modules logiciels de leurs détails d'implantations. Pour ce faire, COM étend la technologie de table de fonctions virtuelles avec une autre indirection qui représente une référence d'interface (Figure 2.6). Dans ces conditions, une interface est une collection d'opérations implantées par un composant. COM ne fait aucune hypothèse sur la manière dont ces opérations sont implantées : l'ensemble des opérations peuvent être implantées

dans une classe C++ au même titre qu'elles peuvent être implantées dans des modules C distincts. En effet, COM favorise l'héritage d'interface à la place de l'héritage d'implantation pour n'avoir aucune hypothèse sur la nature de l'implantation.



(a)

```
struct {
  (int *) addition(int, int) ;
  (int *) soustraction(int, int) ;
  (int *) multiplication(int, int) ;
  (int *) division(int, int) ;
}VFTArithmetique;

struct {
  struct VFTArithmetique *vft ;
} RArithmetique ;
```

(b)

```
class Arithmetique {
  int addition(int, int) = 0 ;
  int soustraction(int, int) = 0 ;
  int multiplication(int, int) = 0 ;
  int division(int, int) = 0 ;
} ;
```

(c)

FIG. 2.6 – Structure binaire d'une interface COM (a) et son codage en C (b) et C++ (c).

Un composant peut implanter plusieurs interfaces ; ceci constitue la manière de supporter le polymorphisme dans le paradigme composant. En revanche, la notion d'interface client n'est pas clairement identifiée ; les implantations peuvent néanmoins contenir des références vers d'autres composants. COM spécifie une interface par défaut qui doit être implantée par tous les composants : il s'agit de l'interface *IUnknown* qui fournit les opérations de compteur de références pour comptabiliser les clients qui maintiennent des références vers le composant, et l'opération *QueryInterface* qui permet d'obtenir une référence vers une autre interface implantée par le composant. Cette dernière peut être considérée comme étant une opération de *cast* dynamique qui fournit un accès transparent au client vis-à-vis du langage d'implantation utilisé pour programmer le composant.

Modèles de composition COM ne spécifie pas de modèle de conteneur pour accueillir les composants ; cette responsabilité est attribuée de manière globale à l'environnement MS-Windows. Quant au modèle de collaboration des composants COM, deux modèles intitulés la *composition* et l'*aggrégation*, sont distingués (Figure 2.7).

Dans le cas du modèle appelé *composition* (Figure 2.7.a), il s'agit d'un composant unique qui maintient une référence vers un autre composant. Étant donné que le deuxième composant n'est connu que par le premier, on dit que le premier composant est composé du deuxième. En suivant cette logique, un composant peut être composé de plusieurs composants, ce qui permet d'avoir une organisation simple pour tracer les interdépendances des composants.

Dans le cas de l'*agrégation* (Figure 2.7.b), l'implantation d'une interface d'un composant est typiquement déléguée à un autre. Du point de vue du client, il s'agit d'un composant à interfaces multiples alors qu'éventuellement l'implantation de ce composant est assurée de manière collaborative par plu-

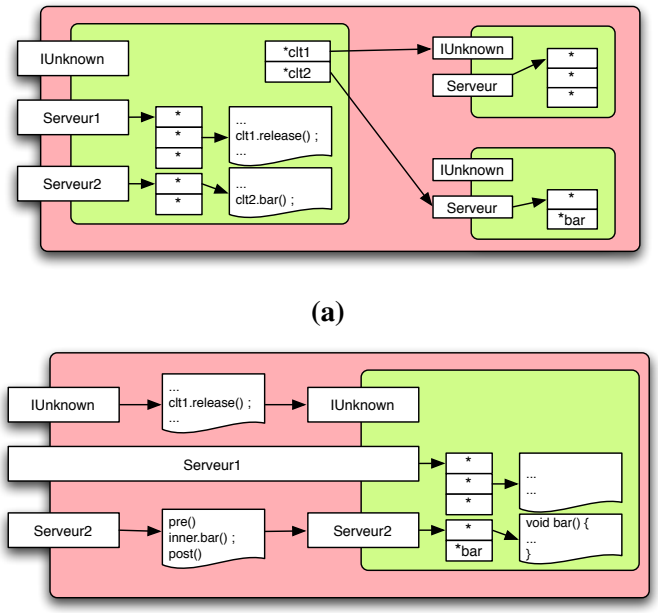


FIG. 2.7 – Les deux modèles de collaboration des composants COM : la *composition* (a) et l'*agrégation* (b).

sieurs composants. Le cas de l'*agrégation* est typique pour implanter des *wrappers* de composants ou des intercepteurs d'interfaces pour effectuer des pre- et post-traitements sur les invocations reçues.

2.2.3.2 Implantation

L'implantation des composants COM peut être effectuée dans des langages de programmation différents en respectant le format d'interface binaire spécifié. Pour unifier la définition des interfaces, Microsoft fournit un langage de description d'interfaces, appelé COM IDL et son compilateur qui peut cibler au moins le C et le C++. Ce compilateur traduit les interfaces décrites en COM IDL vers des structures de données adéquates dans le cas de C, et vers des classes virtuelles dans le cas de C++.

L'instanciation des composants COM, appelée l'*activation*, se fait via le service SCM (*Service Control Manager*) fourni dans l'environnement MS-Windows. Ce service procède à l'instanciation d'un nouveau composant à partir d'un ensemble de définitions d'interfaces et de modules d'implantations et retourne l'interface *IUnknown* du nouveau composant en cas de réussite. COM+ diverge de COM dans l'implantation du service d'instanciation. En effet, COM+ fournit un environnement d'exécution plus sophistiqué, appelé la *ferme de composants*. Ceci permet de réutiliser, en réponse à une requête d'instanciation, des instances de composants déjà présentes mais non-actives en réinitialisant leur cycle de vie. L'environnement d'exécution pour COM fournit aussi un service de répertoire, intitulé *GUID*, qui permet de retrouver des références vers des composants en utilisant des clés globales.

Il est à noter que COM définit un modèle binaire pour l'interaction des composants existant dans le même espace d'adressage. DCOM étend ce dernier avec deux éléments pour en faire une technologie de programmation pour systèmes répartis. Le premier élément consiste en un générateur de couples stub/squelette basé sur la technologie RPC (*Remote Procedure Call*) pour transporter les appels locaux émis par un client jusqu'au serveur destinataire, et de reporter le résultat d'exécution au client de manière transparente. Ce générateur de chaîne de communication utilise en effet les descriptions d'interfaces

spécifiées dans le langage COM IDL, et génère une implantation spécialisée du protocole propriétaire MSRPC. Le deuxième élément introduit avec DCOM est un ramasse-miettes réparti qui permet la suppression automatique des composants non-utilisés en se basant sur le compteur de références fourni via l'interface *IUnknown*.

2.2.3.3 Evaluation

Notons que contrairement à la plupart des modèles conceptuels de composants présentés dans cet état de l'art, COM propose avant tout un modèle binaire pour mettre en place des composants logiciels. Ce modèle augmente en effet l'implantation binaire des objets (en particulier celui de C++), pour en faire des composants qui peuvent fournir plusieurs interfaces distinctes. La définition d'interface client reste optionnelle, et ne fait pas partie des préoccupations principales du modèle COM. De même, la définition des interfaces d'un composant qui répondrait aux problèmes de la conception est possible à l'aide des outils proposés par Microsoft mais reste optionnelle. L'utilisation de ces outils est malheureusement obligatoire pour créer des bibliothèques, des exécutables ou générer le code d'assemblage de composants, ce qui limite l'utilisation des composants COM au cadre de MS Windows. La philosophie Microsoft est aussi maintenue en ce qui concerne les dépendances à l'exécution des composants COM à l'environnement Windows : c'est en effet Windows qui implante l'équivalent d'un conteneur qui fournit l'environnement d'exécution adéquat.

Les composants COM permettent la programmation dans les langages C et C++, ce dernier étant le plus utilisé à cause du confort de programmation. COM est largement utilisé dans les solutions intégrées à Windows comme ActiveX et DirectX. On retrouve ainsi les concepts de ce modèle dans le cœur de .Net qui semble être son successeur.

2.2.4 SystemC

SystemC [OSC05] est une extension du langage C++ proposée par OSCI (*Open SystemC Initiative*) qui regroupe un nombre important d'industriels dans le domaine de la conception de produits microélectroniques pour modéliser des systèmes matériels. Depuis sa version 2.1 publiée en 2005, SystemC intègre un modèle de composants qui est intéressant pour la modélisation des systèmes, au sens large du terme. Dans la suite de cette sous-section, nous présentons le modèle de composants de SystemC et nous donnons quelques informations sur son environnement d'exécution sous-jacent.

2.2.4.1 Modèle

Structure des composants SystemC Les *modules* constituent les éléments d'encapsulation de comportements et de données en SystemC. Les modules peuvent interagir avec d'autres modules se trouvant dans leur environnement par le biais de leurs *ports*. Les *ports* permettent des échanges de données en entrée ou en sortie et sont définis par un type d'interface et par une cardinalité. En d'autres termes, un *port* encapsule un nombre défini d'interfaces de même type. Une interface définit le type des données qui peuvent être échangées ainsi que le sens de l'échange. Les ports des modules ont besoin d'être connectés pour établir leurs interactions. SystemC propose une classe de base qui définit certaines opérations par défaut pour la mise en place de différents types de connecteurs afin de permettre l'échange de différents types de données entre les modules, éventuellement avec des protocoles d'interactions sophistiqués. Un ensemble de connecteurs pré-définis est aussi proposé. Cet ensemble comprend les *signaux* pour échanger des valeurs avec une synchronisation événementielle, des *tampons* avec des sémantiques de remplissage et de vidage différentes pour des échanges de données asynchrones, et des *mutex* et des *sémaphores* pour effectuer uniquement des synchronisations. Enfin, SystemC permet l'utilisation des *objets*, qui ne font pas partie des éléments architecturaux, mais qui sont fournis pour aider la programmation de certains aspects dynamiques.

Modèle d'exécution des composants Les *modules* SystemC peuvent êtres actifs et/ou réactifs. Deux modèles d'exécution sont proposés pour les modules actifs : *sc_thread* associe un flot d'exécution classique à une opération implantée dans un module alors que *sc_cthread* (*clocked thread*) associe un flot d'exécution synchrone, contrôlé par une base de temps gérée via une horloge présente dans le système. Les opérations d'un module peuvent aussi être réactives (*sc_method*), ce qui permet de les exécuter en réaction à l'occurence d'un événement. Dans ce cas, une liste de ports d'entrée qui peuvent déclencher une réaction est spécifiée pour chaque opération réactive. Il est important de noter que les actions et les réactions sont associées aux opérations et non pas aux modules, ce qui permet aux modules d'encapsuler à la fois des opérations actives et réactives.

Modèle d'instanciation Les *modules* SystemC sont définis pour la modélisation des composants matériels. Par conséquent, ils sont instanciés une fois pour toutes avant la simulation du système, et ne peuvent êtres détruits ou changer de configuration au cours de la simulation. Le cycle de vie d'un programme SystemC commence par une phase d'élaboration où tous les modules sont instanciés et interconnectés et continue par la simulation du système jusqu'à ce qu'un des modules fasse appel à la procédure de terminaison. En revanche, les *objets* qui ne font pas partie des éléments de modélisation architecturale, peuvent être instanciés et détruits dynamiquement.

Modèle de composition Alors que la première version de SystemC supportait uniquement un modèle de composition à plat, certaines constructions ont été rajoutées à partir de la version 2 pour permettre la composition hiérarchique. Les modules peuvent donc contenir d'autres instances de modules, et configurer leurs connections. Pour ce faire, ils doivent décrire cette partie de gestion de configuration dans une procédure pré-définie qui sera appelée pendant la procédure d'élaboration du système global. Le partage de module n'est pas permis ; une configuration de système en SystemC est forcément représentée sous forme d'arbre. Enfin, un support d'introspection est défini par SystemC pour découvrir, au cours de l'exécution, la configuration du système en prenant en compte les objets qui sont instanciés dynamiquement.

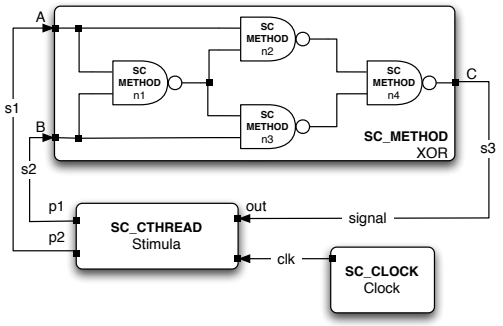
2.2.4.2 Implantation

SystemC fournit deux éléments pour étendre le langage C++ afin d'y intégrer les concepts présentés ci-dessus. Premièrement, une librairie est fournie pour la définition des classes modèles et des procédures qui sont destinées à être utilisées par les programmeurs. Cette libraire comprend, entre autres, un fichier en-tête principal qui fournit les définitions des nombreuses *macros* qui doivent être utilisées pour définir et manipuler des *modules*, des *ports*, des *interfaces*, etc. Le deuxième élément consiste en une librairie *runtime* qui est destinée à être liée avec les programmes SystemC. Cette librairie fournit le point d'entrée des programmes SystemC et assure leur élaboration et leur simulation. Cette dernière fournit aussi un certain nombre d'outils qui permettent d'observer et de contrôler la simulation au cours de son exécution.

2.2.4.3 Programmation en SystemC

Le syntaxe du langage SystemC est celle de C++. SystemC y rajoute uniquement des *macros* et des classes *templates* pour faciliter la description et l'implantation des composants. La figure 2.8 illustre la description d'un module qui implante l'opération booléenne *xor*. Ce module est composé de quatre opérateurs *nand* et définit leurs interconnexions. Les composants *nand* sont des modules réactifs ; ils réagissent à chaque changement de valeurs à leurs entrées A et B. Étant donné que le composant *xor* est constitué d'une simple composition de ces composants, ce dernier est aussi un composant réactif.

Le composant *stimula* est le module actif du système présenté dans la figure 2.8. Il est censé générer des vecteurs de tests pour vérifier le bon fonctionnement du composant *xor*. Pour ce faire, il envoie des signaux d'entrée via les signaux *s1* et *s2* et vérifie la correction du signal de résultat qu'il reçoit via le



(a)

```
SC_MODULE(xor) {
  // Les ports
  sc_in<bool> A, B;
  sc_out<bool> C;
  // Les sous-composants
  nand n1, n2, n3, n4;
  // Les connecteurs
  sc_signal<bool> S1, S2, S3;

  // Constructeur
  SC_CTOR(xor) : n1("N1"), n2("N2"), ... {
    // Desription des connections
    n1.A(A);
    n1.B(B);
    n1.C(S1);
    ....
    n4.A(S2);
    n4.B(S3);
    n4.C(C);
  }
};
```

(b)

FIG. 2.8 – (a) L'architecture d'un système de simulation en SystemC. (b) Le code qui décrit le composant *xor* qui est composé de quatre instances de composants *nand*.

signal *s3*. Le flot d'exécution du composant *stimula* est de type synchrone et sa base de temps est gérée à l'aide du signal d'horloge qu'il reçoit de la part du composant *clock*.

2.2.4.4 Evaluation

Bien qu'il ne s'agisse pas d'un modèle de composants pour la programmation de systèmes logiciels, nous avons tenu à intégrer le SystemC dans cet état de l'art car il présente de nombreuses propriétés intéressantes. Tout d'abord, les composants SystemC ont une propriété d'isolation forte : toute interaction entre composants se fait au travers des ports d'entrée et de sortie. Contrairement à la plupart des modèles, l'interaction entre composants SystemC ne se fait pas par des émissions d'opérations mais par échange de données ou de signaux. En conséquence, les ports SystemC sont caractérisés par le sens de l'échange de données (ou de signaux) et non par l'émission et la réception d'opérations. La connectique entre les composants est explicitement déclarée, et fait partie des éléments architecturaux. De plus, la sémantique de communication est directement implantée au sein des connexions et différents protocoles sont supportés. Bien que SystemC propose certains types de connexion standards, les programmeurs sont libres de programmer les leurs pour satisfaire leurs besoins spécifiques.

On distingue deux types de composants en fonction de leurs modèles d'exécution. En effet, on distingue à ce niveau des composants actifs et des composants passifs. Les composants actifs peuvent être de nature asynchrone ou synchrone, c'est-à-dire cadencés par une horloge. Ces types de composants permettent de modéliser divers types de composants électroniques. La composition hiérarchique est permise, mais ceci est malheureusement mélangé au code fonctionnel, ce qui empêche d'avoir une vue architecturale claire à la conception. L'absence de possibilité d'instanciation dynamique de composants s'explique à juste titre pour la modélisation des circuits électroniques, mais constitue une limitation importante pour la mise en place de systèmes logiciels. Enfin, rappelons que SystemC est largement utilisé pour la modélisation fonctionnelle et transactionnelle dans le domaine de la conception de systèmes électroniques en raison de ses atouts en ce qui concerne les outils de programmation, de deboguage et d'analyse. L'ab-

sence de générateur RTL depuis des modèles SystemC empêche cependant son déploiement dans un processus complet de conception de circuits.

2.3 Modèles de composants académiques

2.3.1 OpenCOM

OpenCOM est une version étendue du modèle de composants COM de Microsoft, issue des travaux de l'Université de Lancaster pour implanter l'intergiciel OpenORB. Nous jugeons intéressant d'étudier ce modèle pour ces apports au modèle COM en termes d'éléments d'architecture logicielle.

2.3.1.1 Modèle

Structure d'un composant OpenCOM étend le modèle de composants de COM qui est basé sur des *interfaces* avec deux éléments : les *réceptacles* et les *connexions*. Les *réceptacles* identifient les services requis par un composant. Dans ce sens, les *réceptacles* constituent les interfaces client des composants OpenCOM. Un composant peut disposer de multiples *réceptacles* pour accéder aux services de types différents, et un *réceptacle* peut contenir de multiples pointeurs pour être connecté à plusieurs fournisseurs d'un même type de service. Pour permettre des reconfigurations dynamiques, OpenCOM permet d'attendre qu'un composant ne contienne plus aucune activité en bloquant les *réceptacles*. Un *réceptacle* doit être connecté à une *interface* pour y émettre une opération. Chaque composant OpenCOM doit implanter une interface intitulée *IConnections* qui permet de modifier les connexions établies sur les réceptacles du composant. Par ce biais, les connexions des composants peuvent être contrôlées par des entités extérieures. Une deuxième interface intitulée *ILifeCycle* doit être fournie par chaque composant de manière à implanter les opérations *start* et *stop* qui seraient appelées par l'environnement de déploiement au moment de la construction et de la destruction du composant.

Support pour la réflexivité Chaque composant OpenCOM doit implanter les interfaces suivantes pour fournir à son environnement des opérations d'introspection et d'interception :

IMetaInterface fournit des opérations d'introspection des types d'interfaces et des réceptacles d'un composant.

IMetaInterception permet de placer des composants d'interception sur les interfaces du composant. Via ce mécanisme d'interception, le comportement du composant peut être contrôlé en effectuant des pré- et post-traitements.

IMetaArchitecture permet d'identifier les connexions établies entre les réceptacles du composant et les interfaces d'autres composants. Cette interface permet entre autres de reconstituer le graphe d'interconnexion d'un système OpenCOM.

Modèle de composition et de déploiement L'unité de composition des composants OpenCOM est l'espace d'adressage. Chaque espace d'adressage contient un environnement d'exécution appelé *OpenCOM* qui implante les opérations de déploiement pour les composants qui lui sont associés. Cet environnement fournit une interface intitulée *IOpenCOM* qui centralise toutes les opérations de création/destruction des composants et la gestion de leurs interconnexions qui pourraient s'effectuer par la demande de tierces parties. L'environnement *OpenCOM* maintient, en plus, la représentation de ses composants et de leurs interconnexions, et la rend disponible sous un format bien défini de graphe à travers l'interface *IOpenCOM*.

2.3.1.2 Implantation

L'implantation du modèle OpenCOM respecte l'implantation du modèle COM tout en y ajoutant de nouveaux aspects comme les *réceptacles* et les interfaces de meta-niveaux présentées ci-dessus. Le langage d'implantation est C++ , mais le format binaire d'implantation reste compatible avec C et d'autres

langages compatibles avec COM. Dans le cadre de ce document, nous ne rentrerons pas plus dans les détails en ce qui concerne l'implantation du modèle OpenCOM. Celle-ci est en effet présentée en détail dans [CBCP01]. Nous tenons uniquement à noter que l'assemblage de structures de données utilisées pour implanter le code des meta-interfaces de manière générique fait partie des sources d'inspiration de l'implantation du canevas Think présenté dans la section 4.4.

2.3.1.3 Evaluation

OpenCOM est une évolution du modèle COM en vue de la mise en place d'intergiciels réflexifs et reconfigurables. Les contributions de ce modèle à COM commencent par la définition des services requis (ce qui est optionnel dans COM) et par l'identification des connexions comme étant des éléments architecturaux. La deuxième contribution majeure d'OpenCOM est d'avoir défini un méta-niveau pour tous les composants. Ce méta-niveau fournit essentiellement des services d'introspection et de reconfiguration dynamique des composants. Certaines limitations de la version 1 du modèle, comme les dépendances à l'environnement Windows et le nombre fixé des fonctions de contrôle, ont été supprimées dans sa version 2.

OpenCom v2 a été récemment doté d'un canevas de chargement et de déploiement dynamique [CBG+04], ce qui renforce sa crédibilité dans le domaine des intergiciels reconfigurables. Néanmoins, l'absence d'un modèle adéquat pour les composants partagés reste parmi les limitations de la version actuelle. Bien que ce modèle fasse l'objet de plusieurs travaux de recherche, on peut noter l'absence d'outils d'aide à la programmation des composants et de vérification des architectures à déployer.

2.3.2 ArchJava

ArchJava [ACN02b, ACN02a], proposé par l'Université Carnegie Mellon, occupe une place toute particulière dans l'état de l'art de la programmation à base de composants car il constitue une des premières propositions qui visent à définir un langage à composants. La particularité de cette approche vient de l'intégration des éléments architecturaux et de l'implémentation des composants au sein d'un même langage qui prend forme d'une extension au langage Java.

2.3.2.1 Modèle

Structure d'un composant Les *composants* sont les seules unités d'encapsulation définies dans le langage ArchJava. Les composants communiquent avec d'autres composants via des *ports*. Un port définit un ensemble de méthodes, au sens Java classique, qui définissent des points d'accès à un composant. Dans ce sens, les ports *ArchJava* peuvent être considérés comme des interfaces des composants au sens large. Un composant peut avoir plusieurs *ports*. Contrairement à la plupart des modèles qui définissent un sens associé à chaque interface (*serveur/client* ou *fourni/requis*), aucune propriété de ce type n'est associée aux *ports* dans ArchJava. En effet, ce sont les méthodes associées à un port qui vont donner ce type d'informations. Dans ces termes, une méthode peut être soit de type *fourni* (implantée par le composant), soit de type *requis* ou *diffusion* (appelée par le composant). Une méthode *diffusion* correspond à une méthode requise sans valeur de retour ; un port contenant des méthodes *diffusion* peut être connecté à plusieurs composants, auquel cas l'invocation de la méthode sera transmise à tous.

Intégrité de la communication Les ports des composants doivent êtres interconnectés pour établir la communication. Etant donné qu'ArchJava intègre la spécification des connections et l'implantation des composants dans un même langage de programmation, il est capable d'interdire toute communication inter-composants qui ne passe pas via des connections établies. Autrement dit, contrairement à ce qui est valable pour les objets de manière générale, il ne suffit pas d'obtenir une référence à un autre composant pour en appeler des opérations. Ceci est une propriété très forte, car la cohérence entre l'implantation et l'architecture est ainsi assurée. De plus, cette propriété est assurée pour tout composant instancié dyna-

miquement, ce qui permet d'observer l'évolution de l'architecture du logiciel au cours de son exécution en termes des composants et de leurs interconnexions.

Modèle d'implantation et de composition ArchJava permet la composition hiérarchique des composants dans le sens où un composant peut posséder des sous-composants. Ainsi, l'implantation des méthodes fournies d'un composant peut être faite directement dans le composant en tant que méthode classique, ou être déléguée à une implantation fournie par un sous-composant. Les connexions entre les composants sont gérées par leur père de premier niveau, c'est à dire le composant à qui ils appartiennent. La structure hiérarchique d'une configuration ArchJava est représentée par un arbre, le partage de composants n'étant pas supporté.

Abstraction pour les connecteurs Originellement, le modèle de communication entre les composants ArchJava était limité à des invocations classiques de méthode [ACN02b]. Récemment, une abstraction a été proposée pour mettre en œuvre des connecteurs supportant diverses sémantiques de communication [ASCN03]. Cette proposition consiste en la définition de classes de bases pour les *ports* et les *connecteurs* qui fournissent des opérations d'introspection sur la signature des méthodes et qui spécifient des méthodes abstraites que doivent fournir les implantations concrètes des *connecteurs*. De cette manière, les programmeurs de connecteurs peuvent mettre en place des *stubs* et des *skeletons* génériques en utilisant la puissance de la programmation réflexive. Cette approche fournit une abstraction homogène et robuste pour la construction de moyens de communication supportant tous les types de communication énumérés dans [SDZ96]. Néanmoins, l'utilisation de la technologie de réflexivité nous semble un choix peu efficace en ce qui concerne la performance à l'exécution.

2.3.2.2 Implantation

ArchJava est une extension du langage Java définissant des *composants*, des *ports* et des *connexions* avec leurs sémantiques associées. L'implantation d'ArchJava est basée sur un pré-processeur et une librairie *runtime*. Le pré-processeur transforme les programmes ArchJava en Java classique en traduisant ses mots clés en des constructions équivalentes dans le langage hôte. La librairie fournit les définitions et les opérations de base associées aux composants et au connexions. Cette libraire implante en plus l'ensemble des opérations de vérification qui sont effectuées au cours de l'exécution pour assurer l'intégrité des communications. [AL03] présente une évaluation de l'implantation d'ArchJava. Les conclusions de ce rapport montrent qu'ArchJava apporte des bénéfices considérables en matière de conception de logiciels flexibles alors qu'il présente certaines surcharges en termes de taille de code binaire et de performance d'exécution.

2.3.2.3 Programmation en ArchJava

Pour illustrer la programmation en ArchJava, nous proposons de reprendre l'exemple du compilateur présenté dans [ACN02b]. La figure 2.10.a présente l'architecture d'un compilateur formé de trois composants. L'implantation d'un composant primitif, tel que l'exemple du composant *parser* (voir la figure 2.9) est constituée de la définition de ses ports d'entrée et de sortie, et de l'implantation des méthodes fournies.

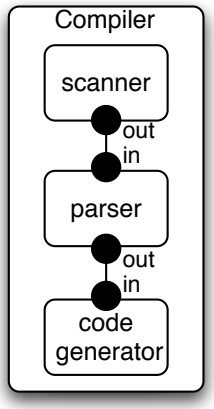
Dans le cas d'un composant composite, tel que l'application compilateur (voir la figure 2.10.b), les sous-composants sont déclarés comme des champs privés et finaux. De plus, les interconnexions entre les ports des sous-composants sont définies à ce niveau. Il est à noter qu'un composant composite peut exporter ou importer les interfaces de ses sous-composants et peut invoquer directement des opérations de ses sous-composants de premier ordre.

```
public component class Parser {
  public port in {
    provides void setInfo(Token symbol, SymTabEntry e);
    requires Token nextToken() throws ScanException;
  }
  public port out {
    provides SymTabEntry getInfo(Token t);
    requires void compile(AST ast);
  }

  void setInfo(Token t, SymTabEntry e) { ... };

  SymTabEntry getInfo(Token t) { ... };
  ...
}
```

FIG. 2.9 – Description du composant primitif *Parser* en ArchJava.



(a)

```
public component class Compiler {
  private final Scanner scanner = ... ;
  private final Parser parser = ... ;
  private final CodeGen codegen = ... ;

  connect scanner.out, parser.in;
  connect parser.out, codegen.in;

  public static void main (String args[]){
    new Compiler().compile(args);
  }
  public void compile(String args[]) {
   // for each file in args do:
    ... parser.parse(file); ...
  }
 }
```

(b)

FIG. 2.10 – L'architecture d'un compilateur à base de trois composants principaux (a), et sa description en ArchJava (b).

2.3.2.4 Evaluation

ArchJava est l'un des premiers travaux qui visent à définir un langage de programmation à base de composants. Dans ce cadre, ArchJava étend le langage Java pour combiner l'implantation des composants avec leurs définitions architecturales. Ce langage supporte la plupart des propriétés que l'on trouve dans des modèles de composants modernes. L'isolation des composants se fait par le biais des ports. Les ports sont des interfaces bidirectionnelles, capables à la fois d'émettre et de recevoir des opérations. Les connexions entre composants sont identifiées comme des éléments architecturaux, et le langage assure l'intégrité de la communication qui est limitée aux composants interconnectés. Le modèle de composition adopté est hiérarchique, mais ne supporte pas le partage de composants.

Parmi les critiques au langage ArchJava, nous pouvons citer l'absence de méta-niveau qui permettrait de contrôler le comportement des composants, la dépendance envers l'environnement Java qui limiterait son utilisation dans un cadre applicatif restreint, et le fait que la programmation des connecteurs flexibles soit uniquement basée sur la technologie de réflexivité, ce qui nous interroge sur les performances en l'absence de résultats concrets publiés.

2.3.3 Classages

Classages [LS05] est une proposition académique de langage qui s'insère dans le cadre de la programmation basée sur des interactions (IOP). Brièvement, IOP fournit des patrons de coordination afin de rendre explicites et aisément maîtrisables les interactions entre composants logiciels. Même si Classages ne prétend pas être un travail sur les modèles de composants, certains de ses concepts clés nous semblent relever de ce domaine. Parmi ces contributions, nous pouvons citer les connecteurs différenciés pour les interfaces internes et externes des composants, et la différentiation des relations statiques et dynamiques entre entités logicielles.

2.3.3.1 Modèle

Interactions statiques Dans le langage Classages, les *classages* correspondent aux entités d'implantation. Deux types de *classages* sont identifiés : les *classages atomiques* sont fournis directement par l'implantation de méthodes alors que les *classages composites* sont implantés par la composition de deux *classages*. En d'autres termes, les *classages atomiques* sont des entités de bases alors que les *classages composites* sont des entités de réutilisation. L'opération de composition se fait avec des *mixers* qui définissent le sens de l'interaction entre deux entités mixées. L'opération de mixage telle que définit dans Classages fournit un modèle plus clair que l'héritage classique pour les interactions statiques entre composants logiciels. En effet, dans le modèle d'héritage, il peut y avoir une interaction bidirectionnelle entre le père et le fils [2] alors que celui-ci est généralement modélisé comme étant une relation unidirectionnelle où le fils utilise le père.

Interactions dynamiques Les instances de *classages* sont appelés des *objectages*. Deux types d'interactions dynamiques entre *objectages* sont identifiés : les interactions internes et externes. Les interactions internes d'un *objectage*, c'est-à-dire les interactions avec des entités qu'il a lui-même créées, se font au travers de *pluggers*. Les interactions avec les autres se font par des *connecteurs*. Les *connecteurs* et *les pluggers* sont en effet des interfaces de communication bi-directionnelles : les *méthodes exportées* sont celles qui sont implantées par l'*objectage* alors que les *méthodes importées* sont celles qui sont utilisées. Les connexions établies via des *connecteurs* ou des *pluggers* peuvent être de durée de vie longue, auquel cas elles peuvent avoir des états associés. Enfin, les interactions dynamiques ne peuvent avoir lieu qu'au travers de ces deux types d'interfaces et on ne peut en aucun cas avoir une référence directe sur un *objectage* qui donne accès à l'ensemble de ses fonctionnalités.

```
classage SequenceBase {
  connector View {
    import void init() {...}
    import void refresh() {...}
    export void getNewValue(){...}
    state Time lastUpdated
  }
  plugger Elem {
    import double getValue() {...}
    import void setValue(double v) {...}
  }
  boolean viewLocked() {...}
}
```

(b)

```
classage Sequence = SequenceBase + ...

classage C {
  connector P {...}
  int m(A x) {
    P c = connect x with P >> Q ;
    c>f(3) ;
  }
}
```

(c)

FIG. 2.11 – Extrait de programmes écrits en Classage.

[2] De manière générale, les méthodes fournies par le fils surchargent et potentiellement utilisent les méthodes du père alors que dans le cas des méthodes abstraites (par exemple en Java) c'est le père qui utilise la méthode du fils.

2.3.3.2 Implantation

Le langage Classages définit sa propre syntaxe pour intégrer les constructions proposées. Un extrait de code est présenté dans la figure 2.11. Ce langage est traduit en Java ; on ne dispose pas d'un compilateur complet fournissant du binaire. L'implantation actuelle du compilateur Classages [Cla] fournit un support complet pour le langage, mais souffre de problèmes d'inefficacité à cause de l'utilisation massive du *downcasting* nécessaire à l'implantation des *connecteurs* et des *pluggers* et de la duplication du code en raison de la mise à plat de l'héritage pour les opérations de mixage.

2.3.3.3 Evaluation

Classages est un nouveau langage qui est conçu pour fournir une isolation forte des composants logiciels en maîtrisant leurs interactions dynamiques et statiques. Le modèle d'interaction dynamique proposé par Classages est proche de celui d'ArchJava. Il est basé sur des ports qui implantent des communications bidirectionnelles. Le modèle d'interaction statique est plutôt original ; il utilise la composition de composants à grain fin (mixage de méthodes) pour remplacer plus clairement l'héritage classique. Comparé aux langages orientés objets classiques, Classages force le respect des interfaces en interdisant le *downcasting* de références d'interfaces pour obtenir une référence à l'objet qui les implante. Parmi les contributions de ce langage, nous tenons aussi à citer la distinction entre les interactions d'un composant avec ses éléments internes (domaine de possession) et externes (domaine de collaboration).

Les points négatives que l'on peut trouver dans Classages sont proches des inconvénients d'ArchJava : absence de partage de composants et implantation uniquement dans un environnement Java. Etant donné que Classages est issu de travaux très récents, nous pensons qu'il sera très intéressant de suivre son évolution dans l'avenir proche.

2.3.4 Fractal

FRACTAL [BCS03] est un modèle de composants ouvert et réflexif dédié à la construction de systèmes. Il est utilisé dans divers projets de recherche allant de la mise en place de systèmes d'exploitation spécialisables pour des plates-formes embarquées [FSLM02, ÖJS05] jusqu'à la mise en place de systèmes autonomes pour des serveurs d'applications d'entreprises [BBH+05] en passant par des systèmes multimédia pour des applications mobiles [LH05]. La suite de cette sous-section présente brièvement le modèle FRACTAL pour le situer dans le cadre de l'état de l'art sur les modèles de composants. Une présentation plus détaillée de ce modèle est proposée dans la section 4.1.

2.3.4.1 Modèle

Structure des composants FRACTAL est un modèle hiérarchique qui distingue deux sortes de composants, les primitifs et les composites. Les composants composites sont formés à partir d'autres composants primitifs ou composites (voir la Figure 2.12). Ceci permet une construction récursive qui s'arrête au niveau des composants primitifs qui sont directement programmés dans un langage de programmation donné. Le modèle FRACTAL permet le partage de composants, c'est-à-dire que plusieurs composites peuvent contenir une même instance d'un composant, que ce soit un composite ou un primitif.

Les seuls points d'accès des composants sont leurs interfaces. Un composant peut avoir plusieurs interfaces. Deux types d'interfaces sont distinguées comme dans la plupart des modèles de composants. Les interfaces client permettent d'émettre des opérations vers d'autres composants alors que les interfaces serveur permettent d'en recevoir. En plus de la nature client ou serveur, des propriétés de contingence (obligatoire ou optionnelle) et de cardinalité (acceptation de connexions simples ou multiples) sont attribuées aux interfaces. Ces propriétés permettent d'améliorer le niveau de description de l'architecture logicielle, et enrichissent entre autres les vérifications architecturales que l'on peut implanter.

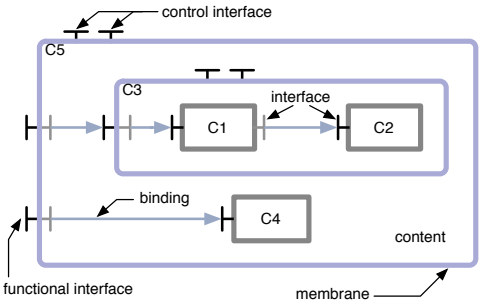


FIG. 2.12 – Un exemple de composant FRACTAL.

Liaisons flexibles La communication entre une interface client et une interface serveur n'est possible que si ces deux interfaces sont interconnectées par une *liaison*. Contrairement à la plupart des modèles, les *liaisons* définies dans FRACTAL n'imposent aucun modèle de communication. En effet, FRACTAL distingue deux types de *liaisons* pour permettre l'implantation de communications inter-composants de sémantiques différentes. Les *liaisons* primitives sont des liaisons langage et par conséquent sont conformes à la sémantique définie par le langage d'implantation des composants (e.g. dispatching de messages synchrone en Java). Des sémantiques autres que celle qui est proposée par le langage d'implantation peuvent êtres implantées avec des *liaisons complexes*. En effet, une *liaison complexe* est réifiée par un composant de complexité arbitraire, dont le rôle est d'implanter une forme de communication donnée (e.g. RPC, JNI).

Séparation du contrôle et du métier FRACTAL permet de séparer l'implantation du métier du composant de sa partie contrôle. Dans le cas de composants primitifs, le contenu est considéré comme une boîte noire dont la fonction est programmée dans un langage de programmation. Dans le cas des composites, le métier est implanté par la composition d'un ensemble de sous-composants. Schématiquement, le contenu d'un composant est entouré par une membrane qui implante la partie contrôle. Parmi les fonctions implantées dans les membranes, nous pouvons, par exemple, citer les fonctions d'interception pour tracer ou filtrer les interactions d'un composant avec son environnement.

La membrane implante aussi des interfaces additionnelles en plus des interfaces métier du composant afin de donner à l'environnement du composant des points d'accès pour les aspects de type contrôle. L'ensemble et la nature des interfaces de contrôle implantées par des composants ne sont pas fixés par le modèle : FRACTAL en propose quelques unes pour le contrôle de l'identité, des liaisons, des attributs, et de contenu basées sur des opérations réflexives comme l'introspection et de l'intercession ; d'autres peuvent être spécifiées et rajoutées en fonction des besoins spécifiques liés au contexte applicatif.

2.3.4.2 Implantation

Le modèle FRACTAL est indépendant du langage d'implantation. À l'heure actuelle, il existe plusieurs implantations de ce modèle dans des langages divers et variés, visant des contextes applicatifs différents. Parmi celles-ci, nous pouvons citer Julia [BCL+06, BCL+04] en Java, et AOKell [AOK] en AspectJ, Think [Thi] en C et C++ et FractNet [Frab] en .Net. Etant donné que les propriétés de ces implantations sont différentes, nous ne rentrons pas dans cette sous-section dans les détails de chacune. Julia et Think, qui sont utilisées dans le cadre de nos travaux, sont présentées respectivement dans les sections 4.3 et 4.4. Les lecteurs intéressés par les autres implantations pourront trouver plus d'informations sur [Fraa].

2.3.4.3 Evaluation

Les particularités du modèle FRACTAL sont multiples. Tout d'abord, le modèle est basé sur un ensemble minimal de concepts qui regroupe les éléments classiques des modèles de composants comme les composants, les interfaces et les liaisons, et y rajoute les propriétés de réflexivité et de séparation des aspects fonctionnels et de contrôles. Ce faisant, FRACTAL suit une philosophie de neutralité en ce qui concerne le modèle d'exécution, le modèle de communication inter-composants, le langage d'implantation, etc. En effet, FRACTAL laisse la définition de ce type d'aspects aux utilisateurs pour une meilleure adaptation aux contextes applicatifs. Parmi les types de spécialisation, nous pouvons citer les méta-niveaux implantés pour supporter par exemple des opérations de reconfiguration dynamique, ou les sémantiques de communications diverses et variées qui sont implantées sous forme de connecteurs de composants. Notons que le modèle de composition des composants supporte la composition hiérarchique et le partage de composants.

Enfin, FRACTAL est doté d'un ADL extensible pour refléter les extensions possibles du modèle. Cet ADL est supporté par une chaîne d'outils pour simplifier la programmation des composants. L'outillage du modèle FRACTAL se concentre plutôt sur le déploiement mais implante très peu de fonctions de vérification pour les architectures à déployer.

FRACTAL est aujourd'hui utilisé par une large communauté. Les applications vont de la mise en place des systèmes d'exploitation légers aux serveurs d'application administrables. De plus, plusieurs travaux ont montré que de nombreux modèles de composants peuvent êtres implantés comme étant des personnalités de FRACTAL. Dans ces termes, FRACTAL peut être considéré comme un méta-modèle de composants dont la spécification englobe la plupart des propriétés définies dans l'état de l'art des composants.

2.4 Synthèse

Nous avons présenté dans ce chapitre différentes propositions pour la programmation à base de composants qui ont été introduites dans la dernière décennie. Parmi ces travaux, certains, comme EJB, CCM, OpenCOM et FRACTAL, proposent des modèles de programmation pour le développement de systèmes à base de composants alors que d'autres, comme ArchJava et Classages, proposent des langages à base de composants.

L'analyse de ces différents travaux nous a permis d'identifier des concepts émergents dans le contexte decette nouvelle aire de la programmation. Ce sont :

1. **Interfaces multiples :** la possibilité d'implanter (ou de requérir) des services au travers de différentes interfaces d'un composant.
2. **Interface client :** élément d'identification des points où un composant peut émettre des opérations.
3. **Port :** élément d'identification des points d'interactions bidirectionnelles des composants. Ces interactions peuvent être sous forme d'émission ou de réception d'opérations ou de données.
4. **Connecteurs :** identification des éléments qui assurent la liaison entre les composants.
5. **Modèle de communication :** nous distinguons les modèles de composants qui fixent un modèle de communication pour l'interaction entre les composants, et ceux qui restent flexibles pour supporter différentes sémantiques d'interactions.
6. **Composition :** possibilité de composer des composants. La plupart des modèles supportent la composition dans des conteneurs alors que certains permettent la composition hiérarchique au sein des composants.
7. **Séparation des aspects fonctionnels et de contrôle :** alors que presque tous les modèles font plus ou moins une séparation des aspects de contrôle à l'aide des conteneurs, nous distinguons ceux qui

proposent une distinction de ces aspects au niveau composant.

8. Multi-langage : support d'implantation dans des langages différents.

9. Réflexion : implantation d'un méta-niveau au sein des composants pour proposer des opérations d'introspection et d'intercession.

10. Extensibilité : possibilité d'étendre le modèle (ou le langage) tout en restant compatible avec la spécification de base.

	JB	EJB	COM(+)	CCM	SystemC	OpenCOM	ArchJava	Classages	Fractal
1	-	-	x	x	-	x	-	-	x
2	-	-	-	-	-	x	-	-	x
3	-	-	-	x	x	-	x	x	-
4	-	-	-	-	x	x	x	x	x
5	Sync/ Evénement	Sync/ Evénement	Sync	Sync/ Evénement	Arbitraire	Synch	Arbitraire	Synch	Arbitraire
6	-	Conteneur	-	Hiérarch.	Hierarch	Hierarch	Hierarch	Hierarch	Hierarch
7	-	Conteneur	-	Conteneur	-	x	-	-	x
8	-	-	x	x	-	x	-	-	x
9	-	-	-	-	-	x	x	-	x
10	-	-	-	-	-	-	-	-	x

TAB. 2.1 – Tableau comparatif synthétisant les caractéristiques des modèles présentés. Les lignes correspondent aux concepts énumérés ci-dessus. (x) indique la présence d'un concept alors que (-) indique son absence.

Le tableau 2.1 récapitule les caractéristiques des travaux présentés sous les axes définis ci-dessus. Nous remarquons que la plupart des modèles supportent l'implantation des interfaces multiples, alors que l'utilisation des interfaces client, de ports et de connecteurs sont des concepts qui émergent dans l'ordre chronologique. C'est aussi vrai pour la composition hiérarchique qui est un concept puissant pour la mise en place de systèmes complexes. Nous remarquons que peu de modèles supportent intrinsèquement des modèles de communications flexibles. En effet, la plupart se basent sur la sémantique de communication fournie par le langage d'implantation sous-jacent. La séparation des aspects non-fonctionnels est un aspect émergeant dans les modèles comme EJB, CCM, COM+ et DCOM, mais il ne sont supportés au niveau composant que dans les modèles OpenCOM et FRACTAL. Enfin, le seul modèle extensible que nous avons présenté est FRACTAL, ce qui explique son utilisation dans des applications variées.

Nous tirons les conclusions suivantes de cette analyse d'état de l'art :

1. La programmation à base de composants propose de bonnes briques de bases pour aborder la programmation d'applications et de systèmes complexes. Néanmoins, il n'y a pas de modèle de composants idéal pour satisfaire l'ensemble des besoins. En d'autres termes, des propriétés qui semblent très intéressantes pour un certain cadre applicatif, comme le support de reconfiguration dynamique dans le cadre des intergiciels, peuvent constituer un inconvénient pour d'autres, comme dans le cadre des systèmes embarqués. Pour cette raison, le modèle doit être flexible pour être adapté à chaque contexte applicatif.
2. Les composants fournissent l'isolation des modules logiciels. Ceci est un atout fondamental pour

la réutilisation et pour la répartition des applications. Or, la plupart des approches restent au niveau du modèle de programmation et sont insuffisantes pour vérifier des propriétés d'isolation.

3. La notion d'architecture logicielle émerge avec la programmation à base de composants. Ceci permet d'une part de maîtriser la composition des logiciels et, d'autre part, de distinguer au moins deux métiers dans le processus de production de logiciels. Ce sont les programmeurs de composants qui implantent les fonctions métiers et les architectes qui composent des applications à partir des composants disponibles. L'existence de chaînes d'outils est cruciale pour le bon fonctionnement d'un tel processus. Dans ce cadre, un axe de recherche important consiste en la mise en place d'outils qui répondront à l'ensemble des besoins de génération de code, de compilation et de déploiement.

Chapitre 3

Conception de systèmes à l'aide de langages de description d'architecture

Sommaire

Un des apports principaux de la programmation à base de composants est de permettre la mise en place d'outils évolués pour faciliter le développement de logiciels et pour augmenter le niveau de maîtrise sur les systèmes logiciels complexes. Ces outils prennent souvent la forme de langages de configuration pour décrire, d'une manière abstraite, l'assemblage d'une application en associant aux éléments assemblés des contraintes et des caractéristiques afin d'assurer des propriétés globales.

Ce chapitre est consacré à la présentation de tels langages de configuration. Nous commençons par définir la notion d'architecture qui ressort dans la plupart des approches proposées. Pour ce faire, nous essayons de préciser ce que l'on appelle "architecture logicielle" et d'identifier ses éléments fondamentaux. Ensuite, nous présentons et analysons différents travaux qui s'intègrent dans le domaine étudié dans cet état de l'art. Ces travaux sont présentés selon quatre axes. D'abord, nous présentons deux exemples de langages de spécification formelle qui visent à vérifier des propriétés sur les assemblages de composants, à l'aide de modèles comportementaux. Par la suite, nous présentons deux exemples de langages de description de configurations logicielles permettant la construction de systèmes d'exploitation spécialisés. Ensuite, nous présentons deux exemples de langages de description d'architectures (ADL) qui

se focalisent sur le déploiement des architectures décrites. Ensuite, nous présentons des langages extensibles dont l'ambition est de fédérer les langages précédents. Nous présentons également les approches de conception de logiciel à l'aide des modèles et concluons le chapitre par une synthèse des travaux présentés.

3.1 Architecture logicielle et langages de description d'architecture

L'architecture est une notion très utilisée dans la plupart des domaines de l'ingénierie comme le génie civil ou encore le génie électrique. Dans ces domaines, elle correspond à l'art ou à la science de la construction d'édifices pour une utilisation quelconque par des humains [1]. Néanmoins, le mot architecture est de plus en plus utilisé dans le domaine de l'informatique, et ce à des fins diverses et variées. Pour recadrer ce que nous entendons par le concept d'architecture, nous trouvons utile de revenir à la définition concise, mais claire de [Ei90].

Architecture : *The organizational structure of a system or computer.*

Cette définition exprime clairement le fait que l'architecture logicielle concerne la structure organisationnelle d'une application logicielle. En effet, il est évident que toute application a une structure organisationnelle, implicite ou explicite. Or, pendant plusieurs décennies, de multiples applications ont été développées, sans parler explicitement de leur architecture. Que peut on alors attendre comme apport de ce concept qui ressort tardivement dans le processus de conception des systèmes informatiques ? Pour y répondre, nous reprenons la citation de Medvidovic [MT00] pour définir ce qu'est l'architecture logicielle.

> [Software architecture is a level of design that] *goes beyond the algorithms and data structures of the computation : designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure ; protocols for communication, synchronization, and data access ; assignment of functionality to design elements ; physical distribution ; composition of design elements ; scaling and performance ; and selection among design alternatives.*

Cette définition considère l'architecture comme un élément de base de la conception de logiciels. Elle prétend que la maîtrise architecturale d'un système informatique a des conséquences sur différents aspects rencontrés lors de la mise en place d'applications complexes. Ces aspects sont à la fois quantitatifs (e.g. passage à l'échelle, performance) et qualitatifs (e.g. exploration des choix de conception). En plus des éléments mentionnés dans la définition ci-dessus, il est admis que la connaissance architecturale d'un système peut faciliter le debogage, améliorer la réutilisation de code, permettre la spécialisation du logiciel et enfin simplifier la gestion et l'évolution d'un projet de développement de grande taille.

Les langages de description d'architectures constituent le moyen de décrire *une vue architecturale* d'un logiciel. Nous insistons sur le fait qu'il n'existe pas de consensus à ce jour sur *la vue architecturale* qu'il faudrait décrire. Ceci est du à l'existence de différentes préoccupations, en fonction des domaines applicatifs, parmi celles qui sont citées ci-dessus. Par exemple, dans le domaine des applications critiques, ces langages vont plutôt se focaliser sur les preuves de correction qui nécessitent la mise en œuvre de modèles comportementaux, alors que l'identification des contraintes environnementales suffirait pour assembler des systèmes d'exploitation ou intergiciels spécialisés. Remarquons qu'une description d'architecture n'est pas forcément une spécification statique ; elle peut aussi permettre de décrire des comportements dynamiques pour tenir compte des aspects évolutifs du logiciel au cours de son exécution. Les concepts suivants sont cependant communément acceptés par les architectes d'applications [MT00, Ei90].

[1] Traduction directe de la première définition citée dans *Oxford English Dictionnary, 1993 Edition*

Composants ce sont les unités de base d'un système. Ils représentent des élements d'encapsulation de données et de comportements.

Connecteurs ce sont les éléments qui réifient l'interaction entre les composants.

Configurations ce sont des descriptions d'architectures pour une une application complète ou partielle. Elles définissent des assemblages de composants interconnectés par l'intermédiaire de connecteurs.

Chaînes d'outils ce sont des logiciels qui accompagnent un langage de description d'architectures (ou de configuration). Parmi les fonctions implantées par les chaînes d'outils, citons la vérification, la simulation, l'aide à la conception, la génération de code, la compilation, le déploiement et la gestion dynamique à l'exécution.

3.2 Langages de spécification formelle d'architectures

3.2.1 Rapide

Rapide [LKA+95] est un langage de modélisation abstraite développé par une collaboration entre l'Université Stanford et la société TRW inc. dans la deuxième moitié des années 90. Sa préoccupation principale est de fournir un langage de prototypage qui permet de vérifier des propriétés de type synchronisation, concurrence et flots de données sur des assemblages de composants.

3.2.1.1 Caractéristiques

Rapide permet de décrire l'architecture d'un logiciel comme un assemblage de composants. Les composants sont les unités élémentaires du langage et encapsulent des comportements. Les interactions d'un composant avec son environnement se font par son interface. L'interface d'un composant expose quatre modes d'interactions :

- Les opérations *produites* sont les services implantés par le composant et s'exécutent de manière synchrone.
- Les opérations *requises* sont les services dont un composant a besoin. Ces opérations sont exécutées de manière synchrone.
- Les opérations de type *action entrante* modélisent les événements recevables par un composant.
- Les opérations de type *action sortante* modélisent les événements qu'un composant peut émettre.

Une des caractéristiques importantes de Rapide est le modèle d'échange d'événements qu'il définit [LV95]. En effet, Rapide propose un ordre partiel pour les ensembles d'événements échangés et fournit aux programmeurs un langage formel pour décrire les relations entre ceux-ci. Ce langage contient un opérateur de causalité ($\rightarrow$), un opérateur de simultanéité (&) et enfin un opérateur d'indépendance ($\|$).

Les descriptions d'architecture donnent lieu à la création d'un prototype exécutable sur le simulateur Rapide. Ceci permet de faire évoluer le système dans le temps et de vérifier si les contraintes spécifiées par les programmeurs sont respectées ou pas.

3.2.1.2 Mise en œuvre

Commençons par rappeler que Rapide est un langage de modélisation, et non un langage d'implantation. Pour cette raison, ce langage fournit des opérateurs et des structures de données simples et de haut niveau. Il est ainsi aisé de mettre en œuvre des modèles comportementaux sans prendre en compte les aspects liés à l'efficacité d'exécution.

La figure 3.1 illustre la description en Rapide d'un système de ping-pong. Le comportement d'un composant est décrit en utilisant la construction *type*. La description d'un type de composants se fait en deux parties : une partie interface qui décrit les opérations et les messages que le composant peut émettre et recevoir, et une partie comportement qui décrit les réactions du composant à la réception

```
type Ping (Max : Positive) is interface
  action out ping(N : Integer)
  action in  pong(N : Integer)
  behavior
    Start => ping(0);
    (?X in Integer) pong(?X) where ?X < Max => ping(?X + 1);
end Producer

type Pong is interface
  action out pong(N : Integer)
  action in  ping(N : Integer)
  behavior
    (?X in Integer) ping(?X) => pong(?X);
end Consumer

architecture PingPong() return SomeType is
  ping : Ping(100)
  pong : Pong
  connect
    (?n in Integer)
    ping.ping(?n) => pong.ping(?n);
    pong.pong(?n) => ping.pong(?n);
end architecture PingPong
```

FIG. 3.1 – Exemple d'une description de système Ping-Pong en Rapide.

des événements. Dans l'exemple présenté, deux types de composants sont définis. Le premier lance l'exécution en envoyant un message *ping* et continue par la suite à répondre à des événements *pong* en incrémentant à chaque fois la valeur entière du message jusqu'à ce que cette dernière atteigne un seuil maximal. Le composant *pong* implante un comportement symétrique à celui du composant *ping* sauf qu'il ne fait que répondre aux événements *ping* en envoyant un *pong*.

Ces deux composants sont instanciés au sein d'une architecture *PingPong*. La définition de l'architecture attribue une valeur initiale au seuil du composant ping, et connecte les interfaces *ping* et *pong* des deux composants de manière appropriée. L'exécution de cette architecture permet d'évaluer le comportement des deux composants dans le temps, et permet d'observer si des violations aux contraintes protocolaires surgissent au cours de l'exécution.

3.2.1.3 Evaluation

Rapide constitue un exemple intéressant de système visant à vérifier des propriétés comportementales et architecturales d'un logiciel complexe. Pour ce faire, Rapide propose un langage de haut niveau permettant de décrire des comportements et des assemblages de composants. Les descriptions faites dans ce langage peuvent être exécutées par le simulateur Rapide afin de vérifier la conformité du modèle aux contraintes spécifiées par les programmeurs. Nous pouvons néanmoins noter l'absence d'une suite d'outils accompagnant le langage Rapide qui permettrait d'aller au delà de la modélisation dans le processus de développement, c'est à dire l'implantation et le déploiement.

Remarquons que l'ADL de Rapide reste peu évolué pour décrire des structures organisationnelles. Par exemple, la composition hiérarchique et le partage de composants ne sont pas supportés par ce langage. Un autre point négatif est l'absence de modèle de connecteur. En effet, les interconnexions entre les composants se font par des déclarations simples, et les sémantiques de communication se limitent à celles qui sont définies au niveau des interfaces. Par conséquent, des sémantiques plus sophistiquées que celles qui sont implantées par défaut ne sont pas modélisables. En contrepartie, notons que Rapide

propose un des rares ADL qui permettent de décrire le comportement dynamique d'un système comme la création de composants et la modification d'interconnexions au cours de l'exécution.

3.2.2 Wright

Wright [AGD97, AG98] est une proposition ultérieure à Rapide pour la modélisation architecturale. Il a été proposé par l'Université Carnegie Mellon. Tout comme Rapide, Wright est un langage qui permet de prototyper des sytèmes complexes ; en revanche, contrairement à Rapide, il se concentre plus sur la vérification des aspects architecturaux comme la vérification de conformité des ports interconnectés et la détection d'interblocages au cours de l'exécution.

3.2.2.1 Caractéristiques

Comme dans la plupart des ADL, les composants dans Wright sont les entités élémentaires qui encapsulent des comportements. Les points d'accès d'un composant sont spécifiés à l'aide de ports faisant partie de son interface. Une des particularités des ports tels qu'ils sont définis dans Wright est qu'ils n'ont pas de sémantique associée de type opération ou message entrant ou sortant. Cette sémantique est en effet déduite du comportement du composant qui a pour rôle de définir les relations protocolaires entre les différents ports d'un composant. Les spécifications de comportement des composants sont spécifiées dans le langage CSP [Hoa78].

Une des caractéristiques intéressantes de Wright est la description du comportement des connecteurs. Les connecteurs sont les éléments d'interconnexion qui lient les ports des composants. Contrairement à Rapide qui ne fournit pas d'abstraction pour modéliser le comportement des connecteurs, Wright considère ces derniers comme des éléments de base de l'architecture et fournit des constructions adéquates pour décrire des comportements protocolaires sophistiqués.

Les assemblages de composants sont appelés des *configurations*. Une *configuration* est une description statique d'architecture qui spécifie les instances de composants présents et leurs interconnexions par le biais d'instances de connecteurs. Wright permet aussi de décrire l'évolution dynamique des systèmes. Ceci ce fait par l'intermédiaire des *configurateurs*. Un *configurateur* est une description de reconfiguration dynamique qui est déclenchée lorsque la présence d'une condition bien précise est observée. Un langage de description qui permet de décrire les conditions de réaction est fourni à cette fin [AGD97]. Les opérations de reconfiguration que l'on peut modéliser sont la création/destruction de composants et de connecteurs, ainsi que la modification des interconnexions.

Les configurations décrites dans l'ADL Wright sont exécutées à l'aide d'un simulateur. Les principaux types de vérifications effectuées sont la détection d'interblocages et la compatibilité des ports interconnectés. Remarquons que la spécification et la vérification des architectures évolutives sont des propriétés très intéressantes pour la modélisation des systèmes répartis complexes. Malheureusement, l'utilisation de Wright se limite à la modélisation, et ne permet pas d'aller vers des implantations concrètes.

3.2.2.2 Mise en œuvre

Une description statique d'architecture en Wright s'écrit à l'aide des trois types de constructions citées ci-dessous, les *component*, *connector* et *configuration*. La figure 3.2 illustre la description d'une application ping-pong.

La description d'un composant se fait en deux parties. Premièrement, une liste de *ports* définit les points d'interaction du composant en précisant une liste de messages qui peuvent circuler sur chacun d'eux. Deuxièmement, une partie *computation* définit les relations comportementales entre les ports du composant. Par exemple, la partie *computation* du composant *Pong* spécifie que le protocole implanté par ce composant consiste globalement à envoyer un message *pong* via le port p à la suite de la réception d'un message *ping* sur le port p. L'opérateur $\surd$ indique la terminaison d'une description protocolaire. Les

opérateurs $\sqcap$ et $\triangle$ signifient respectivement les rôles iniateur/terminateur et esclave de la communication effectuée par les ports.

Component Ping
  Port p = $\overline{ping} \rightarrow pong \rightarrow$ p $\sqcap \surd$
  Computation = internalCompute $\rightarrow \overline{p.ping} \rightarrow p.pong \rightarrow$ Computation $\sqcap \surd$

Component Pong
  Port p = $ping \rightarrow \overline{pong} \rightarrow$ p $\triangle \surd$
  Computation = $p.ping \rightarrow$ internalCompute $\rightarrow \overline{p.pong} \rightarrow$ Computation $\triangle \surd$

Connector Link
  Role i = $\overline{ping} \rightarrow pong \rightarrow$ p $\sqcap \surd$
  Role o = $ping \rightarrow \overline{pong} \rightarrow$ p $\triangle \surd$
  Glue = $i.ping \rightarrow \overline{o.ping} \rightarrow$ Glue $\triangle$
    $o.pong \rightarrow \overline{i.pong} \rightarrow$ Glue $\triangle \surd$

Configuration PingPong
  Instances
  ping : Ping ; pong : Pong ; link : Link ;
  Attachments
  ping.p as link.i ; pong.p as link.o ;
End Configuration

FIG. 3.2 – Exemple d'une description de système Ping-Pong en Wright.

La description d'un connecteur définit le protocole de transmission des messages qui circulent. Pour ce faire, des rôles sont attribués aux entrées/sorties du connecteur. Dans l'exemple du connecteur *Link*, les messages *ping* reçus par l'interface *i* sont transmis vers l'interface *o*, et les messages *pong* sont transmis symétriquement à ce dernier.

Enfin, la configuration *PingPong* décrit la structure architecturale de l'application. La partie instance d'une configuration définit les instances de composants et connecteurs qui sont présentes dans l'architecture. Les interconnexions entre composants se font dans la partie *attachement* où la présence des connecteurs est explicitement prise en compte. Nous remarquons que la syntaxe pour définir les liaisons entre les composants et les connecteurs est très similaire à celle du langage SystemC présenté dans la section 2.2.4.

3.2.2.3 Evaluation

Wright est un langage de description d'architectures qui permet d'associer des comportements protocolaires aux composants pour en déduire des propriétés globales à une configuration de logiciel. Le langage de description proposé est basé sur le calcul formel CSP. Tout comme Rapide, Wright est destiné à être utilisé à des fins de modélisation et de vérification du comportement dynamique des systèmes complexes. Comparé à Rapide, le langage Wright se focalise plus sur la vérification des assemblages et sur la détection d'interblocages. Dans ce cadre, les connecteurs jouent un rôle important : ils sont explicitement présents dans les descriptions et ils peuvent encapsuler des sémantiques arbitraires d'échange de données. Enfin, une construction qui permet de modéliser la création et destruction de composants ainsi que la modification des interconnexions est spécifiée afin de modéliser les actions de reconfigurations dynamiques.

La chaîne d'outils qui supporte le langage de description Wright ne permet malheureusement pas d'aller plus loin que la modélisation dans le processus de développement de logiciel. Enfin, l'absence de supports pour la composition hiérarchique et pour la modélisation des composants partagés constitue le

point faible de ce langage lorsqu'il est comparé à des ADL plus évolués.

3.3 Langages de description de configuration logicielle

3.3.1 Knit

Knit [RFS+00, Rei01] est un langage de description de configuration développé à l'Université d'Utah. Il est destiné à assembler des modules logiciels écrits en C. Nous analysons dans la suite les caractéristiques de ce langage, et celles de sa chaîne d'outils.

3.3.1.1 Caractéristiques

Knit est un langage d'assemblage pour lier des modules logiciels écrits en C. La motivation du projet est de mettre en place un éditeur de liens sophistiqué qui permet d'assembler et de lier des composants se trouvant dans des bibliothèques afin de construire des systèmes spécialisés, tout en évitant la modification des fichiers sources. Pour illustrer l'intérêt de l'approche, considérons le cas de deux pilotes logiciels, p_1 et p_2, qui utilisent un service de traçage au travers de la fonction printf. Dans le cas de la programmation C classique, si l'on veut faire en sorte que le traçage des deux pilotes soit assuré par deux serveurs différents, on doit implanter deux fonctions de traçages avec des noms différents, par exemple printf_traceur1 et printf_traceur2, et ensuite modifier le code des pilotes p_1 et p_2 pour qu'ils appellent ces fonctions respectives. La proposition de Knit est de décrire ce type d'association dans un langage de description d'architectures, et par ce biais, d'automatiser le renommage des symboles au niveau binaire, juste avant l'édition des liens.

Les composants sont appelés des *unités* (*units*) dans Knit. Notons qu'il s'agit de composants conceptuels qui ne sont pas présents à l'exécution. En effet, Knit est neutre vis-à-vis de la technologie d'implantation. Il est par exemple utilisé avec OSKit [FBB+97] qui est une technologie à base de composants COM pour concevoir des systèmes d'exploitation spécialisés.

Deux types d'*unités* sont distingués. Les *unités primitives* représentent des modules logiciels dont l'implantation est effectuée en C. Elles spécifient les *services exportés* et *importés* et les dépendances systèmes d'un module logiciel et indiquent l'ensemble des fichiers source qui sont utilisés pour implanter ce dernier. Les *unités composites* décrivent un ensemble d'*unités* qui sont intégrées dans une configuration logicielle et spécifient la manière dont leurs *services exportés* et *importés* sont liés.

Le support pour Knit est constitué d'un générateur de code et de pilotes de compilation. Le générateur de code est utilisé pour implanter certaines optimisations. Knit permet par exemple de donner dans une description d'architecture des morceaux de code source destinés à être tissés dans le code source d'implantation. À titre d'exemple, [ERFL01] présente comment Knit peut être utilisé dans le cadre de la programmation par aspects. Le générateur de pilotes de compilation génère essentiellement des *makefiles* pour assurer la compilation et l'édition de liens conformément aux descriptions d'architecture. De cette manière, Knit reste neutre vis-à-vis des outils de compilation, ce qui le rend compatible avec n'importe quel type de plate-forme matérielle.

3.3.1.2 Mise en œuvre

Pour illustrer la mise en œuvre du langage Knit, nous nous proposons d'écrire comme précédemment l'application composée de deux composants faisant des *pings-pongs*. Comme illustrée dans la figure 3.3, la description d'une configuration en Knit se fait par la définition des *unités* de manière hiérarchique. Le composant *Ping* implante la fonction *main* qui constitue le point d'entrée du programme que l'on veut mettre en œuvre. De même, il fournit la fonction *pong* qui pourrait être invoquée par une autre unité, et utilise les fonctions *printf* et *ping* dans son implantation. L'unité *Ping* a une description similaire mais utilise de manière symétrique les fonctions *ping* et *pong*. Quant à l'*unité composite PingPong*, elle

exporte la fonction main qui est implantée par le composant *Ping* et décrit les liaisons entre ses sous-composants de manière à lier correctement les services *ping* et *pong*.

```
unit Ping = {
  imports [ io: {printf}, ping ];
  exports [ main: {main}, pong ];
  depends { pong needs io; };
  files { "ping.c" };
}
unit Pong = {
  imports [ io: {printf}, pong ];
  exports [ ping ];
  depends { ping needs io; };
  files { "pong.c" };
}

unit PingPong = {
  imports [io: {printf}]
  exports [main: {main}]
  link {
    // exporter main,
    // utiliser io et Pong pour invoquer printf et ping
    [main] <- Ping <- [io, Pong]
    // utiliser io et Ping pour invoquer printf et pong
    []     <- Pong <- [io, Ping]
  }
}
```

FIG. 3.3 – Exemple d'une description de système Ping-Pong en Knit.

Les lecteurs intéressés peuvent se référer à la référence complète du langage [Rei01], ainsi qu'à un *tutorial* [Gro01].

3.3.1.3 Evaluation

Knit propose un langage de description d'architectures pour assembler des modules logiciels qui ne sont pas forcément écrits à l'aide d'un modèle de composants. Il illustre comment les dépendances et les services implantés dans des modules écrits en C peuvent être spécifiés, et comment ces informations peuvent être exploitées pour automatiser l'assemblage de ces modules. Ceci constitue un moyen pour la réutilisation des composants logiciels avec des caractéristiques proches des solutions à base d'objets. La différence est que dans Knit, les composants sont uniquement présents à la conception, et donc aucun surcoût lié à la modularité n'est payé lors de l'exécution.

Néanmoins, les services de génération de code proposés par les outils de Knit restent limités à la génération de pilotes de compilation et n'aborde pas les questions comme l'optimisation d'une architecture à base de composants ou l'adaptation des interfaces de composants hétérogènes. Par conséquent, Knit n'adresse pas la problématique de la mise en œuvre de systèmes répartis.

3.3.2 CDL / eCos

Component Definition Language (CDL) est un langage descriptif conçu pour construire des systèmes d'exploitation spécialisés en utilisant l'environnement eCos [eco]. Nous analysons dans la suite cette proposition industrielle qui vise à regrouper des composants systèmes tout en fournissant des outils de vérification de conflits et de contraintes fonctionnelles.

3.3.2.1 Caractéristiques

eCos est un environnement de conception de systèmess d'exploitation embarqué qui suit une approche à base de composants pour faciliter la mise en place des instances de systèmes spécialisés. Cet environnement est doté d'un langage de description d'assemblages de composants, intitulé CDL, et d'outils graphiques et textuels qui permettent de compiler des noyaux et des libraires en utilisant les descriptions CDL.

Un *composant*, dans CDL, est une unité qui implante une fonction donnée. Des exemples de composants peuvent être des ordonnanceurs, des pilotes de matériels, des gestionnaires de mémoires, etc. Les composants implantent des services et ont des contraintes d'environnement. Par exemple, un navigateur web écrit en plusieurs *threads* a besoin du service d'ordonnancement implanté par un ordonnanceur. Des *interfaces* peuvent être utilisées pour associer des noms symboliques aux services. Ces dernières sont entre autres utilisées pour déclarer des dépendances indépendamment d'un composant d'implantation. Par exemple, le service d'ordonnancement peut être implanté par deux composants différents, l'un qui implante un ordonnacement à base de priorités, l'autre qui l'implante une politique de tourniquet.

Les *composants* sont distribués sous forme de paquetages. Un paquetage peut regrouper un ou plusieurs composants, et contient des informations supplémentaires comme le numéro de version, ou un texte de description. Les *composants* (ou les *paquetages*) peuvent exposer des *options* pour permettre aux concepteurs de les spécialiser afin de les adapter aux contraintes environnementales. Des exemples d'*options* peuvent être le nombre de priorités disponibles dans un ordonnanceur, l'*endianness* de la plateforme, etc.

Le support du langage CDL permet non seulement de construire des noyaux à partir des composants présents dans une bibliothèque, mais fournit également un certain nombre de vérifications afin de détecter des problèmes de configuration. Parmi ces vérifications, citons la détection de services manquant dans la configuration ou la détection de conflits qui peuvent arriver, par exemple, en cas d'utilisation de plusieurs versions d'un même composant. Notons que ce type de fonctionnalités est crucial pour l'assemblage de systèmes complexes comme les systèmes d'exploitation.

Comme les descriptions en CDL sont assez longues et qu'il est difficile d'illustrer un exemple d'assemblage de système dans ce document, nous ne présenterons pas la mise en œuvre de ce langage. Nous invitons le lecteur intéressé à consulter le site de eCos [eco] où l'on peut trouver de nombreux exemples introductifs.

3.3.2.2 Evaluation

CDL constitue une approche très pragmatique pour construire des instances spécialisées de systèmes d'exploitation par assemblage de composants. Bien que les composants soient utilisés au niveau conceptuel pour identifier des modules configurables et des contraintes d'environnement, il ne sont pas présents à l'exécution. Par conséquent, eCos n'impose pas de surcoût par rapport aux approches monolithiques. En revanche, l'absence de composants à l'exécution supprime toute possibilité d'administration ou de reconfiguration à base de composants lors de l'exécution d'un système déployé. Contrairement à Knit, l'utilisation de CDL n'a aucun impact sur le modèle de programmation utilisé pour implanter les composants.

Bien que CDL propose l'assemblage de composants, il ne fournit pas de vue architecturale du système assemblé. Les composants sont à gros grain, la notion d'interface est quasi absente, et les connexions entre les composants ne sont pas capturées par le langage. Ces caractéristiques font de CDL un langage considérablement restreint par rapport à d'autres propositions présentées dans ce chapitre. Enfin, remarquons que le langage et les outils proposés par eCos sont propriétaires et non-extensibles, ce qui rend difficile leur adaptation à des besoins spécifiques.

3.4 Langages et environnements de déploiement

3.4.1 Darwin

Darwin est un langage de description d'architecture proposé par l'Imperial College au milieu des années 90. Nous jugeons sa présentation particulièrement intéressante dans cette état de l'art car il constitue un des rares exemples d'ADL permettant l'implantation des systèmes informatiques décrits et supportant la description d'aspects dynamiques.

3.4.1.1 Caractéristiques

Darwin [MDEK95] est un langage développé pour décrire l'organisation structurelle de systèmes répartis évoluant dans le temps. Il est fondé sur deux abstractions principales : les composants et les services. Les composants constituent les unités d'encapsulation de comportements et interagissent avec leur environnement via des services. Un service correspond à une opération (une méthode) implantée ou requise par le composant. Darwin distingue deux types de composants : les composants primitifs sont implantés à partir de code fonctionnel, alors que les composants composites sont implantés par la composition d'autres composants, primitifs ou composites.

L'exécution des systèmes décrits en Darwin nécessite un environnement d'exécution réparti. Regis [MDK94] constitue un environnement adéquat pour déployer et exécuter des architectures décrites en Darwin et implantées en C++. D'autres supports d'exécution peuvent être utilisés en utilisant des générateurs d'interfaces pour adapter le format d'appel des opérations à l'aide de *stubs* et de *skeletons*.

Une particularité intéressante de Darwin est sa capacité de décrire à la fois des architectures statiques et dynamiques. Une architecture statique est décrite par des instances de composants et leurs interconnexions. Elle peut être conçue de manière hiérarchique à l'aide des composants composites. Quant à la description des architectures dynamiques, différentes constructions sont proposées pour décrire des structures qui évoluent. Parmi elles, citons la construction *forall* qui permet de décrire des opérations itératives (par exemple des instanciations de composants) et *when* qui permet de détecter l'occurrence d'une condition recherchée pour activer une opération.

Les opérations de reconfiguration décrites en Darwin sont supportées par un environnement d'exécution, intitulé Conic [KM90]. L'environnement Conic implante en effet un algorithme de reconfiguration qui permet d'ajouter ou de retirer des composants, ainsi que de modifier des interconnexions en détectant un état stable du système. Pour ce faire, l'environnement supervise le cycle de vie de tous les composants présents dans le système, et capture des état gelés des composants pour les supprimer ou pour modifier une de leurs interconnexions. Ainsi, des architectures de systèmes répartis évoluant dans le temps peuvent être spécifiées en Darwin et être exécutées de manière adéquate.

3.4.1.2 Mise en œuvre

Darwin est un langage assez simple qui permet de décrire uniquement la structure organisationnelle d'une application en termes de composants et de services. La figure 3.4 illustre la mise en œuvre de l'application *ping-pong*. Les deux premières constructions de composants décrivent les composants primitifs *ping* et *pong* de l'application. Comme ces deux composants sont primitifs, ils sont uniquement décrits par le biais des services qu'ils fournissent et qu'ils requièrent.

Le composant *PingPong* consitue un exemple de composant composite qui n'a pas d'interaction avec son environnement. La description de ce composant commence par définir des tableaux d'identificateurs qui peuvent enregistrer jusqu'à n instances de composants *ping* et *pong*, n étant un paramètre qui permet de spécialiser ce composant. Remarquons que l'instanciation et l'interconnexion des composants sont faites plus tard au sein de la construction itérative *forall*. Étant donné que le composant *PingPong*

```
component Ping {
  provide reception_pong <int>
  require envoie_ping    <int>
}

component Pong {
  provide reception_ping <int>
  require envoie_pong    <int>
}

component PingPong (int n) {
  // Pas de service produit ou requis

  // Définitions pour n instances de couple Ping et Pong
  Array : ping[n] : Ping ;
  Array : pong[n] : Pong ;

  forall k : 0 .. n {
    Inst ping[k] ; // Création de k'ième instance de ping
    Inst pong[k] ; // Création de k'ième instance de pong
    Bind ping[k].envoie_ping - pong[k].reception_ping ;
    Bind pong[k].envoie_pong - ping[k].reception_pong ;
  }
}
```

FIG. 3.4 – Exemple d'une description de système Ping-Pong en Darwin.

ne spécifie aucune contrainte conditionnelle pour effectuer l'opération de déploiement[2] décrite dans la construction *forall*, cette configuration sera crée pour chaque instance de composant *PingPong* présent dans le système déployé.

3.4.1.3 Évaluation

Darwin est un exemple d'ADL qui permet de décrire l'architecture de systèmes répartis évoluant au cours de l'exécution. Ce langage supporte la composition hiérarchique et paramétrable, ce qui est un atout pour décrire le type de systèmes ciblés, mais l'absence du partage de composants constitue une limitation importante pour décrire des systèmes contenant des essources partagées. La reconfiguration dynamique des systèmes en termes d'ajout et de suppression de composants et de modification d'interconnexions est supportése par le langage, et un environnement d'exécution qui implante les opérations de reconfiguration adéquates est défini.

Un aspect qui nous semble manquer dans le langage Darwin est le fait que les services ne soient pas regroupés au sein d'interfaces typées qui permettraient d'effectuer des vérifications plus évoluées sur des assemblages de composants. En effet, le fait que le langage ne soit pas extensible limite ses évolutions dans les directions citées ci-dessus pour prendre en compte des aspects non prévus. Un exemple d'extension aurait pu être la spécification des aspects comportementaux des composants afin d'effectuer la détection d'interblocage comme c'est le cas dans Wright. Enfin, nous pouvons aussi noter le fait que l'outillage de Darwin soit limité au déploiement et qu'il ne fournisse pas d'outils de génération de code.

3.4.2 Olan

Olan est un environnement de déploiement développé par l'équipe SIRAC à l'INRIA Rhône-Alpes. Son étude nous semble intéressante car il propose un système de négociation de ressources qui met en relation la spécification des ressources matérielles et leur utilisation par les composants logiciels.

[2]Les conditions peuvent être spécifiées à l'aide de la construction *when*

3.4.2.1 Caractéristiques

L'environnement Olan [BBB+98, Bel, Bel97] est destiné au déploiement et à l'administration de systèmes répartis et potentiellement hétérogènes. Il est composé de deux éléments.

Le premier élément d'Olan est son langage de description d'architectures. Ce langage permet de décrire une architecture de systèmes en termes de regroupement de *composants* et de leurs interconnexions via des *connecteurs*. Olan distingue deux types de *composants*. Les *composants primitifs* représentent du code encapsulé et correspondent à des modules logiciels propriétaires. La spécification d'un tel composant décrit la nature du module (code source, bytecode, bibliothèque, binaire exécutable, etc.) et donne des indications sur le langage de programmation utilisé. Les *composants composites* sont les unités de structuration. D'une part, il représentent l'architecture des composants qu'ils encapsulent et, d'autre part, il donnent des indications sur les ressources dont ils ont besoin en termes de charge de CPU, espace mémoire, etc. Les *composants composites* peuvent être réutilisés dans d'autres composants. De cette manière, des représentations et spécifications hiérarchiques peuvent être obtenues.

Les connecteurs ne sont pas considérés comme des éléments de premier niveau dans le langage de description d'architecture d'Olan. Pourtant, les spécifications architecturales des composants permettent non seulement la mise en place des connexions avec des protocoles de communication arbitrairement complexes, mais aussi l'adaptation de formats de données échangées entre des composant écrits dans différents langages. De cette manière, l'hétérogénéité du système réparti est rendue en grande partie transparente aux composants propriétaires.

Le deuxième élément d'Olan est son environnement de support d'exécution. Cet environnement fournit des services d'installation, de déploiement et d'exécution de systèmes répartis à partir des descriptions d'architecture. Entre autres, cet environnement fournit un moyen de placer les composants logiciels dans des processus répartis à l'échelle d'un réseau d'ordinateurs. Pour ce faire, Olan propose d'utiliser des spécifications de ressources au niveau des composants qui les associent directement à des nœuds présents dans le système ou bien de trouver un scénario de placement qui satisfait au mieux les besoins spécifiés.

3.4.2.2 Mise en œuvre

Les composants, dans Olan, présentent une seule interface. Une interface peut regrouper les opérations fournies, et requises (notifiables) par le composant. La figure 3.5 illustre la description de l'architecture de l'application *PingPong* avec deux composants. Cette description commence par la définition des interfaces des deux composants *Ping* et *Pong*. La définition des composants primitifs, appelés *modules* dans Olan, se fait en identifiant le langage d'implantation du composant, l'interface qu'il implante et le nom et la localisation de son code source. Une fois que toutes les définitions de composants sont effectuées, on peut définir des composants composites, appelés *implementations* dans Olan. La description d'un composant composite indique les instances de composants qu'il encapsule et spécifie leurs liaisons.

Les descriptions de déploiement sont décrites dans des fichiers joints qui permettent d'identifier l'association des composants et les nœuds qui seront utilisés pour les exécuter. La figure 3.6 illustre la structure qui doit être remplie pour représenter chaque nœud présent dans le système réparti. Cette structure contient des informations sur le nom de la machine, sur le système d'exploitation et les librairies implantées et sur la charge d'utilisation. Finalement, les composants sont associés aux nœuds comme présenté en bas de la figure 3.6. Notons qu'un certain nombre d'expressions logiques peuvent être utilisées pour mettre en place des associations plus ou moins sophistiquées.

```
// Descriptions d'interface pour les deux composants.
Interface pingItf {
  provide pong() ;
  notify  ping() ;
}

Interface pongngItf {
  notify  pong() ;
  provide ping() ;
}

// Description des composants primitifs Ping et Pong
module "python" PingMod : pingItf {
  path: "${SRC}/PingPong"
  sourceFile: "ping.py"
}
...

// Description du composant composite PingPong
Implementation PingPong uses PingMod, PongMod {
  Ping = instance PingMod ;
  Pong = instance PongMod ;
  // Liaisons
  Ping.ping() => Pong.ping() ;
  Pong.pong() => Ping.pong() ;
  //
}
```

FIG. 3.5 – Exemple d'une description de sysème Ping-Pong en Olan.

```
management attribute Node {
  string name  ;      // Le nom de la machine
  string IPAdr ;      // L'adresse internet
  string platform ;   // Architecture de la plate-forme
  string os ;         // (posix, unix, nt, etc.)
  short osVersion ;   // Numéro de version
  long CPULoad ;      // charge moyenne des 10 dernières minutes
  long UserLoad ;     // nombre d'utilisateurs connectés
}

Management PingPongMgt: PingPong {
  // Les composants seront exécutés sur la machine suivante
  Node.name == "sirac.inrialpes.fr" ;
}
```

FIG. 3.6 – Descripteurs de déploiement pour l'application Ping-Pong en Olan.

3.4.2.3 Evaluation

Olan est une proposition très pragmatique dans le cadre de l'utilisation des langages de description d'architecture pour le déploiement des systèmes répartis. Il permet la réutilisation des composants propriétaires, potentiellement écrits dans des langages différents. Ainsi, il met en place automatiquement les connecteurs qui implantent l'adaptation de la communication entre les composants distants et/ou hétérogènes. Le langage de description d'architectures intègre les contraintes de déploiement et les caractéristiques des ressources matérielles présentes dans le système. L'ensemble de ces informations sont utilisées par l'environnement d'exécution proposé afin de déployer et de configurer le système, conformément aux spécifications effectuées dans un langage de haut niveau.

Olan propose un certain nombre de fonctionnalités intéressantes. Néanmoins, ni le langage d'entrée, ni l'environnement d'exécution ne sont prévus pour être étendus. Par conséquent, cette infrastructure n'est pas facilement utilisable par des concepteurs qui voudraient y intégrer de nouveaux aspects spécifiques à leur domaine d'utilisation. De plus, Olan propose un ADL propriétaire, ce qui rend difficile sa collaboration avec d'autres outils.

3.5 Langages extensibles

3.5.1 ACME et xACME

ACME [GMW97] est, non pas un langage de description d'architecture, mais un format pivot pour fédérer différents ADL. Tout comme Wright, il a été développé à l'Université Carnegie Mellon.

3.5.1.1 Caractéristiques

Le constat de base qui motive la définition d'ACME est qu'il existe de nombreux ADL qui ont chacun des caractéristiques diverses et variées. Les développeurs remarquent qu'une tentative de définition d'un nouveau langage qui regrouperait les caractéristiques des ADL existants ne serait pas une solution durable étant donné qu'il est très difficile de définir un langage implantant l'union des caractéristiques désirées. En revanche, ils constatent qu'une grande partie des ADL sont basés sur les mêmes concepts, surtout en ce qui concerne la définition des structures organisationnelles.

ACME définit un langage dont la base repose sur ces concepts partagés. On peut noter que les caractéristiques de base sont héritées de Wright : la composition hiérarchique est supportée et les connecteurs sont considérés comme des éléments de premier niveau dans la description d'architecture. De plus, des *templates* sont proposés afin de définir des systèmes (*configurations* dans le jargon Wright) réutilisables et paramétrables. La particularité du langage ACME est qu'il est doté d'annotations pour intégrer des expressions spécifiques à un ADL donné. Parmi les utilisations communes des annotations, citons l'expression des modèles comportementaux et des aspects non-fonctionnels comme des contraintes de temps réel ou de qualité de service. Remarquons que les annotations fournissent une forme d'extensibilité au langage de base. Ces aspects sont mieux formalisés dans xACME [xAC] qui propose l'utilisation de schémas XML pour décrire la grammaire et certaines propriétés des éléments d'extension qui seraient décrites sous forme d'annotations dans ACME.

Parce qu'ACME est un langage pivot pour fédérer divers langages, son outillage est censé fournir des traductions entre les langages supportés. Un exemple d'utilisation d'ACME est illustré dans [GMW97] où les auteurs présentent comment une description écrite en Wright peut être traduite en Rapide en passant par ACME.

3.5.1.2 Mise en œuvre

La figure 3.7 présente l'application *PingPong*. Les deux instances de *composants Ping* et *Pong* sont décrites séparément en précisant leurs *ports*. Remarquons qu'il n'y a pas de sémantique de type en-

trée/sortie ou fournis/requis associée aux *ports*. Une annotation est ajoutée au composant *Ping* pour déterminer un seuil à partir duquel il n'enverra plus de messages.

La description des *connecteurs* se fait similairement à celle des *composants*. À ce stade, on déclare les *rôles* qui correspondent aux points d'interaction des *connecteurs*. Nous illustrons dans la figure 3.7 l'association des propriétés protocolaires (*synchrone* ou *asynchrone*) aux *connecteurs* en utilisant des annotations.

```
System PingPong = {
  Component Ping = {
    port ping ;
    port pong ;
    Properties  = {max : int = 100 ;}
  }

  Component Pong = {
    port ping ;
    port pong ;
  }

  Connector BufferedLink = {
    Roles {i, o}
    Properties = {buffered : boolean = true ;}
  }

  Connector SingletonLink = {
    Roles {i, o}
    Properties = {buffered : boolean = false ;}
  }

  Attachments = {
    Ping.ping to Buffered.i ;
    Buffered.o to Pong.ping ;
    Pong.pong to SingletonLink.i ;
    SingletonLink.o to Ping.pong ;
  }
}
```

FIG. 3.7 – Exemple d'une description de sysème Ping-Pong en ACME.

Une fois la définition de toutes les instances de composants et de connecteurs terminée, il reste à définir dans la partie *attachment* leurs interconnexions. Notons qu'une description de système telle que présentée dans la figure 3.7 peut être réutilisée dans une autre description et peut contenir des *ports* pour définir ses points d'interactions avec son environnement.

3.5.1.3 Evaluation

ACME est, à notre connaissance, le premier travail qui constate que la diversité des ADL devient très importante et qu'il est nécessaire de chercher à fédérer les langages existants et leurs outillages associés. Dans ce contexte, il propose un langage pivot pour passer d'un ADL à un autre. Ce langage contient des constructions de base, inspirées de Wright, pour décrire l'organisation structurelle d'un système, et bénéficie d'annotations pour intégrer des informations secondaires.

Bien qu'au niveau des idées ACME apporte une contribution considérable, sa mise en œuvre reste malheureusement difficile à utiliser. En effet, aucun outillage n'est proposé avec le langage ACME. Par conséquent, pour chaque extension définie pour fédérer deux ADL, les concepteurs doivent écrire des traducteurs entre ces ADL et ACME. De plus, la traduction ne suffirait pas étant donné que les concepteurs doivent manipuler des fichier ACME étendus pour enrichir la représentation unifiée obtenue en ACME avant de traduire cette dernière dans le langage cible. Par exemple, si l'on veut fédérer Wright

et Rapide (dans le sens indiqué), on doit d'abord utiliser (après l'avoir écrit) un traducteur pour passer de Wright à ACME, ensuite modifier les fichiers intermédiaires pour y intégrer des informations de type Rapide et enfin traduire cette représentation unifiée en Rapide. Cette procédure de fédération d'un langage vers un autre étant déjà complexe, il nous semble que l'utilisation d'ACME sera peu efficace pour fédérer des descriptions effectuées dans de multiples langages.

3.5.2 xArch et xADL

xArch [DvdHT05] et son évolution xADL [DdHT01, DvdH01] sont deux exemples de langages extensibles qui ont pur but d'intégrer les constructions classiques des ADL et les informations spécifiques aux contextes d'utilisation en se basant sur la technologie XML. Ils sont issus de travaux de l'Université de Californie à Irvine et de l'Université Carnegie Mellon.

3.5.2.1 Caractéristiques

xArch est un des premiers efforts qui visent à proposer un langage ouvert et flexible pour remplacer de multiples ADLs propriétaires par un ADL extensible, afin d'intégrer plusieurs concepts. La démarche adoptée est proche de celle d'ACME : définir un langage avec des éléments de base pour décrire l'organisation structurelle d'une architecture à composants, et définir un moyen d'extension pour permettre aux utilisateurs d'enrichir le langage avec des informations propres à leur contexte d'utilisation. xArch se différencie d'ACME par l'utilisation de schémas XML pour décrire la grammaire des extensions apportées au langage de base.

Le langage xArch permet de décrire une instance de système à l'aide de *composants*, d'*interfaces*, de *connecteurs* et de *liens*. Notons que les *connecteurs* peuvent encapsuler des sémantiques de communication arbitrairement complexes et sont liés aux composants à l'aide de *liens*. Une architecture est décrite sous la forme d'un assemblage de ces quatre éléments. Elle peut être composée de plusieurs fichiers, chacun constituant une *sous-architecture*. Enfin, xArch définit une notion de *groupe* pour désigner un ensemble quelconque de *composants* ou de *connecteurs*. Un *groupe* peut par exemple être utilisé pour identifier les composants qui ont la même localisation dans le système réparti.

xADL est une évolution de xArch qui vise à décrire l'architecture des familles de produits, ce dernier étant dédié uniquement à la description des instances de systèmes. La notion de famille de produits fait référence à un ensemble de logiciels qui ont une représentation identique à quelques éléments près. À titre d'exemple, nous pouvons citer un logiciel qui a différentes versions, ou bien un logiciel d'aide à la conception qui intègre de multiples fonctionnalités optionnelles dans le cadre d'une même base architecturale.

xADL se base sur les schémas XML. Nous présentons ci-dessous les schémas d'extension définis pour xADL.

Structures et types Ce sont les éléments qui permettent de modéliser les unités de conception. Il s'agit par exemple des descriptions *templates* des architectures avec des *composants* et *connecteurs* abstraits. Ces descriptions sont destinées à être enrichies par des informations liées aux implantations et au déploiement des composants pour devenir une description d'instance à la xArch. Un système de type basé sur des signatures est aussi spécifié afin de capturer la compatibilité des assemblages de composants.

Groupes Ils représentent des regroupements conceptuels de *composants* et de *connecteurs*. À titre d'exemple, citons un *groupe* de composants écrits par le même auteur.

Implantation Elle représente les informations liées à la mise en œuvre des composants logiciels. Par exemple, un schéma d'implantation en Java est spécifié afin de lier les éléments structurels comme les *composants*, les *interfaces* et les *connecteurs* à leurs classes d'implantation en Java.

Options Elles représentent les éléments optionnels d'un assemblage de composants. Nous pouvons par exemple imaginer qu'un composant de *debuggage* est optionnel dans l'architecture d'un environnement d'aide au développement (IDE).

Variantes Elles représentent des points de variations qui peuvent être intégrés à une architecture. Il s'agit, plus précisément, d'énumérer différentes variantes pour implanter une sous-architecture afin de permettre au concepteur d'en choisir une. Citons par exemple différentes variations de l'interface graphique d'un logiciel multimédia.

Versions Elles représentent les différentes versions d'un même composant. Cet élément nous semble très utile lorsque la description d'une famille de produits est utilisée pour représenter l'architecture d'un logiciel dont l'implantation des composants évolue au cours du temps.

3.5.2.2 Mise en œuvre

Les langages xArch et XADL sont basés sur XML. Le choix de XML est justifié par son extensibilité et par les outils de support déjà présents et largement utilisés. À l'origine, ces deux langages étaient basés sur la technologie XML-DTD. Les schémas XML ont été adoptés à partir de xADL v2.0 pour ses nombreux avantages par rapport à XML-DTD. Parmi ceux qui semblent utiles dans le cadre de xADL, citons l'aspect hiérarchique des descriptions qui permet de définir des relations d'héritage entre les schémas, et le système de type intégré qui permet d'effectuer de nombreuses vérifications sur des documents XML. Les extensions apportées à ces deux langages se font sous forme de nouveaux schémas qui étendent la grammaire initiale pour y intégrer de nouvelles propriétés.

La mise en œuvre de xADL v.20 imposant trop de lignes de code, nous nous limitons dans la figure 3.8 à la présentation de l'application de *PingPong* en xArch. La description se fait par une liste de balises qui décrivent les *instances* présentes dans l'architecture de l'application. Les lignes 4 à 24 décrivent le composant *Ping*. Cette description comprend l'attribution d'un nom (lignes 5 à 7), et la définition des interfaces *pingItf* et *pongItf* (lignes 8 à 23). À chaque instance de composant, de connecteur ou d'interface est associé un identifiant global sous forme d'un numéro. La description du composant *Pong* n'est pas représentée dans la figure mais serait très similaire à celle du composant *Ping* en inversant le sens des interfaces.

La description d'un *connecteur* est illustrée aux lignes 28 à 40. La description des *connecteurs* est très similaire à celle des composants, mais leur rôle est différent. Conformément à notre exemple classique de *PingPong*, deux instances de connecteurs sont nécessaires ; nous n'en avons représenté qu'une seule à cause des limitations d'espace. Une fois toutes les instances de *composants* et de *connecteurs* définies, les liens doivent être mis en place pour décrire la manière dont ces derniers sont assemblés. Un exemple de lien est illustré aux lignes 42 à 50. La description du lien contient deux balises *point* qui définissent les points de connexions. Les identifiants des interfaces sont alors utilisés. Par exemple, le lien qui lie le composant *Ping* au connecteur *ConnecteurSimple* est lié via deux points : l'interface *pingItf* du composant *Ping* qui est identifiée par *2* et l'interface *entrée* du connecteur *ConnecteurSimple* qui est identifiée par *8*.

Comme nous pouvons le remarquer dans l'exemple, le langage xArch (et naturellement xADL) est très verbeux. Ceci est d'abord dû au fait que la syntaxe utilisée est XML, et ensuite au fait que les balises contiennent des informations répétitives. De plus, la gestion des identifiants globaux est une difficulté importante. Notons que ces langages ne sont destinés à être manipulés par des programmeurs, mais par des outils, graphiques ou autres. Les auteurs donnent quelques pistes sur les outillages existants dans [DdHT01]. Néanmoins, il n'y a, à notre connaissance, pas d'outil pour intégrer ces derniers dans un processus de développement complet.

```
<xArch>
 <instance:archInstance instance:id="0" xsi:type="instance:ArchInstance">
  <!— Définition du composant Ping —>
  <instance:componentInstance instance:id="1" xsi:type="instance:ComponentInstance">
   <instance:description xsi:type="instance:DescriptionInstance">
    name = Ping
   </instance:description>
   <instance:interfaceInstance instance:id="2" xsi:type="instance:InterfaceInstance">
    <instance:description xsi:type="instance:DescriptionInstance">
     name = pingItf
    </instance:description>
    <instance:direction in xsi:type="instance:DescriptionInstance">
     out
    <instance:direction/>
   </instance:interfaceInstance>
   <instance:interfaceInstance instance:id="3" xsi:type="instance:InterfaceInstance">
    <instance:description xsi:type="instance:DescriptionInstance">
     name = pongItf
    </instance:description>
    <instance:direction in xsi:type="instance:DescriptionInstance">
     in
    <instance:direction/>
   </instance:interfaceInstance>
  </instance:componentInstance>
  <!— Définiton du composant Pong —>
   Les identifiants attribués les interfaces sont pingItf:5 pongItf:6
  <!— Fin de la définition du composant Pong —>

  <!— Définition des connecteurs —>
  <instance:connectorInstance instance:id="7" xsi:type="instance:ConnectorInstance">
   <instance:description xsi:type="instance:DescriptionInstance">
    name = ConnecteurSimple
   </instance:description>
   <instance:interfaceInstance instance:id="8" xsi:type="instance:InterfaceInstance">
    <instance:description xsi:type="instance:DescriptionInstance">
     name = entrée
    </instance:description>
    <instance:direction in xsi:type="instance:DescriptionInstance">
     in
    <instance:direction/>
      ...
  </instance:connectorInstance>
  <!— Définition de les autres connecteurs —>
    ...
  <!— Définition des liens —>
  <instance:linkInstance instance:id="10" xsi:type="instance:LinkInstance">
   <instance:point xsi:type="instance:Point">
    <instance:anchorOnInterface xlink::href="#2"
   </instance:point>
   <instance:point xsi:type="instance:Point">
    <instance:anchorOnInterface xlink::href="#8"
   </instance:point>
  </instance:linkInstance>
  <!— Définition des autres liens —>
   ...
 </instance:archInstance>
</xArch>
```

FIG. 3.8 – Exemple d'une description de système Ping-Pong en xArch.

3.5.2.3 Evaluation

xArch part du même constat qu'ACME, c'est-à-dire le manque d'un environnement unificateur pour intégrer différentes préoccupations des ADL. Néanmoins, il propose une autre solution basée sur un nouvel ADL minimaliste et extensible. Cette proposition consiste en un langage basé sur XML qui regroupe des constructions de base pour décrire des assemblages de composants et suggère l'utilisation des schémas XML pour procéder à des extensions. xADL est une extension de xArch pour décrire l'architecture des familles de produits.

Comparé aux ADL présentés dans les sections précédentes, xArch ne définit pas de modèle de composants. De par sa définition, xArch est compatible avec d'autres modèles de composants et permet potentiellement d'en utiliser plusieurs au sein d'une même architecture. De plus, l'extensibilité du langage nous semble capable de répondre à tout type de préoccupations que l'on pourrait avoir dans un processus de conception à base de composants. En conséquence, xArch nous semble fournir un véritable environnement unificateur pour les ADL et modèles de composants existants.

Néanmoins, le support pour xArch reste *ad hoc*. En effet, aucune architecture d'outillage n'est définie pour supporter les extensions du langage. En conséquence, cette proposition ne va pas plus loin qu'ACME en dehors de l'utilisation des schémas XML pour aider les concepteurs à vérifier certaines propriétés des descriptions. De plus, xArch propose d'unifier tous types de préoccupations dans un même langage à base de XML. Cette obligation de décrire dans un seul langage des architectures ayant différentes préoccupations nous semble une limitation importante ; nous pensons que l'utilisation de langages spécifiques aux domaines (DSL) constitue une solution bien plus confortable pour exprimer certaines informations de type protocolaire ou comportementale.

3.6 Approches basées sur des modèles

Bien qu'il ne soit en relation directe avec la conception de systèmes à partir de descriptions d'architecture, nous dédions cette section à une présentation brève des approches de conception de systèmes logiciels à l'aide des modèles pour enrichir cet état de l'art avec des références aux travaux connexes. Il s'agit des approches d'ingénierie dirigées par des modèles [Sch06] (MDE pour *Model Driven Engineering*) ou d'architectures dirigées par des modèles [BG01] (MDA pour *Model Driven Architecture*)[3].

3.6.1 Vision de l'Object Management Group

L'auteur de plusieurs standards connus comme CORBA [OMG01b] et UML [UML04], l'OMG a défini en 2001, une nouvelle approche pour la conception des systèmes. Cette approche est appelé l'architecture dirigée par des modèles ou MDA [MDA]. Au cœur de cette approche se trouve l'idée de faire de la conception, qui est plutôt perçu comme un processus de développement, le produit centrale du développement de logiciel [MW]. Ainsi, une fois le modèle d'un logiciel à mettre en œuvre est spécifié par les concepteurs, des outils de support de type CASE[4] pourront êtres utilisées pour analyser, vérifier, déployer et tester le modèle spécifié sur une plate-forme d'implantation donnée.

L'approche MDA favorise la séparation des préoccupations dans le sens de la définition de différentes vues abstraites d'un même système logiciel afin de permettre aux concepteurs de se concentrer uniquement sur les aspects qui les intéressent. Les deux grandes classes de modèles sont définies dans cette objectif sont :

– **les modèles indépendants de plate-forme** (PIM pour *Platform Independent Model*) qui sont utilisés pour modéliser de manière non-ambigue et sans détails liés à la plate-forme d'implantation

[3]MDA est une marque déposée par l'*Object Management Group* [dmddl] alors que l'acronyme MDE ne l'est pas. Par conséquent, ce dernier est utilisé par une communauté plus large de recherche.

[4]CASE pour *Computer Aided Software Engineering*.

les fonctions métier d'un système logiciel.
– et **les modèles spécifiques aux plates-formes** (PSM pour *Platform Specific Mode*) qui correspondent à l'application d'un modèle PIM à des plates-formes d'implantation données (e.g. CORBA, EJB, J2EE, etc.).

L'abstraction des modèles misent en œuvre par les concepteurs des détailles liées à des plates-formes d'implantation est motivé par la réutilisation d'une même conception de système (i.e. d'un PIM) pour l'implanter sur des plates-formes différentes (i.e. plusieurs PSM). Idéalement, MDA suggère la mise en œuvre des modèles exécutables et des spécifications détaillées des plates-formes d'exécution de manière à pouvoir automatiser la migration de ces derniers à l'aide des outils de type CASE. Pour ce faire, différents moyens de modélisation et de méta-modélisation sont spécifiés, parmi lesquelles nous présentons par la suite UML, MOF et XMI.

Unified Modeling Language (UML) [UML04] constitue un formalisme de référence pour la modélisation des systèmes logiciels. UML est originellement conçu par l'OMG pour la modélisation des programmes à base d'objets mais intègre depuis sa version 2.0 [UML04] des extensions permettant la modélisation des logiciels à base de composants. La spécification d'UML propose un formalisme très riche couvrant un large spectre de préoccupations, allant de la qualité de service au temps-réel. Il permet de même la spécification des modèles exécutables, c'est-à-dire des modèles portant une sémantique suffisamment détaillée pour être exécuté sur une machine abstraite [RFBLO01]. L'approche MDA propose d'utiliser UML comme le langage de spécification des modèles PIM et PSM.

Meta Object Facility (MOF) [CDI+97, MOF00] est un outil de méta-modélisation défini par l'OMG dans le but de standardiser un langage abstrait pour la spécification des langages de modélisation tels que UML, ou CWM [CWMa, CWMb]. L'apport principale de MOF se trouve dans son accentuation des concepts manipulés de manière à permettre l'utilisation de plusieurs syntaxes pour écrire des méta-modèles. MOF peut ainsi être utilisé comme un environnement pivot pour supporter l'interopérabilité ou la translation entre des modèles écrits dans différentes langages de modélisation.

XML Metadata Interchange (XMI) [XMI02] constitue l'application du standard MOF sur la syntaxe XML. Est décrite dans le standard XMI, la manière dont des méta-modèles écrits en MOF peuvent être transformés en des DTD, pour permettre aux concepteurs (ou aux outils) de mettre en œuvre des descriptions MOF en XML. Ainsi, les méta-modèles MOF peuvent être diffusés sous une forme sérialisée, pour assurer l'interopérabilité entre des systèmes répartis hétérogènes basés sur des méta-modèles distincts.

De nombreux outils académiques et commerciaux sont actuellement disponibles pour supporter le processus de développement à la MDA. Parmi ces outils, nous pouvons citer des outils d'édition et de manipulation de modèles (e.g. NetBeans Metadata Rhapsody [Rha], ModFact [Mod], Eclipse Modeling Framework [EMF]), des outils de génération de code (e.g. Rational Rose [Rat], EclipseUML [Ecl], Poseidon [Pos] et ArgoUML [Arg]), des outils de vérification basés sur des contraintes [Ham06] et des outils de translation de modèles (e.g. UMT-QVT [UMT], ArcStyler [Arc]). Enfin, l'ensemble des outils suscités répondant uniquement à une phase donnée du développement de logiciel, certains outils comme ModelBus [BGS04] et FrameKit [Frac] proposent des environnements d'intégration dans l'objectif de permettre aux concepteurs de composer des chaînes de traitements dédiées en fonction de leurs besoins.

3.6.2 MDE/MDA et ADL

La conception de systèmes à l'aide des ADL peut être considéré comme une approche de conception à la MDA où les structures des systèmes logiciels sont modélisés sous la forme de descriptions d'architecture. Comme expliqué dans les sections précédentes, ces modèles sont utilisés à des fins de vérification, de génération de code où de déploiement. Il serait alors judicieux de constater que la conception de systèmes à l'aide des ADL s'intègre tout à fait dans la vision MDA présentée par l'OMG, mais se restreint

dans le cadre de la structuration des programmes à base de composants.

Enfin, notons que les travaux présentés dans [Mar02] soulignent l'intérêt de conjuguer ces deux approches, et illustrent comment des outils MDA peuvent être utilisés pour la méta-modélisation des langages de description d'architectures afin d'assurer l'intéropérabilité entre des descriptions écrites en des ADL différents.

3.7 Synthèse

Nous avons présenté et analysé dans ce chapitre divers langages de description d'architectures qui ont été proposés dans les milieux académiques et industriels. Nous avons regroupé ces travaux sous l'angle de leurs contextes d'utilisation. Sans avoir cherché à être exhaustif, nous avons présenté différents axes d'utilisation : la modélisation et la validation de systèmes complexes, l'assemblage de composants à l'aide de générateurs de code pour créer des configurations logicielles spécialisées et, enfin, le déploiement et l'administration de systèmes répartis.

La synthèse des caractéristiques des travaux présentés illustre la diversité des préoccupations liées à l'architecture logicielle et nous permet de dégager des éléments de considérations différents en fonction des contextes d'utilisations.

- Les environnements de validation comme Rapide et Wright utilisent les composants comme des éléments d'encapsulation de comportements et se préoccupent de fournir des éléments de modélisations protocolaire ou comportementale. Dans Wright, les connecteurs sont reconnus comme des éléments architecturaux de premier niveau qui peuvent modéliser des sémantiques de communication complexes. La sémantique associée aux interfaces est soit faible, soit concentrée sur le sens des données échangées (entrée/sortie). Enfin, notons que Rapide et Wright définissent leur propre modèle de composants. En conséquence, ils ne peuvent pas directement être utilisés pour modéliser des architectures réalisées à l'aide d'autres modèles.
- Les environnements de déploiement ont des considérations architecturales qui sont proches de celles des environnements de validation, mais ils se concentrent sur la description de l'organisation structurelle des systèmes répartis. Les composants sont des unités d'encapsulation de comportements et de données qui interagissent uniquement par le biais de leurs interfaces. En revanche, il n'existe quasiment pas de moyen pour spécifier le comportement des composants. La sémantique associée aux interfaces est plutôt fonction du sens d'invocation (*client/serveur* ou *exporté/importé*). Les connecteurs sont en général des éléments de premier niveau qui sont utilisés pour implanter des protocoles de communication complexes correspondant aux besoins spécifiques des systèmes répartis. Dans ce cadre, certains travaux comme Olan, ou CIDL de Corba présentés dans le chapitre précédent fournissent la génération automatique des adaptateurs de communication entre des composants hétérogènes. Même si certains travaux proposent des solutions, la prise en compte de l'évolution dynamique de l'architecture doit être renforcée. Enfin, les outils sont en général associés à des modèles de programmation qui doivent être respectés lors de l'implantation des composants.
- Les environnements de configuration se préoccupent de réutiliser des modules logiciels afin d'assembler des systèmes spécialisés. Les exemples comme Knit de OSKit ou CDL de eCos illustrent bien le fait que l'architecture logicielle est une préoccupation secondaire. Le but principal est de donner suffisamment d'informations de haut niveau qui permettent d'assembler des modules en satisfaisant leurs dépendances. Les composants sont des unités d'encapsulation à gros grain qui remplissent des fonctions bien identifiées. Les notions d'interface et de connecteurs ne sont quasiment pas présentes. Enfin dans certains cas, comme dans Knit, le langage de description prend réellement la forme d'un éditeur de liens sophistiqué qui assure l'assemblage des composants écrits en langage C.

Certains travaux comme ACME ou xADL constatent la diversité des langages de description d'architecture et de leurs préoccupations, ce qui les amènent à proposer des solutions fédératrices ou intégratrices. Les deux argumentent sur le fait qu'un langage unificateur est impossible à mettre en œuvre car même si on arrivait à prendre en compte l'ensemble des préoccupations mentionnées ci-dessus, il y aurait toujours de nouvelles préoccupations à prendre en considération dans le futur. Pour cette raison, ils proposent des langages d'« intersection » extensibles. Ceux-ci fixent la représentation d'un minimum d'éléments se trouvant en commun dans la plupart des langages, et laissent les concepteurs étendre le langage pour introduire des concepts spécifiques à leurs besoins. Néanmoins, ces propositions ne fournissent pas de solution pour le support des langages définis. Ils échouent en particulier sur l'identification des éléments nécessaires au sein des outils associés pour la prise en compte des extensions introduites au niveau langage.

Notre analyse nous permet de faire les constats suivants.

- Les langages de description d'architectures prennent en considération de multiples éléments qui comprennent des éléments de structure organisationnelle (composants, interfaces, liaisons), et des éléments d'implantation (comportements, protocoles, déploiement, compilations, etc.). Il est en effet **impossible de définir un langage standard, ou unificateur qui prendrait en compte l'ensemble de ces éléments**. Dans ce cadre, les langages extensibles constituent une piste prometteuse. Néanmoins, nous ne croyons pas qu'un seul langage extensible puisse satisfaire les concepteurs pour décrire l'ensemble des aspects qui les préoccupent. Nous pensons que l'utilisation des **langages spécifiques aux domaines (DSL) permettra d'exprimer de manière modulaire et confortable les différents éléments des architectures logicielles**, que nous qualifions dorénavant d'**hétérogènes**.
- Chacun des travaux présentés remplit de manière isolée une fonction intéressante qui pourrait être intégrée dans un processus de développement complet. **Les propositions de fédération restent au niveau de langages extensibles** et ne spécifient pas comment les extensions pourraient être prises en compte pour concevoir des outils intégrés. Dans ce cadre, **il y a un besoin crucial en termes d'outils extensibles** qui pourraient intégrer l'ensemble des informations architecturales à supporter, et ce de manière évolutive en prenant en compte les extensions apportées au niveau du langage.
- Enfin, il existe de multiples domaines de l'ingénierie logicielle, allant des systèmes embarqués aux serveurs d'entreprises autonomes, qui bénéficient du développement basé sur l'architecture. Ceci crée une hétérogénéité en ce qui concerne les résultats attendus des outils. Citons par exemple la génération de code dans des langages de programmation variés ou le déploiement sur des plates-formes d'exécution hétérogènes. Pour cette raison, **un support idéal devrait produire plusieurs types de résultats, et en particulier devrait être « reciblable » , afin d'être adapté à des contextes d'utilisation différents**.

Chapitre 4

Le modèle de composants FRACTAL et les outils associés

Sommaire

Après l'analyse de l'état de l'art des modèles de composants et des environnements de programmation basés sur la description d'architecture, nous allons nous concentrer plus particulièrement sur le modèle FRACTAL. Ce modèle fournit en effet de nombreuses caractéristiques intéressantes par rapport aux autres environnements de programmation à base de composants. Parmi ces caractéristiques, la généricité et l'extensibilité du modèle et de son ADL, qui sont destinés à être adaptés en fonction du contexte d'utilisation, constituent un atout majeur pour la mise en place d'une structure commune à des projets de développement qui doivent respecter des contraintes variées.

Ce chapitre introductif commence par la présentation du modèle de composants et de son langage de description d'architecture. Ensuite, nous nous intéressons à l'aspect multi-langage du modèle qui permet des implantations dans des langages de programmations différents. Dans ce cadre, nous présentons deux implantations de FRACTAL, THINK et JULIA, qui fournissent des environnements de programmation en Java et en C, respectivement.

4.1 Le modèle de composants FRACTAL

FRACTAL est un modèle de composants dédié à la mise en place de systèmes logiciels complexes qui couvrent une large échelle d'application. Parmi ces applications, citons le déploiement autonome de serveurs d'applications [BBH+05], la mise en place de systèmes d'exploitation embarqués [FSLM02], la conception de systèmes dynamiquement reconfigurables [POS06, DL03], la mise en œuvre d'intergiciels adaptables [LQS05], l'implantation de systèmes multimédia [LH05] ou encore de canevas pour la gestion de qualité de service [TBO05]. La motivation principale du modèle est de faciliter la conception, l'implantation, le déploiement et l'administrations de tels systèmes. Les principales caractéristiques qui le distinguent des autres modèles de composants sont :

- les **composants composites** qui permettent d'avoir une vue uniforme des applications à des niveaux arbitraires ;
- les **composants partagés** qui fournissent une abstraction adéquate pour la représentation des ressources partagées telles que des gestionnaires de mémoire ou des pilotes de réseau ;
- la **capacité d'introspection** qui permet de se renseigner sur un système et de le contrôler lors de son exécution ;
- la **capacité de reconfiguration dynamique** qui permet de modifier le comportement d'un système lors de son exécution en installant ou en supprimant des composants ou en modifiant des interconnexions ;
- la **capacité d'extension** qui permet d'ajuster le modèle en fonction des contextes spécifiques d'application, par exemple en modifiant le niveau de contrôle supporté ;
- et le **canevas de liaison flexible** qui permet d'encapsuler des protocoles de communication arbitrairement complexes au sein de liaisons qui sont réifiées sous forme de composants.

Nous présentons par la suite les principaux éléments du modèle FRACTAL et détaillons les caractéristiques mentionnées ci-dessus.

4.1.1 Composants, composition hiérarchique et partage

Dans FRACTAL, les *composants* réifient les unités de conception, de composition et de configuration. Ils représentent ainsi des unités à l'exécution qui assurent l'encapsulation de comportements et de données.

Les *composants* interagissent avec leur environnement au travers de leurs *interfaces*. Les *interfaces* réifient des points d'accès typés qui regroupent des collections d'opérations qui peuvent être échangées entre le composant et son environnement. Le sens de l'échange de ces opérations donne lieu à la distinction de deux rôles concernant les *interfaces* : les *interfaces client* permettent d'appeler des opérations de l'environnement du composant alors que les *interfaces serveur* permettent d'en recevoir.

Comme illustré dans la figure 4.1, FRACTAL distingue deux types de composants : les *composants primitifs* sont des boîtes noires qui encapsulent des comportements potentiellement programmés dans un langage de programmation quelconque alors que les *composants composites* ont une architecture interne constituée par la composition de sous-composants, *primitifs* ou *composites*. Ceci constitue une architecture de composition hiérarchique et récursive s'arrêtant au niveau des composants primitifs et dans laquelle il existe toujours un composant de plus haut niveau qui encapsule le reste. Remarquons que les *composants composites* permettent entre autres de réifier des comportements et des structures internes arbitrairement complexes, ce qui permet de modéliser des niveaux d'abstraction arbitraires.

Enfin, en FRACTAL, une instance de composant peut être le sous-composant de plusieurs composants composites. Ceci correspond au partage de ce composant par ses composites englobants. Le partage de composant fournit une représentation adéquate pour modéliser des ressources partagées que l'on rencontre souvent dans le contexte des systèmes. Des exemples de telles ressources pourraient être des

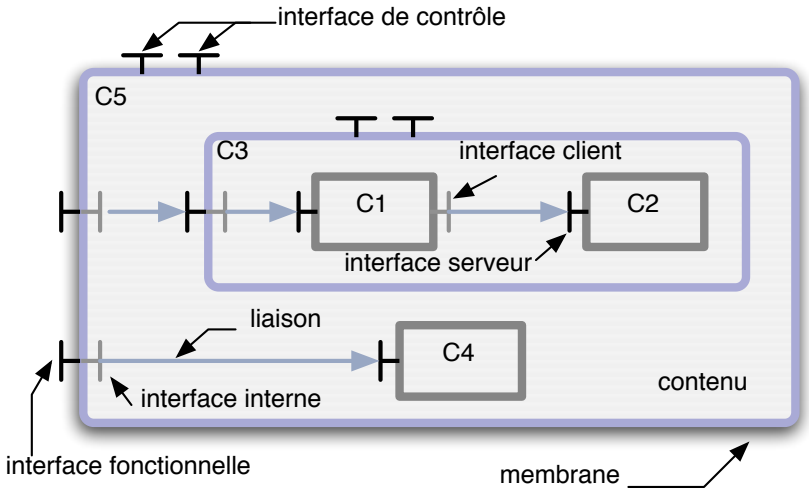


FIG. 4.1 – Un exemple de composant FRACTAL.

gestionnaires de mémoire et d'interruptions dans le cadre des systèmes d'exploitation, des gestionnaires de messages dans le cadre des intergiciels et des bases de données dans le cas des serveurs d'entreprises.

4.1.2 Séparation des préoccupations

FRACTAL supporte la séparation des préoccupations en séparant les parties des composants qui implantent les aspects fonctionnels et les aspects non-fonctionnels. Comme le montre la figure 4.1, ces deux parties sont le *contenu*, qui implante les fonctions métiers du composant sous forme de code fonctionnel ou d'un ensemble fini de sous-composants, et la *membrane* qui implante les opérations de contrôle du composant.

La *membrane* peut être considérée comme une épaisseur qui encapsule le *contenu* du composant. Elle peut avoir des *interfaces externes* et *internes*. Les *interfaces externes* sont accessibles par l'environnement du composant alors que les *interfaces internes* sont accessibles par le contenu. Les interfaces de la *membrane* sont implantées par un ensemble de *contrôleurs* qui peuvent être considérés comme des méta-objets.

Le modèle FRACTAL n'impose aucun méta-protocole qui devrait être implanté par les composants. En effet, il laisse les concepteurs décider du méta-protocole à réaliser en fonction des besoins spécifiques liées au contexte applicatif. Par conséquent, une *membrane* peut être vide (c'est à dire que les composants sont vus comme des boîtes noires sans comportement de contrôle), ou peut contenir un ensemble arbitraire de contrôleurs (par exemple pour l'introspection de sa structure interne).

Bien que leurs utilisation soit optionnelle, FRACTAL spécifie un ensemble de contrôleurs qui fournissent des opérations minimales d'introspection et de gestion de configuration et de vie. Ce sont :

- le **contrôleur d'identité** similaire à l'interface *IUnknown* du modèle COM [Box98] qui fournit des opérations d'introspection sur le nom, le type et les interfaces du composant,
- le **contrôleur d'attributs** pour configurer les attributs du composant,
- le **contrôleur de liaisons** pour établir/rompre des liaisons entre le composant et son environnement,
- le **contrôleur de cycle de vie** pour contrôler les phases comportementales du composant comme le démarrage et l'arrêt,
- et enfin le **contrôleur de contenu** pour accéder à la structure interne du composant ainsi que pour

ajouter ou supprimer des sous-composants.

4.1.3 Liaisons flexibles

Le modèle FRACTAL impose que les interfaces des composants soient liées pour établir des interactions. Ceci fait des *liaisons* des éléments architecturaux du premier niveau au même titre que les composants. Contrairement à la plupart des modèles de composants, FRACTAL implante n'importe quelle sémantique de communication entre les composants au travers des deux types de liaisons présentées ci-dessous.

- Les **liaisons simples** (4.2.a) implantent la sémantique de communication imposée par le langage dans lequel les composants sont implantés. Par exemple, si les composants sont implantés en C ou en Java, les liaisons réifieront l'appel synchrone de méthode.
- Les **liaisons complexes** (4.2.b) réifient des protocoles arbitraires de communication. Une *liaison complexe* est implantée par un composant (dit *composant de communication*) dont les interfaces sont liées aux deux composants interconnectés par des **liaisons simples** et dont la fonction est d'implanter la communication avec une sémantique donnée.

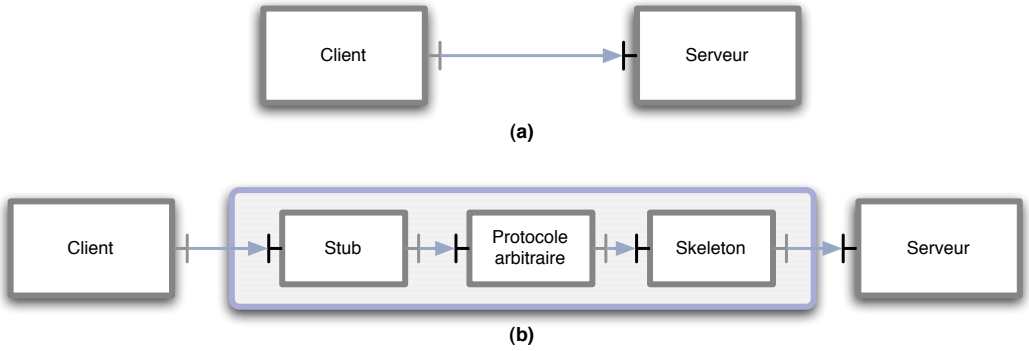


FIG. 4.2 – Liaisons simple (a) et complexe (b).

4.1.4 Système de types

Le modèle FRACTAL définit un système de types pour les composants et leurs interfaces. Le type d'un composant correspond à l'union de ses *interfaces client* et *serveur*. Le type d'une interface est constitué des éléments suivants :

- la **signature** qui détermine un ensemble fini d'opérations qui peuvent être appelées au travers de l'interface, ainsi que leurs paramètres,
- le **rôle** qui détermine la modalité d'appel de l'interface. Le rôle d'une interface peut être soit *client*, soir *serveur*,
- la **cardinalité** d'une interface qui détermine le nombre de connexions qui peuvent être acceptées ; la cardinalité peut être soit *singleton* (une seule connexion), soit *collection* (multiples connexions),
- et enfin la **contingence** qui détermine si les opérations de l'interface doivent être obligatoirement présentes à l'exécution ou pas. Si une interface est *obligatoire*, les opérations doivent êtres implantées. L'interface est dite *optionnelle* sinon.

Comme la prise en charge du système de types peut s'avérer coûteuse, par exemple dans le cas de la construction de systèmes embarqués, elle est facultative dans FRACTAL. Le système de types est particulièrement utile dans le cas des systèmes dynamiquement reconfigurables pour permettre de procéder à des vérifications d'équivalence lors de la modification des liaisons ou de la substitution de composants.

Notons que le système de types de FRACTAL implante aussi des relations de sous-typage pour améliorer la vérification des contraintes que doivent respecter les connexions d'interfaces et les substitutions de composants.

4.2 FRACTAL ADL : le langage de description d'architecture de FRACTAL

Le modèle de composants FRACTAL est accompagné d'un langage de description d'architecture, intitulé FRACTAL ADL. Ce langage s'inscrit dans la classe des ADL extensibles. Par la suite, nous présentons d'abord la base du langage FRACTAL ADL, puis ses capacités d'extension. Nous terminons cette section en présentant l'architecture de l'usine de déploiement qui réalise cet ADL dans le cadre du canevas JULIA.

4.2.1 Le langage FRACTAL ADL

Le langage FRACTAL ADL est un langage déclaratif de haut niveau qui permet de décrire des architectures logicielles à base de composants FRACTAL. La base du langage est minimale : contrairement aux autres ADL qui fixent un ensemble de propriétés qui doivent être spécifiées, FRACTAL ADL fournit uniquement des constructions de base pour énumérer des composants, des interfaces, des liaisons et des attributs et laisse les concepteurs étendre le langage pour intégrer d'autres informations spécifiques à leur cadre d'utilisation.

La syntaxe du langage FRACTAL ADL est basée sur XML. La figure 4.3 illustre un exemple de description d'architecture pour une application simple contenant un composant client qui appelle des opérations sur un composant serveur. Chaque fichier ADL contient la description d'un composant (*primitif* ou *composite*) et commence par un élément XML intitulé `definition`. Cet élément a un attribut obligatoire, appelé `name`, qui précise le nom du composant qui est décrit dans cette définition.

Une définition peut contenir un ensemble de sous éléments. Dans le cas de la définition d'un *composant primitif*, cet élément de base peut contenir un ensemble d'interfaces, *client* et *serveur*, la description et potentiellement les valeurs initiales des attributs du composant, une référence au fichier d'implantation du contenu et une référence à un descripteur de membrane. Dans le cas d'un *composant composite*, le contenu est plutôt implanté par un ensemble de sous-composants. La définition peut alors contenir des sous-éléments pour décrire l'ensemble des sous-composants ainsi que leurs liaisons.

FRACTAL ADL permet de répartir dans plusieurs fichiers la description d'une architecture. Les relations entre ces fichiers peuvent prendre des formes différentes. Premièrement, une définition peut en étendre une autre en y rajoutant certains éléments ou en surchargeant certaines propriétés. Un exemple de ce type de relation est illustré dans le cas de la définition *ClientImpl* qui étend la définition *ClientType*. Deuxièmement, un élément `component` qui est utilisé pour définir le contenu d'un composite peut faire réfférennce à une définition faite dans un autre fichier. Par exemple, le composant *client* fait référence à la définition *ClientImpl*.

La grammaire du langage FRACTAL ADL est spécifiée à l'aide de la technologie *Document Type Definitions* (DTD) de XML. Comme illustré dans la figure 4.4, la structure de chaque élément présent dans l'ADL est définie en utilisant une construction mini-DTD. Un mini-DTD est constitué de deux parties. La première partie, appelée *element*, définit l'ensemble des sous-éléments qu'un élément donné peut accepter. Par exemple, l'élément *component* accepte comme sous-élément des *interfaces*, des *composants*, des *liaisons*, un *contenu*, etc. La deuxième partie, appelée *attlist*, spécifie les arguments qu'accepte un élément donné. Par exemple, l'élément *component* accepte deux attributs, `name` et `definition`, et la présence du premier est obligatoire.

```
<definition name="ClientType">
 <interface name="r" role="server"
            signature="java.lang.Runnable"/>
 <interface name="s" role="client"
            signature="printer.api.Service"/>
</definition>

<definition name="ClientImpl" extends="ClientType">
 <content class="ClientImpl"/>
 <controller desc="primitive"/>
</definition>

<definition name="HelloWorld">
 <interface name="r" role="server"
            signature="java.lang.Runnable"/>
 <component name="client" definition="ClientImpl"/>
 <component name="server">
  <interface name="s" role="server"
             signature="Service"/>
  <content class="ServerImpl"/>
  <controller desc="primitive"/>
 </component>
 <binding client="this.r" server="client.r"/>
 <binding client="client.s" server="server.s"/>
</definition>
```

FIG. 4.3 – La description d'une application *Hello World* composée d'un composant client et d'un composant serveur.

```
<!ELEMENT component
   (interface*,component*,binding*,
    content?,attributes?,controller?) >
<!ATTLIST component
  name CDATA #REQUIRED
  definition CDATA #IMPLIED
>

<!ELEMENT content () >
<!ATTLIST content
  class CDATA #IMPLIED
>
```

FIG. 4.4 – Un extrait de la spécification de la grammaire du langage FRACTAL ADL.

4.2.2 Extensibilité de FRACTAL ADL

Le modèle de composants FRACTAL est utilisé pour programmer des systèmes appartenant à des classes d'applications très variées. Dans ce contexte, la logique de conception du langage FRACTAL ADL est de fournir une base minimale et extensible afin de pouvoir satisfaire les contraintes spécifiques à différents contextes applicatifs. Ces extensions sont de trois types : des extensions qui correspondent à des extensions faites au niveau du modèle de composants, des extensions qui concernent le canevas d'implantation utilisé et enfin des extensions pour ajouter dans l'ADL des informations supplémentaires, comme des commentaires. Notons que ce dernier type est orthogonal au modèle de composants et au canevas d'implantation. Nous reprenons, par la suite, chacun de ces types d'extensions afin d'illustrer leurs prise en compte au travers d'exemples.

4.2.2.1 Extensions reflétant des extensions au modèle FRACTAL

Etant donné que FRACTAL est un modèle extensible, son ADL doit accepter les extensions qui peuvent être faites au niveau du modèle de composants. Un exemple typique d'extension au modèle de composants est présenté dans le canevas DREAM [Qué05] qui est consacré à la construction d'intergiciels spécialisés. DREAM étend le modèle FRACTAL en définissant deux types de composants, appelés *actifs* et *passifs*. Les *composants* actifs peuvent contenir leurs propres flots d'exécution alors que les *composants passifs* sont traversés par des flots d'exécution. Les *composants actifs* implantent tous une interface spécifique, appelée *TaskController*, qui permet à l'environnement du composant de contrôler ses flots d'exécution. L'implantation de cette interface de contrôle est mise en œuvre par une extension à la membrane des *composants actifs* et expose des paramètres de configuration pour ajuster le nombre de flots d'exécution à initialiser dans le composant. Par conséquent, le langage ADL est lui-même étendu afin de refléter la présence de ces contrôleurs et de leurs paramètres en vue de permettre aux architectes de système de configurer les composants actifs dans la description d'architecture.

Un autre exemple d'extension au modèle FRACTAL qui a été effectuée dans le cadre de DREAM concerne la sémantique de communication implantée dans les liaisons. En effet, DREAM distingue les *liaisons synchrones* et les *liaisons asynchrones* entre les composants. Il s'est avéré utile de spécifier la sémantique de communication implantée par les liaisons au niveau de l'ADL. Dans ce but, une extension d'ADL a été effectuée en ajoutant un argument appelé ıprotocol à la spéfication des liaisons. La figure 4.5 illustre un extrait d'ADL et la modification effectuée au niveau de la mini-DTD des liaisons pour prendre en compte cette extension.

```
<definition name="ComplexHelloWorld">
 <component name="client1" definition="ClientImpl"/>
 <component name="client2" definition="ClientImpl"/>
 <component name="serveur" definition="ServeurImpl"/>
 <binding client="client1.s" server="serveur.s"
          protocol="asynch"/>
 <binding client="client2.s" server="serveur.s"
          protocol="synch"/>
</definition>
```

```
<!ELEMENT binding () >
<!ATTLIST binding
  client CDATA #REQUIRED
  server CDATA #REQUIRED
  protocol CDATA #IMPLIED // Ligne ajoutée pour l'extension
>
```

FIG. 4.5 – Un extrait d'ADL qui illustre la distinction entre les liaisons synchrones et asynchrones. L'extension au langage est effectuée en ajoutant une ligne au mini-DTD de l'élément *binding*.

4.2.2.2 Extensions au canevas d'implantation

Il existe de multiples implantations du modèle FRACTAL dans divers langages de programmation. Les canevas d'implantation peuvent avoir des propriétés variées qu'il peut être intéressant de représenter au niveau de l'ADL. Pour illustrer ce type d'extension, considérons le cas du canevas logiciel JULIA qui fournit une librairie *runtime* pour le déploiement des composants écrits en Java. Dans ce cadre, un exemple d'extension à l'ADL serait la prise en compte des paramètres de configuration de la machine virtuelle java en fonction des types de composants déployés. La figure 4.6 présente un extrait d'ADL où l'on spécifie que la machine virtuelle doit être configurée de manière à prendre en compte les assertions utilisées dans l'implantation des composants. La prise en compte de cette extension nécessite la définition d'un nouvel élément d'ADL, appelé *jvmargs*, qui est un sous-élément de la spécification du contenu des

composants. À cette fin, un nouveau mini-DTD est défini pour spécifier la sous-grammaire de l'élément *jvmargs* et ce dernier est enregistré comme une sous-balise de l'élément *content*.

```
<definition name="ClientImpl" extends="ClientType">
 <content class="client" language="julia">
   <jvmargs  value="--enableassertions" />
 </content>
</definition>
```

```
<!ELEMENT jvmargs EMPTY> // Définition d'un nouvel élément jvmargs
<!ATTLIST jvmargs
  value CDATA #REQUIRED
>
<!ELEMENT content(jvmargs*) > // Acceptation de multiples jvmargs comme sous-éléments
<!ATTLIST content
  class CDATA #IMPLIED
  language CDATA #IMPLIED
>
```

FIG. 4.6 – Un extrait d'ADL qui illustre la spécification de la prise en compte des assertions dans la machine virtuelle où le composant *ClientImpl* sera déployé.

4.2.2.3 Extensions pour l'intégration d'informations annexes

Un troisième type d'extensions supporté par le langage FRACTAL ADL concerne les informations annexes qui sont orthogonales au modèle de composants et aux canevas d'implantations. Il s'agit par exemple de la documentation du comportement des composants ou bien de l'inclusion de coordonnées pour chaque composant afin de permettre leurs affichage à l'aide d'une interface graphique. La figure 4.7 illustre un exemple très simple de ce type d'extensions ou l'on rajoute des commentaires dans les balises de l'ADL. La prise en compte de cette extension se fait en définissant un nouvel élément pour les commentaires, de façon semblable à ce qui à été illustré au paragraphe précédent. Cet élément est ensuite référencé comme sous-élément dans la spécification de tous les autres éléments de l'ADL qui peuvent inclure des commentaires.

```
<definition name="ClientImpl" extends="ClientType">
 <content class="ClientImpl"/>
  <comment lang="en" text="Prints HelloWorld via a printer"/>
 </content>
 <controller desc="primitive"/>
</definition>
```

```
<!ELEMENT comment EMPTY >
<!ATTLIST comment
  language CDATA #IMPLIED
  text CDATA #IMPLIED
>
<!ELEMENT content (comment*) >
<!ATTLIST content
   class CDATA #IMPLIED
>
```

FIG. 4.7 – Une définition de composant incluant des commentaires.

4.3 JULIA : Une implantation du modèle FRACTAL en Java

Il existe plusieurs implantations du modèle FRACTAL en Java. Parmi ces implantations, citons ProActive [BCM03] qui est destiné à la programmation des grilles d'ordinateurs, AOKell [SPC06] qui utilise la programmation par aspects [FECA04] au travers d'AspectJ [Kis02], et finalement JULIA [BCL+06] qui est l'implantation de référence.. Nous consacrons cette section à la présentation de JULIA car elle illustre comment une implantation à usage général du modèle FRACTAL peut être réalisée à l'aide d'un langage orienté objets.

De manière générale, JULIA est un canevas logiciel qui permet d'instancier des composants FRACTAL dont le contenu est implanté sous forme de classes Java. Pour ce faire, il fournit en ensemble de contrôleurs ainsi qu'une *fabrique* de composants qui permet de les assembler avec l'implantation des contenus. Le canevas se présente sous la forme d'une libraire *runtime* qui s'exécute au dessus d'une machine virtuelle Java (JVM) classique. Notons que JULIA s'exécute sur tout type de JVM, même sur celles qui sont les plus restreintes comme KVM et Java *Micro Edition*.

4.3.1 Structures de données associées aux composants

Comme illustrée dans la figure 4.8, un composant JULIA est implanté par plusieurs objets Java. Ces objets peuvent être regroupés en trois classes.

- Les objets qui implantent le **contenu** du composant. Ces objets peuvent être aussi bien l'implantation du contenu d'un composant primitif que l'ensemble des objets qui implantent les sous-composants.
- Les objets qui implantent la **membrane** du composant. Deux types d'objets sont distingués à ce stade : les contrôleurs qui implantent les interfaces de contrôle et les intercepteurs qui implantent l'interception des appels entrant ou sortant qui sont effectués sur les interfaces fonctionnelles du composant. Etant donné que les contrôleurs et les intercepteurs peuvent avoir des interdépendances, ils tiennent à jour des références qui leur permettent d'accéder directement aux contrôleurs voisins.
- Les objets **interfaces**. Ces objets constituent les seules références que le composant rend accessible à son environnement. En fonction de la localisation des objets référencés, on peut distinguer les *objets interfaces serveur* qui contiennent des références vers des objets qui sont dans le composant et des *objets interfaces client* qui contiennent des références vers d'autres objets interfaces qui sont rendus visibles au travers des liaisons.

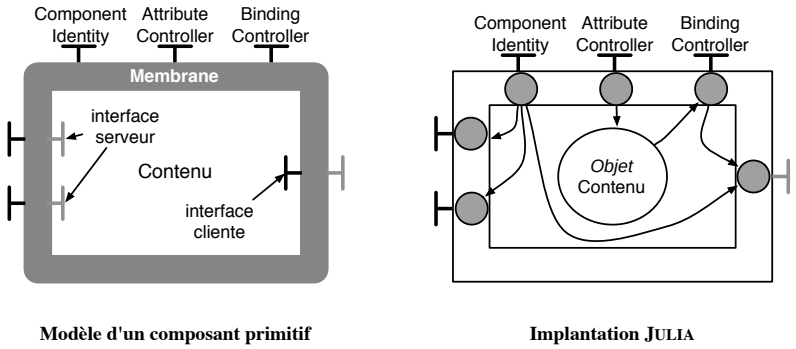


FIG. 4.8 – Implantation d'un composant FRACTAL dans JULIA.

JULIA fournit une *fabrique* de composants pour simplifier l'instanciation des composants. Cette *fabrique* prend en entrée trois listes de classes conformément à la classification présentée ci-dessus et

retourne une référence à une instance de composant. Cette fabrique peut soit être utilisée directement en Java, soit être dirigée par le langage de description d'architecture de FRACTAL présenté dans la section 4.2

JULIA offre deux mécanismes d'optimisation pour améliorer les performance des composants. Le premier est une optimisation interne à chaque composant qui vise à réduire la taille mémoire des objets de contrôle en les réunissant au sein d'un même objet. Cette optimisation est basée sur des manipulations de *bytecode* à l'aide de l'outil ASM [ASM02]. Les lecteurs intéressés peuvent trouver des détails techniques de cette optimisation dans [BCL+06]. Le deuxième mécanisme d'optimisation vise à améliorer la chaîne de communication entre des composants. Il intervient au niveau des schémas de liaisons qui traversent plusieurs composites. L'optimisation consiste à court-circuiter les interfaces exportées et importées en liant directement les deux composants communicants. Cette optimisation a l'inconvénient d'être une implantation qui n'est pas conforme au schéma architectural proposé par FRACTAL. Comme cette non conformité peut empêcher des reconfigurations ultérieures, les concepteurs peuvent annuler cette optimisation afin de respecter la conformité entre l'implantation et l'architecture des composants en payant le coût des indirections.

4.3.2 Mise en place des contrôleurs

Comme on l'a présenté à la section précédente, le modèle FRACTAL accepte des schémas arbitraires de contrôle sur les composants. Ceci est mis en œuvre par l'implantation de plusieurs objets contrôleurs qui interagissent potentiellement. Pour permettre l'extension systématique du méta-comportement des composants en les encapsulant dans des membranes différentes, JULIA propose une méthodologie d'implantation et d'assemblage de contrôleurs.

L'idée de base est de fournir un moyen d'écrire des classes distinctes pour différents contrôleurs, tout en assurant leurs interactions de manière efficace. La première solution serait d'utiliser la délégation, mais celle-ci est inefficace. Une deuxième idée serait l'héritage des objets. Celle-ci est écartée car elle impose une interaction hiérarchique, ce qui fait qu'un schéma d'héritage qui supporterait l'interaction entre n contrôleurs nécessiterait l'écriture de 2^n classes. Une solution efficace pourrait être l'utilisation de la programmation par aspects. Cette solution est en effet adoptée par AOKell en utilisant AspectJ, mais JULIA vise une implantation en Java pur.

La solution adoptée par JULIA est par conséquent l'utilisation des classes *mixins* pour programmer les contrôleurs de composants. Remarquons que l'utilisation des classes *mixins* de JULIA ne nécessite pas un compilateur Java modifié ou un pré-processeur de code source comme c'est le cas pour JAM [ALZ00]. En effet, JULIA utilise un patron logiciel pour assurer la transformation de *bytecode* à l'aide de l'outil ASM [ASM02]. Ce patron consiste à écrire toutes les classes contrôleurs sous forme de classes abstraites en respectant certaines conventions de nommage. Plus précisément, les champs ou méthodes d'une classe père doivent être préfixés par _super_ et ceux qui sont locaux à la classe en question doivent être préfixés par `_this_`.

La figure 4.9 illustre un exemple de mise en œuvre de contrôleurs à l'aide du canevas JULIA. Soit la classe `DefaultAttrCtrl`, un niveau de contrôle qui doit être implanté par tous les composants. Cette classe assure en effet que la modification d'un attribut sera interdite pendant l'exécution du composant. La méthode `setAttr` de cette classe est générique et invoque la méthode correspondante de son père après avoir effectué ses vérifications. L'implantation spécifique de la méthode `setAttr` est fournie par la classe `BaseAttrCtrl` qui affecte une valeur à un champ d'attribut. Ces deux classes d'implantation de contrôleurs sont compilées en *bytecode* par un compilateur classique de Java. Les *bytecode* obtenus sont fusionnés au moment de l'instanciation du composant par la *fabrique de composants*. Le résultat de la fusion est présenté dans la même figure à droite. On retrouve dans cette classe, de manière fusionnée, l'ensemble des méthodes et champs déclarés dans les deux premières classes. Ainsi, plusieurs classes d'implantation

```
abstract class DefaultAttrCtrl {
  abstract void _super_setAttr () ;
  public boolean running ;
  public void setAttr (int val) {
    if(!running)
      _super_setAttr(val) ;
    else
      throw new
            Exception("Running␣component");
  }
}

abstract class BaseAttrCtrl {
  public void setAttr (int val) {
    attr = val ;
  }
}
```

```
public class GeneratedAttrCtrl {

  // de BaseAttrCtrl
  public void setAttr$0 (int val) {
    attr = val ;
  }

  // de DefaultAttrCtrl
  public void setAttr (int val) {
    if(!running)
      setAttr$0(val) ;
    else
      throw new
            Exception("Running␣component");
  }

}
```

FIG. 4.9 – La mise en œuvre d'un contrôleur d'attributs à deux niveaux en JULIA (à gauche) et le code obtenu après le mixage (à droite).

de contrôleurs forment un seul objet, ce qui rend leur interaction possible et efficace.

4.3.3 Mise en place des intercepteurs

JULIA permet le développement d'intercepteurs dont le rôle est d'effectuer des pré- et des post-traitements à chaque invocation des méthodes d'une interface donnée. Un exemple d'intercepteur pourrait être un compteur de *threads* qui compte le nombre d'appels entrants et qui décompte les retours de méthodes ou les invocations sortantes afin de comptabiliser le nombre de *threads* qui sont actifs au sein d'un composant à un moment donné.

Afin de faciliter le développement d'intercepteurs, JULIA fournit un *générateur dynamique d'intercepteur*. Ce générateur prend en entrée une interface et sa classe d'implantation, ainsi qu'un ensemble de tisseurs de code qui sont écrits en respectant un patron de conception. En résultat, le générateur construit une classe qui hérite de la classe d'implantation et qui implante toutes les méthodes de l'interface en ayant intégré tous les aspects d'interception tels qu'ils sont spécifiés par les tisseurs de code. Remarquons que la génération d'intercepteur se fait au niveau *bytecode* et dynamiquement lors de l'instanciation d'un composant à l'aide de l'outil ASM [ASM02].

4.3.4 L'usine de déploiement pour FRACTALADL

Un outil de déploiement, intitulé l'usine de FRACTALADL[1], accompagne les implantations en Java du modèle afin de faciliter l'instanciation des configurations à partir des descriptions d'architecture. Cette usine adopte elle-même une architecture à base de composants afin d'être extensible pour accepter les extensions au modèle de composants et/ou à son ADL.

L'architecture de l'usine ADL comporte cinq composants principaux, comme illustré dans la figure 4.10. Nous présentons ci-dessous les fonctions de ces composants.

- Le composant *factory* assure le contrôle de l'exécution de l'usine. Après avoir initialisé les composants de l'usine, le *factory* appelle successivement les composants *loader*, *compiler* et *scheduler*.
- Le composant *loader* est en charge de construire une représentation abstraite de l'architecture (AST pour *Abstract Syntactic Tree*) décrite dans les fichiers ADL. Ce composant contient une chaîne de sous-composants. Tout au fond de la chaîne, le composant *parser* lit les fichier de des-

[1]Le nom original en anglais est FRACTAL*ADL Factory*.

cription et les transforme en AST. Ensuite, cet AST est analysé par les composants qui procèdent à des vérifications sémantiques particulières. L'ensemble des fonctions d'analyse implantées dans le composant *loader* est extensible dans le sens qu'il suffit de rajouter un nouveau composant dans la chaîne pour supporter de nouveaux types d'analyses.

– Le composant *compiler* est en charge de parcourir la représentation abstraite de l'architecture et de créer des tâches à exécuter afin de déployer l'architecture spécifiée. Ce composant contient un ensemble de sous-composants qui visitent tous l'AST en suivant un patron visiteur. Chaque sous-composant est dédié à la création d'une tâche bien spécifique. Des exemples typiques de tâches sont la création d'un composant, l'établissement d'une liaison, etc.
– Le composant *backend* encapsule un ensemble de sous-composants fournissant l'implantation concrète de tâches créées par le *compiler*. Des *backends* différents peuvent être utilisés pour obtenir des caractéristiques de déploiement différentes. Par exemple, parmi les *backends* fournis dans le paquetage de l'usine FRACTAL ADL, on en trouve un qui déploie directement une configuration ADL sur une machine virtuelle Java, alors qu'un autre génère le code de déploiement.
– Le composant *scheduler* exécute les tâches créées par le composant *compiler* dans un ordre correct en résolvant leurs contraintes de dépendance.

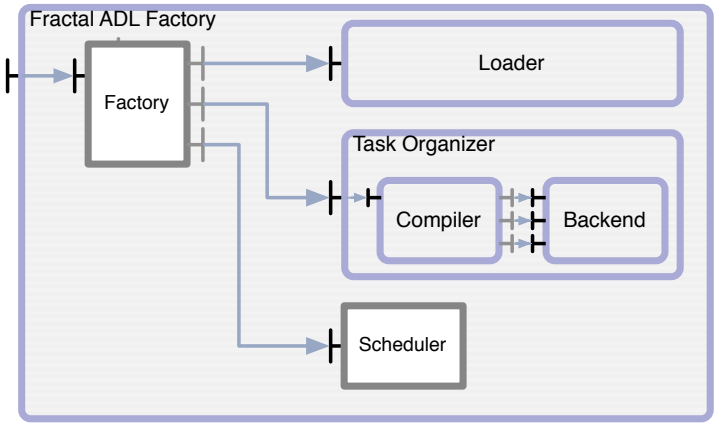


FIG. 4.10 – Architecture de haut niveau de l'usine de déploiement de FRACTAL ADL.

4.4 THINK : Une implantation du modèle FRACTAL en C

Au départ, THINK a été conçu comme un canevas logiciel pour la construction de noyaux de systèmes d'exploitation spécialisés en se basant sur la technologie de programmation à base de composants [FSLM02, Fas01]. En effet, la création de THINK précède la spécification du modèle FRACTAL. Le modèle de composant adopté dans sa version originale [FSLM02] était inspiré de RM-ODP [ISO95]. Cette version était accompagnée d'une bibliothèque de composants contenant des éléments de systèmes d'exploitation et d'un support ADL. Au début de l'année 2004 a été définie la deuxième version du canevas THINK. À partir de cette version, le modèle de composant adopté est devenu FRACTAL. La contribution majeure a consisté à transformer le modèle d'implantation binaire des composants et à adapter l'ADL afin de supporter des interfaces de contrôles FRACTAL. Une des contributions des travaux présentés dans la partie suivante de ce document a été la mise en place d'une nouvelle chaîne d'outils pour supporter la programmation des composants THINK ainsi que l'amélioration du modèle binaire pour permettre des optimisations dans l'implantation des composants ; l'intégration de ces améliorations a donné naissance à la version 3.

Nous consacrons cette section à la présentation de la version 2 du canevas THINK pour identifier les

bases que nous avons considérées dans le cadre de nos travaux. La présentation détaillée des éléments de la bibliothèque de composants de systèmes d'exploitation fournie avec THINK étant en dehors de notre contexte d'intérêt dans ce chapitre, nous nous concentrons uniquement sur la structure de données proposée pour l'implantation des composants ainsi que sur la chaîne d'outils qui facilitent la programmation des composants.

4.4.1 Structures de données associées aux composants

Le langage C n'offre pas de mécanisme de base pour l'implantation du concept d'interface. Pour combler ce manque, THINK définit une représentation binaire d'interface qui se présente sous la forme de la structure de données présentée dans la figure 4.11. Le descripteur d'interface est un couple de pointeurs *vtbl, données*. Le pointeur *vtbl* pointe vers une table de fonctions virtuelles qui contient les adresses d'implantation de l'ensemble des méthodes appartenant à une même interface. Ainsi, il est possible de sélectionner la méthode recherchée d'une interface à partir du descripteur de cette dernière. Les composants peuvent avoir des données d'instances de façon semblable aux champs d'un objet C++. Il est alors nécessaire de stocker au niveau des interfaces un pointeur vers les données d'instance du composant. C'est le rôle du pointeur *données*. Ce pointeur permet de retrouver les données propres à une instance de composant à partir d'un descripteur d'interface, et de passer la référence à ces données aux méthodes lors de leurs invocations via le paramètre `_this`.

Remarquons que ce modèle d'implantation est très similaire aux tables de fonctions virtuelles du langage C++. La seule différence qui sépare le modèle THINK de ce dernier est le fait que les données d'instances ne sont pas stockées dans le descripteur d'interface. Ce choix est justifié par le fait que, comme un composant COM, un composant THINK peut fournir de multiples interfaces serveur, d'où la nécessité de partager les mêmes données d'instances par plusieurs descripteurs d'interface. Enfin, notons que seule la structure de données d'instances et les descripteurs d'interfaces sont dupliqués au moment de la création d'un composant d'un même type. Les tables de fonctions virtuelles et le code d'implantation des méthodes sont partagés par les différentes instances d'un même composant et sont configurés pour une instance donnée au moment de l'exécution par le biais du paramètre `this`.

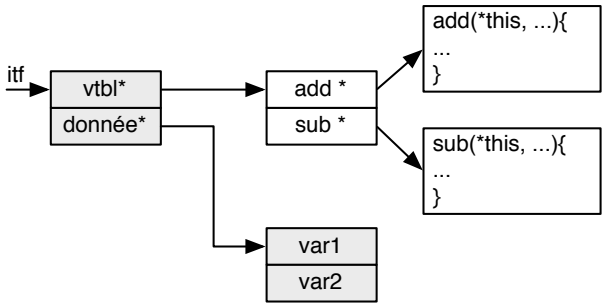


FIG. 4.11 – Représentation binaire des interfaces de composant en THINK. Les éléments en blanc sont partagés par plusieurs instances de composants de même type alors que les éléments en gris sont propres à chaque instance.

L'implantation des *composants primitifs* conformément au modèle FRACTAL se greffe au dessus de ce modèle binaire de base. La figure 4.12 illustre l'organisation des structures de données qui permettent d'implanter les interfaces de contrôles spécifiées par FRACTAL. Nous détaillons ci-dessous le rôle de chacun des éléments présents dans cette figure.

– La **structure des interfaces serveur** commence par un descripteur d'interface correspondant à l'interface de contrôle *component identity*. Cette interface a pour rôle de fournir des méthodes

d'introspection. Pour ce faire, elle utilise la structure des interfaces serveur elle-même et le nombre d'interfaces figurant dans la case suivante. Les autres éléments (à partir de la troisième ligne) représentent les descripteurs d'interfaces enrichies par des données d'introspection comme le nom ou le type. Notons que les pointeurs de données des interfaces de contrôle peuvent optionnellement pointer sur d'autres structures que la structure des données d'instance. Ceci est par exemple le cas de l'interface *binding controller* dont les données sont constituées par une structure spéciale des interfaces client.

- La **structure des interfaces client** commence par un champ qui note le nombre d'interfaces présentes. Les autres entrées sont des couples $nom, itf*$ qui permettent de retrouver des interfaces client à partir de leurs noms. Remarquons que cette structure ne contient pas directement les valeurs obtenues au travers des liaisons, mais pointe vers les champs concernés qui sont déclarés manuellement par le programmeur parmi les données d'instance. L'interface de contrôle *binding controller* est implantée de manière générique en se basant sur des informations présentes dans cette structure de donnée.
- La **structure des attributs** commence par un champ qui note le nombre d'attributs du composant. Le reste de la structure est constitué d'un tableau de couples $nom, attr*$ qui permet de retrouver les attributs qui sont déclarés parmi les données d'instances du composant en fonction de leurs noms. L'interface *attribute controller* est implantée de manière générique en se basant sur des informations présentes dans cette structure de données.
- La **structure de données d'instance** contient l'ensemble des références aux interfaces client, aux attributs et aux variables privées du composant.

L'implantation des *composants composites* est elle aussi basée sur des structures de données similaires à celles des *composants primitifs*. Par contre, étant donné que l'implantation d'un *composant composite* consiste uniquement en l'implantation d'une *membrane*, l'organisation des structures de données peut être radicalement différente en fonction des comportements de contrôle fournis par ce dernier. Pour cette raison, nous nous proposons de retenir simplement que différentes organisations sont possibles et nous renvoyons le lecteur intéressé à l'analyse des différentes implantations de membranes proposée dans [Thi].

4.4.2 Génération des structures d'interfaces

THINK utilise un langage de description d'interface (IDL) afin de spécifier les interfaces de composants dans un langage fortement typé. L'IDL de THINK est très similaire aux constructions d'interface du langage Java. Un compilateur d'interface est alors utilisé pour transformer les descriptions d'interface en des structures binaires telles que présentées dans la figure 4.11.

La figure 4.13 illustre un exemple d'interface et le résultat de sa compilation. Dans cette exemple, il s'agit d'une interface qui contient deux méthodes correspondant à des opérations arithmétiques. Si l'on analyse le résultat de la compilation de cette interface, on retrouve la déclaration de deux structures de données. Le première structure de donnée implante le descripteur d'interface, intitulé `A`rithmeticItf, pour l'interface `A`rrithmetic. Le descripteur d'interface encapsule un pointeur pour faire référence à la table de fonctions virtuelles et un pointeur pour désigner la structure des données d'instance. À ce niveau, le pointeur vers les données d'instance n'est pas typé car cette structure sera définie par le programmeur en fonction des différents composants qui implanteront cette interface. La deuxième structure de données présente dans le résultat de compilation est la structure de la table de fonctions virtuelles qui contient des pointeurs vers les implantations des méthodes de l'interface. Remarquons qu'un pointeur `_this` est inséré dans la déclaration de toutes les méthodes en tant que premier paramètre pour permettre le passage de la référence aux données d'instances lors de leur invocation.

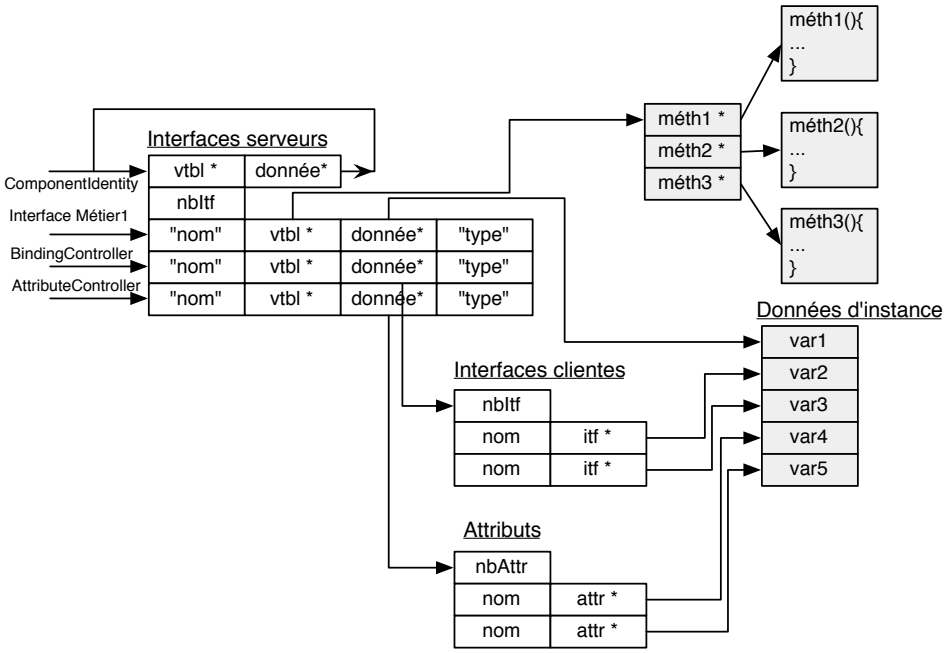


FIG. 4.12 – Implantation d'un composant primitif FRACTAL en THINK. Les cases dont le fond est gris sont définies par le programmeur alors que celles dont le fond est blanc sont générées par le compilateur.

```
// Arithmetic.idl

public interface Arithmetic {
  int   add (int a, int b) ;
  float div (double a, double b) ;
}
```

```
// Arithmetic.idl.h

struct ArithmeticMeth ;
// Descripteur d'interfaces
typedef struct {
  struct ArithmeticMeth *vtbl ;
  void *instance_data ;
} ArithmeticItf ;

// Table de fonctions virtuelles
struct  ArithmeticMeth {
  int   (*add) (void *this, int a, int b) ;
  float (*div) (void *this, double a, double b) ;
}
```

FIG. 4.13 – Description d'une interface (en haut) et le résultat de sa compilation (en bas).

4.4.3 Patron de programmation

La programmation des composants THINK en C nécessite la prise en compte d'un certain nombre de conventions de nommage et l'initialisation de certaines structures de données. Les éléments gris présentés dans la figure 4.12 représentent les parties de code qui doivent être mises en œuvre par le programmeur.

Afin d'illustrer comment est implanté un *composant primitif* en C, nous nous proposons d'analyser l'extrait de code illustré dans la figure 4.14 où il s'agit d'une implantation possible de l'interface d'opéra-

tions arithmétiques présentée dans la figure 4.13. Le fichier d'implantation commence par la déclaration de la structure d'instance. Le nom de cette structure doit porter le nom du composant, dans ce cas *myComp*, suffixé par `data`, afin d'assurer la compatibilité avec le code généré par le compilateur de THINK [2]. On trouve ensuite, l'implantation des méthodes fournies. Remarquons que le pointeur `_this` récupéré en premier paramètre est *casté* au début de chaque méthode afin d'indexer correctement les données d'instance du composant. La *macro* `CALL` est utilisée pour invoquer une opération via une interface client du composant. Une fois terminée la déclaration des méthodes fournies, le programmeur doit initialiser la structure de la table de fonction virtuelle correspondant aux interfaces fournies. Une convention de nommage est encore à respecter lors du nommage de cette table.

```
// Données d'instance
struct myCompdata {
  int nbrInvocation ; // Attribut
  ConsoleItf *console ;
}

// Implantation des fonctions
static int myComp_add (void *this, int a, int b) {
  struct myCompdata *self = (struct myCompdata)this ;
  CALL(self->nbrInvocation, afficheEntier, self->nbrInvocation ++) ;
  return a+b ;
}

static float myComp_div (void *this, float a, float b) {
  struct myCompdata *self = (struct myCompdata)this ;
  CALL(self->nbrInvocation, afficheEntier, self->nbrInvocation ++) ;
  return a/b ;
}

// Initialisation de la table de fonctions virtuelles
static struct ArithmeticMeth myCompArithmeticMeth {
  add : myComp_add, div : myComp_div,
} ;
```

FIG. 4.14 – Un extrait de code d'implantation en THINK.

4.4.4 Outil de génération de code

Le canevas THINK fournit un outil de génération de code pour faciliter la programmation des composants. Cet outil intègre les trois langages mis en œuvre qui sont l'ADL et l'IDL THINK ainsi que le langage de programmation C. Sa fonction principale est de générer une partie structurelle du code des composants en se basant sur les données architecturales et de diriger la compilation des codes générés ainsi que les codes d'implantation écrits par les programmeurs.

Le flot d'exécution de l'outil en question est illustré dans la figure 4.15. Tout d'abord le compilateur IDL est lancé pour convertir les fichiers de description d'interface en fichiers en-têtes qui contiennent les structures de descripteurs d'interface correspondant à ceux présentés dans les sections précédentes. Ce compilateur peut aussi être utilisé pour générer le fichier d'implantation de *stubs/skeletons*. Le compilateur ADL est ensuite lancé pour générer le code *glue* des composants. Cette génération de code concerne l'implantation des membranes des composants *composites* et *primitifs* comme illustré dans les parties blanches de la figure 4.12. Une fois l'exécution de ce générateur de code terminée, le compilateur d'ADL assemble le code d'implantation des composants à partir de la bibliothèque de composants et transforme l'ensemble des fichiers source et en-têtes en fichiers d'objets binaires. Enfin, un éditeur de liens est utilisé pour lier l'ensemble de ces fichiers objets afin de créer le fichier image résultant.

[2] Les parties en blanc présentées dans la figure 4.14.

Remarquons que le code généré par les générateurs de code d'ADL et d'IDL est du C standard. De plus, le compilateur et l'éditeur de liens qui sont utilisés pour compiler les fichiers sources sont des compilateurs standards (gcc) pour une plate-forme cible donnée. Par conséquent, l'outil de compilation de THINK est complètement reciblable et peut être facilement adapté à des plates-formes matérielles différentes.

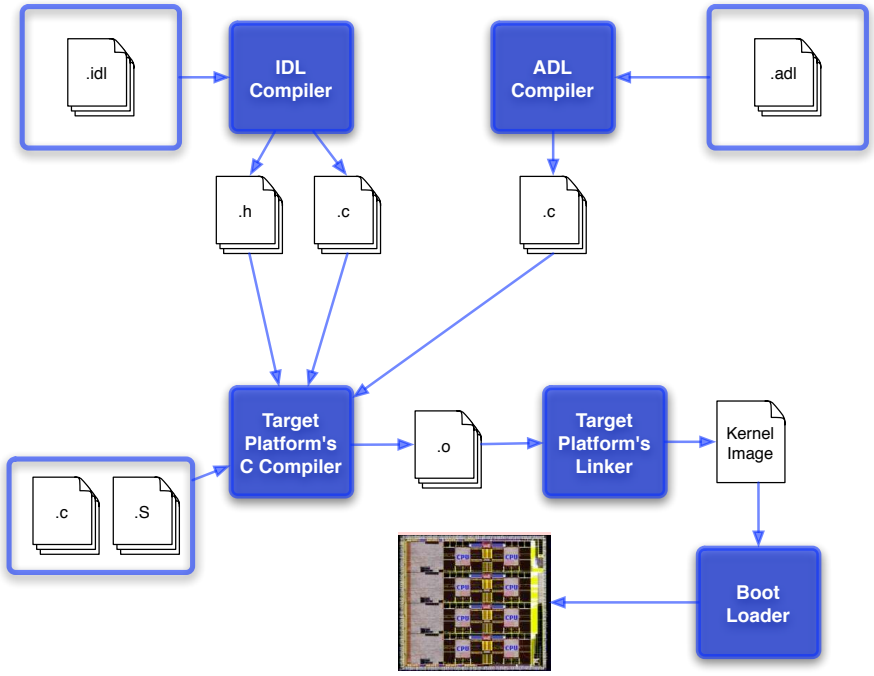


FIG. 4.15 – Flot d'exécution du générateur de code pour la version 2 de THINK.

Deuxième partie

Une chaîne d'outils extensible et multi-cibles pour le traitement de descriptions d'architectures

Chapitre 5

Présentation générale

Sommaire

La deuxième partie de ce manuscrit est consacrée à la présentation du canevas de construction d'outils de traitement d'architecture que nous avons mis au point pour contribuer à l'automatisation d'une partie du processus de développement de systèmes à base de composants. Ce chapitre est une présentation générale de notre approche. Nous le débutons par une analyse de l'état de l'art qui permet de souligner les limitations des outils existants. Nous identifions ensuite nos motivations et nos objectifs. Enfin, nous décrivons succinctement les caractéristiques de la chaîne d'outils mise au point et nous présentons ses principales contributions à l'état de l'art.

5.1 Choix du modèle de composants

Nos travaux sont motivés par la mise au point d'une méthodologie de développement pour la conception et l'implantation de systèmes logiciels complexes à destination de plates-formes multi-processeurs sur puce (MPSoC). Dans ce cadre, il est nécessaire de considérer un large spectre de domaines d'applications, allant des systèmes d'exploitation spécialisés aux applications multimédia. La première partie de ce document a été consacrée à une présentation des travaux académiques et industriels proposant des modèles et des outils de programmation dans des contextes connexes.

La programmation à base de composants est un fondement intéressant pour construire des processus de développement destinés à la conception des systèmes complexes. Les composants logiciels permettent une encapsulation forte des données et des comportements à l'aide d'interfaces qu'ils utilisent pour communiquer avec les autres composants au travers de liaisons explicites. Cette propriété d'encapsulation permet de découpler la manière dont les composants sont implantés de la manière dont ils sont assemblés. Par conséquent, l'architecture de l'assemblage des composants constituant le système, autrement dit l'architecture du logiciel, devient naturellement une préoccupation principale. Le chapitre 2 a présenté différents modèles de composants. Ce chapitre a montré qu'il y avait une grande diversité dans les préoccupations et les caractéristiques des différents modèles. Force est de constater qu'il n'existe aucun

modèle de composants satisfaisant l'ensemble des préoccupations possibles. En revanche, nous pensons que l'utilisation d'un modèle de composants ouvert, ayant la capacité d'extensions pour être adapté aux besoins spécifiques, peut satisfaire la problématique de la programmation des systèmes sur puce. Ainsi, **nous avons décidé d'adopter le modèle de composants FRACTAL**, présenté dans le chapitre 4, comme base de nos travaux.

5.2 À la recherche d'une chaîne d'outils ADL

Un avantage indéniable de la programmation à base de composants est qu'elle est souvent associée à un ensemble d'outils permettant de faciliter (et parfois d'automatiser) une partie du processus de développement des logiciels. Notre objectif est précisément de définir une telle chaîne d'outils qui soit adaptée au processus de programmation des systèmes sur puce. Une telle chaîne d'outils devrait répondre aux différentes préoccupations qui doivent êtres considérées pour la programmation de ce type de plates-formes. Dans la lumière de cette perspective, nous soulignons, par la suite, les limites des outils existants.

5.2.1 Limites des outils de traitement d'architectures existants

Une grande partie des outils conçus pour accompagner le processus de développement de logiciels à base de composants sont basés sur des descriptions d'architectures effectuées à l'aide de langages appelés ADL. Nous avons présenté un certain nombre de ces outils au sein du chapitre 3. Nous avons constaté une grande diversité dans les outils disponibles qui comprennent des outils d'analyse et de vérification d'architectures, des outils de génération de code permettant l'assemblage de systèmes à partir de bibliothèques, ou encore des outils de déploiement et de configuration dont le rôle est de déployer des architectures logicielles à partir des descriptions ADL. Nous avons aussi constaté que les langages ADL varient grandement en fonction des fonctionnalités implantées par les outils. Par exemple, nous avons vu que les outils de génération de code et de déploiement reposent sur des ADL permettant d'exprimer des aspects structurels du système (composants, interfaces, liaisons, etc.), alors que les outils de vérification utilisent des descriptions comportementales du système (protocoles de communications, stratégies de synchronisation, etc.). Nous constatons par conséquent que la prise en compte d'un processus de développement complet nécessite la considération de différents aspects (e.g. architecturaux, comportementaux). Nous appelons cela la prise en compte des **descriptions hétérogènes d'architecture**.

Notre objectif est de proposer une solution généraliste pour le processus de développement de logiciels basés sur des architectures à composants. Par conséquent, nous pensons qu'il est nécessaire de construire des outils *extensibles* pouvant être déclinés en des personnalités différentes, afin de satisfaire des besoins spécifiques associés à des contextes applicatifs différents. Les travaux sur les d'ADL extensibles s'inscrivent dans cet effort. Néanmoins, les propositions actuelles ne fournissent pas d'extensibilité au niveau outillage, ce qui est primordiale pour traiter, de manière systématique, les extensions effectuées au niveau de l'ADL.

5.2.2 Travaux connexes

Il existe d'autres travaux, non présentés dans l'état de l'art, fournissant des infrastructures extensibles, similaires à celle que nous recherchons dans le cadre de nos travaux. Il s'agit en particulier des outils développés dans le cadre de la *programmation générative*.

Les outils de *programmation générative* ont pour rôle de générer des programmes dans des langages généralistes à partir de programmes écrits dans des langage spécifiques à un domaine (DSL pour *Domain Specific Language*). L'objectif de cette approche est de définir des langages spécifiques qui fournissent des constructions et des abstractions adéquates à un domaine d'application donné. De multiples langages sont alors définis pour des contextes d'application différents, ce qui justifie l'objectif de trouver une

architecture de générateur de code extensible permettant de supporter différents langages d'entrée. Les travaux sur Soft/Arch/MTE [GCL05], Argo/MTE [CGH04] et Clearwater [SPJ+05] s'inscrivent dans ce contexte. Tout comme pour les ADL extensibles, ces outils reposent sur le langage XML qui est utilisé comme langage intermédiaire permettant de décrire de manière homogène les informations contenues dans les différents langages d'entrée. Généralement, XSLT est utilisé pour le traitement du langage intermédiaire. Bien que la problématique adressée par ces travaux soit proche de la notre, les propositions faites sont limitées à la génération ou à la transformation de code. En effet, aucune d'entre elles ne fournit de solution adaptée à la mise au point d'un outillage extensible assurant d'autres types de traitement (e.g. vérification, déploiement).

Enfin, le canevas logiciel *Eclipse Modeling Framework* (EMF) [EMF] propose des fonctions de génération de code, situé plutôt dans un cadre de conception à la MDA en se basant sur un formalisme intitulé *Ecore*. Ce dernier est un formalisme très pragmatique se limitant à la modélisation des objets logiciels. EMF est doté de nombreux outils de génération de code, destinés actuellement à la programmation en Java. Le fait que nous soyons à la recherche d'un outil dirigé par des descriptions d'architectures qui soit générique pour implanter différentes fonctions et indépendamment des langages de programmation nous oblige à surmonter les limites des outils dont nous disposons à l'heure actuelle. Cependant, notons que notre approche n'est pas incompatible avec ce type de travaux. Au contraire, les outils que l'on vise à mettre en place pourront directement s'intégrer dans ce type de processus de conception.

5.3 Vers un outil extensible pour le traitement des ADL hétérogènes

5.3.1 Objectif

L'objectif de cette thèse est de mettre au point un outil extensible qui servirait d'auxiliaire au processus de développement à base de composants en fournissant différentes formes de fonctions (e.g. déploiement, vérification, génération de code) à partir de descriptions hétérogènes d'architectures. Les caractéristiques principales que doit revêtir un tel outil sont les suivantes :

- **Intégration de divers langages de description d'architectures**. Les concepteurs doivent pouvoir exprimer différents éléments architecturaux et comportementaux d'un système logiciel, et ce à l'aide de langages spécifiques. Il est donc nécessaire que l'outil puisse être aisément étendu afin de prendre en compte un nouveau langage.
- **Support pour des opérations d'analyse variées**. L'outil doit permettre la mise en œuvre de fonctions d'analyse typiques, comme la vérification architecturale d'un assemblage de composants (e.g. liaisons correctes entre interfaces). L'ensemble des opérations d'analyse supportées doit être extensible afin de permettre aux concepteurs d'introduire de nouvelles opérations spécifiques à leurs besoins.
- **Support pour des fonctions de traitement variées**. L'outil doit permettre d'automatiser diverses fonctions, allant de la génération de code à la compilation ou au déploiement. Les modules de traitements doivent pouvoir prendre en compte différentes cibles afin de supporter des langages de programmation et des plates-formes matérielles divers. Notons qu'il est également nécessaire que les modules de traitements soient extensibles afin de pouvoir prendre en compte des éléments introduits tardivement par les concepteurs.

5.3.2 Proposition

Pour répondre aux objectifs suscités, nous avons mis au point un canevas logiciel permettant de construire des outils pour accompagner le processus développement de systèmes à base de composants. Ce canevas propose une architecture minimale et extensible autorisant l'intégration de plusieurs modules de chargement et de traitement. Les modules de chargement permettent la prise en compte de divers langages de description (ADL) ; les modules de traitement implantent des fonctions variées (e.g. géné-

ration de code, déploiement, compilation). Ces travaux ont pour base les travaux qui ont été menés par France Télécom et l'INRIA sur le langage de description d'architectures FRACTALADL, présenté dans le chapitre précédent.

Afin de prouver l'extensibilité du canevas construit, nous l'avons étendu afin de construire une personnalité dédiée à la génération de code. L'objectif de ce générateur de code est de simplifier l'assemblage de systèmes et d'applications embarquées à base de composants. Cette personnalité a été étendue afin de l'adapter à deux contextes applicatif différents : programmation d'applications multimédia et conception de systèmes d'exploitation reconfigurables.

La suite de cette section introduit le canevas logiciel proposé, ainsi qu'une personnalité de compilateur ADL que nous avons mis au point pour fournir des services de type génération de code. Cette personnalité a été utilisé dans le cadre de nos projets de développement de systèmes d'exploitation et d'application multimédia pour systèmes multiprocesseurs sur puce. De plus, cette personnalité à été encore étendue pour satisfaire deux domaines d'applications différents. La première personnalité concerne la programmation d'applications multimédias réparties. Cette extension est décrite dans le chapitre 9. La deuxième concerne des travaux principalement mené par une équipe de France Telecom pour la construction de systèmes d'exploitation reconfigurables. Elle n'est par conséquent pas présentée dans ce manuscript.

5.3.2.1 Un canevas logiciel extensible pour le traitement d'ADL hétérogènes

Les caractéristiques fondamentales du canevas logiciel mis au point peuvent être résumées en trois points. Tout d'abord, cet outil est capable d'accepter en entrée des langages de description (ADL) hétérogènes et de créer une représentation homogène de l'ensemble des informations reçues. D'autre part, l'outil intègre des composants de traitements travaillant sur la représentation unifiée de l'architecture. Enfin, la conception de l'outil est en très grande partie basée sur des *plugins*, ce qui facilite son extension par des tierces parties.

Afin de rendre extensibles les fonctions suscitées, nous avons opté pour une structure de type *exogiciel* [EK95] pour le cœur de l'outil. Ceci consiste à la fois à architecturer le cœur de l'outil sous forme d'un canevas logiciel à composants, et à définir un ensemble de patrons de programmation. Ces patrons facilitent la programmation des différents composants du canevas. Ce canevas logiciel est constitué des éléments de base suivants :

- **Un AST** (*Abstract Syntactic Tree*) qui est une représentation intermédiaire sous forme d'arbre fusionnant l'ensemble des informations architecturales reçues en entrée. Cet AST est extensible. Il peut ainsi être adapté à différents langages de description d'architectures. Son extensibilité est assurée par deux facteurs. Premièrement, chaque nœud de l'AST implante des interfaces différentes pour accéder à des propriétés de natures différentes. Typiquement, l'interface d'accès aux données associées au nœud est séparée des interfaces qui permettent de naviguer vers ses fils. Cette architecture garantit que de nouvelles propriétés peuvent être associées aux nœuds sans modifier ses interfaces existantes. Deuxièmement, un mécanisme de génération automatique d'implantation des nœuds de l'AST est fourni. Ce mécanisme se base sur une description abstraite de l'organisation des nœuds d'AST.
- **Une chaîne de composants de construction de l'AST**. Cette chaîne intègre des composants à grain fin, implantant chacun des fonctions bien spécifiques. Parmi ces composants, nous pouvons distinguer (i) des analyseurs syntaxiques qui lisent des fichiers en entrée et qui participent à la construction d'un AST unifié, et (ii) des analyseurs sémantiques qui implantent des opérations de vérification et de modification d'architecture. Le contenu de cette chaîne de construction d'AST peut être modifié par les concepteurs afin de prendre en compte de nouveaux langages d'entrée et de rajouter de nouvelles opérations d'analyses.

– **Un canevas de tâches** qui permet de modéliser l'ensemble des traitements à mettre en œuvre à partir de la représentation intermédiaire. Ce canevas permet de construire un graphe de tâches à exécuter en identifiant précisément celles devant être exécutées, leurs interdépendances et les flots de données qui circulent entre elles. Le canevas de tâches est extensible, afin de permettre aux concepteurs de définir de nouveaux types de tâches. Enfin, un modèle de programmation original est défini afin de faciliter la construction du graphe de tâche.
– **Un module de traitement** dont le rôle est de lire l'AST afin de construire un graphe de tâches modélisant les opérations à exécuter. Ce module intègre l'ensemble des opérations de traitement supportées pour automatiser le processus de développement. Il adopte une architecture basée sur le patron de programmation *visiteur*. De plus, l'architecture du module de traitement est basée sur des *plugins*, afin de permettre aux concepteurs de mettre en place les opérations de traitements spécifiques à leurs besoins.

5.3.2.2 Une personnalité pour la génération de code

Nous avons construit une personnalité du canevas de traitement d'ADL hétérogènes dédiée à la génération de code. L'objectif recherché était de permettre l'assemblage de systèmes ou d'applications à partir de bibliothèques de composants. Cela comprend d'une part la génération de code *glue* pour lier des composants, et d'autre part la compilation du code en une structure binaire destinée à être exécutée sur une plate-forme matérielle de type MPSoC.

La personnalité réalisée est composée des éléments suivants :

– Une chaîne de construction spécialisée qui lit des fichiers de descriptions variés (notamment FRACTALADL, THINKADL et THINKIDL) et qui permet de traiter des langages de programmations différents (C, C++ et Java).
– Une version spécialisée du canevas de tâches qui est dédiée à la génération de code et à la compilation. Cette personnalité fournit en particulier des types de tâches adéquats pour effectuer la génération de code de manière modulaire et hiérarchique.
– Un module de traitement multi-cibles qui permet d'organiser et d'exécuter les opérations de génération et de compilation. Ce module est organisé en plusieurs hiérarchies de *plugins* afin de fournir une architecture capable de procéder à la génération de code en C, en C++ et en Java. Ce module est facilement spécialisable pour compiler le code généré vers des plates-formes matérielles diverses.

5.4 Organisation de la seconde partie

La suite de cette partie est organisée en deux chapitres. Le chapitre 6 présente l'architecture du canevas logiciel proposé. Sont décrits dans ce chapitre les modèles architecturaux et les mécanismes de base qui assurent la modularité et l'extensibilité de ce canevas. Nous décrivons ensuite, dans le chapitre 7, une personnalité adaptée à la génération de code multi-cibles. Ce chapitre illustre la façon dont les modèles architecturaux et les patrons de programmation présentés dans le chapitre 6 peuvent être utilisés pour implanter une chaîne d'outils générant du code pour les applications à base de composants FRACTAL implantés dans des langages de programmation variés.

Chapitre 6

Un canevas logiciel extensible pour le traitement d'ADL hétérogènes

Sommaire

Ce chapitre est consacré à la présentation de l'architecture du canevas logiciel mis au point dans le cadre de nos travaux. Ce canevas permet de construire des outils de traitement d'architectures logicielles prenant en entrée des descriptions d'architectures (potentiellement écrites dans des langages hétérogènes), et fournissant un ensemble de fonctionnalités pour la vérification d'architectures, la génération de code, la compilation, ou encore le déploiement.

Nous commençons par donner une vue d'ensemble permettant d'introduire l'architecture générale de ces outils de traitements d'architectures, ou compilateurs ADL, et la façon dont ils s'exécutent. Ensuite, nous présentons les caractéristiques de l'arbre de syntaxe abstraite qui constitue l'élément unificateur permettant de représenter une architecture décrite à l'aide de langages hétérogènes. Nous présentons également la chaîne de composants qui est en charge de construire cet AST, ainsi que le module de traitement de cet arbre abstrait. Celui-ci est constitué d'un ensemble de composants extensibles qui créent et exécutent une liste de tâches afin de procéder aux opérations implantées par le compilateur ADL. Enfin, nous présentons les mécanismes d'extension qui permettent de spécialiser l'outillage afin de l'adapter, de manière systématique, à des contextes d'utilisations spécifiques.

6.1 Vue d'ensemble

Comme nous l'avons introduit dans le chapitre précédent, nous basons nos travaux sur l'usine FRACTALADL. Notre objectif est de généraliser les fonctionnalités fournies par l'usine FRACTALADL pour en faire un outil généraliste pouvant accepter en entrée des descriptions d'architectures hétérogènes afin de produire des résultats de natures différentes (e.g. génération de code, compilation, etc.).

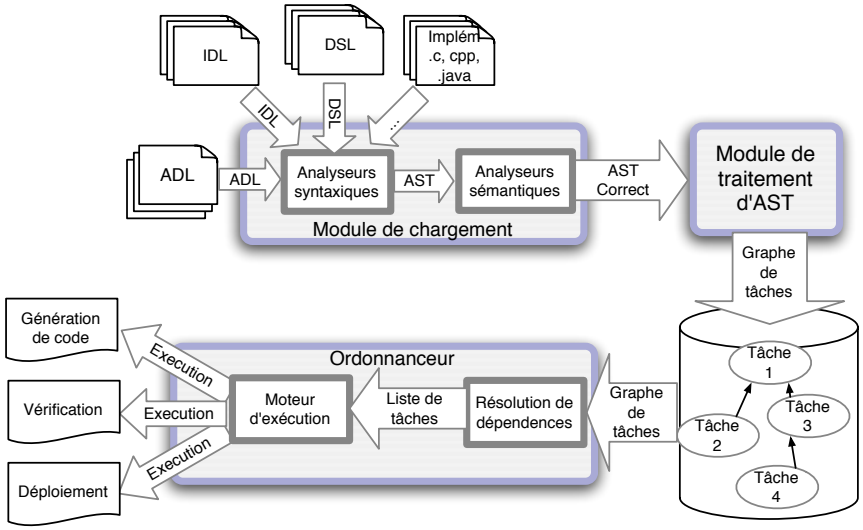


FIG. 6.1 – Flot d'exécution de l'usine THINKADL.

La figure 6.1 présente le flot d'exécution de l'usine que nous avons mise en place. Le module de chargement (*loader*) est en charge de lire l'ensemble des fichiers de description qui sont utilisés pour décrire l'architecture du logiciel cible. Il construit un arbre de syntaxe abstraite (AST) unifiant l'ensemble des informations obtenues en entrée. Notons que l'exécution du module de chargement est principalement dirigée par les fichiers ADL qui décrivent l'organisation structurelle du logiciel à construire. L'analyse de ces fichiers entraîne la lecture d'autres fichiers annexes pour intégrer, par exemple, des descriptions d'interfaces citées dans l'ADL, ou d'autres informations complémentaires décrites à l'aide de langages spécifiques à des domaines (DSL[1]). Une fois l'AST construit, une chaîne d'analyseurs sémantiques est exécutée afin de vérifier un certain nombre de propriétés sur l'architecture du logiciel à construire. Les analyseurs sémantiques peuvent identifier des erreurs de conception et également enrichir l'architecture avec des éléments supplémentaires. Par exemple, dans le cas d'un logiciel hétérogène contenant des composants écrits en C et en Java, un composant d'analyse sémantique peut détecter les interactions inter-plateformes et insérer des ponts *JNI*[2] dans l'architecture par le biais de la modification de l'arbre de syntaxe abstraite.

L'arbre de syntaxe abstraite construit par le module de chargement est utilisé par le module de *traitement de l'AST* pour construire un graphe de tâches à exécuter. Ce graphe modélise les dépendances entre tâches, ainsi que les données qu'elles échangent. A titre d'exemple, considérons un graphe de tâches (à gros grain) : une tâche en charge du déploiement d'un composant nécessite d'avoir à disposition une version compilée de ce composant ; la compilation du code de ce composant est implantée par une tâche qui dépend des tâches de génération du code source de ses interfaces et de sa *membrane*. Le composant de *traitement de l'AST* est composé de composants d'organisation et de composants implantant des tâches. Les composants d'organisation créent des tâches abstraites et définissent leurs inter-dépendances, alors

[1] DSL pour *Domain Specific Languages*.
[2] JNI pouur (*Java Native Interface).*

que les composants d'implantation fournissent la mise en œuvre concrète des tâches. Remarquons qu'un même ensemble de composants d'organisation peut être réutilisé avec des jeux de composants d'implantation différents. Ceci peut être utilisé, par exemple, pour générer du code (à l'aide d'un même graphe de tâches) dans des langages de programmation différents.

Le graphe de tâches est exécuté par le module d'*ordonnancement*. Celui-ci exécute les différentes tâches en respectant les contraintes de dépendances.

6.2 Arbre de syntaxe abstraite extensible

Un arbre de syntaxe abstraite (AST) est un formalisme de représentation de données indépendant de la représentation physique (ou concrète) des données. Les AST sont largement utilisés en tant que représentation intermédiaire pour mettre en place des compilateurs.

Dans le cadre de nos travaux, nous proposons de mettre en place un outil de compilation acceptant des fichiers de description écrits dans des langages hétérogènes. Afin de masquer cette hétérogénéité, nous optons utilisons un AST qui permet d'unifier la représentation des données. Remarquons qu'étant donné que nous souhaitons être capable d'étendre les compilateurs ADL après leur construction, il est nécessaire que l'AST soit lui-même extensible. Ceci permet notamment d'intégrer de nouveaux types de données.

Dans cette section, nous présentons l'architecture de l'AST extensible. Nous expliquons la façon dont il est créé, ainsi que les actions nécessaires à son extension.

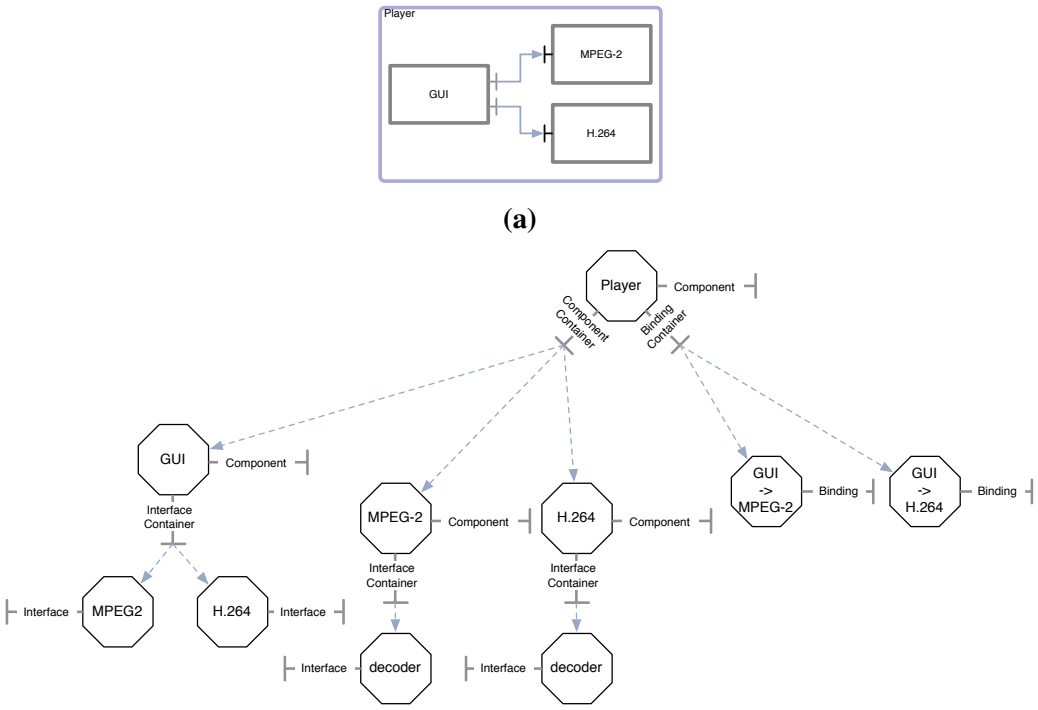
6.2.1 Architecture de l'arbre de syntaxe abstraite

L'AST est implanté sous forme d'un arbre d'*objets* (qui forment les nœuds). Chaque nœud de l'AST représente un élément bien défini de l'architecture, stocke les informations associées à ce dernier et donne accès à ses sous-éléments afin de permettre la navigation dans l'AST. Les nœuds de l'AST implantent différentes interfaces pour donner accès aux données qu'ils encapsulent. Ces interfaces peuvent être regroupées en trois classes :

1. **Une interface de base** qui donne accès aux informations générales sur l'élément représenté. Des exemples de ce type d'informations sont le nom de fichier dans lequel l'élément se trouve, ses coordonnées dans le fichier (numéro de ligne et de colonne), etc. De plus, cette classe de base permet à chaque nœud de porter des décorations. Les décorations sont stockées dans une *map* qui associe les couples de type $nom, valeur$. Les exemples typiques de décorations utilisées lors de la compilation sont des informations sémantiques, les résultats de vérifications déjà effectués, etc.
2. **Une interface propre au type d'élément réifié**. Cette interface donne accès aux informations qui sont associées à l'élément représenté par le nœud. Par exemple, dans le cas d'un nœud qui représente un composant décrit avec FRACTALADL, cette interface permet d'accéder au nom du composant, ainsi qu'à sa définition. L'accès à ces informations s'effectue via des opérations `get` et `set`.
3. **Des interfaces de navigation au sein de l'AST**. Chaque nœud peut être un conteneur de sous éléments ; les interfaces de navigation fournissent la possibilité de parcourir l'AST (accéder aux fils d'un nœud). Si l'on reprend l'exemple d'un composant décrit à l'aide de FRACTALADL, le nœud qui le réifie peut fournir des interfaces permettant d'accéder aux nœuds réifiant ses interfaces, ses attributs et ses sous-composants. Remarquons que ces interfaces de conteneur sont propres à chaque type de sous-éléments et qu'elles fournissent non-seulement des opérations de consultations (i.e. `get`) mais aussi des opérations de mise-à-jour (`set` et `add`) pour permettre l'enrichissement de l'AST au cours de son analyse et de son traitement.

Le fait que les nœuds de l'AST exposent des interfaces différentes à leur environnement permet une séparation entre leur implantation et leur protocole d'utilisation. Ceci est un élément clé pour garantir l'extensibilité de l'AST. En effet, prenons le cas d'une extension de l'AST visant à étendre un type de nœud en lui ajoutant un nouveau type de sous-élément. Il est possible de donner accès à ce type de sous-éléments en définissant une interface spécifique, tout en conservant les autres interfaces du nœud. En conséquence, la modification est faite de façon transparente vis-à-vis des parties de l'outil qui ne sont pas concernées par cette extension.

La figure 6.2.1 représente l'architecture de l'AST d'un composant décodeur multimédia. Le composant *Player* est un composite contenant trois composants : *GUI* gère l'interface graphique, *MPEG-2* et *H.264* sont des décodeurs vidéo. Etant donnée que le composant *Player* est le composant de plus haut niveau, le nœud représentant ce composant constitue la racine de l'AST. Ce nœud est de type composant, c'est pourquoi il implante une interface `Component` donnant essentiellement accès au nom du composant. Le composant *Player* étant un composite, son nœud dans l'AST fournit deux interfaces de type conteneur permettant respectivement de naviguer vers les sous-composants et d'accéder aux nœuds représentant les liaisons définis au sein du composant *Player*. Les nœuds représentant les composants *GUI*, *MPEG-2* et *H.264* sont similaires à celui du *Player*. Cependant, ils implantent des conteneurs d'interfaces pour donner accès à leurs interfaces. Ces interfaces sont représentées par des nœuds implantant une interface spécifique, appelée `Interface`, qui donne accès au propriétés de l'interface (e.g. signature, nom, rôle, etc.).



Afin d'illustrer un exemple d'extension de l'AST, considérons une extension de l'ADL qui permettrait de rajouter des éléments de commentaires aux définitions de composants (voir la section 4.2.2.3). Pour intégrer une telle extension, une solution possible est de représenter les commentaires par le biais d'un nouveau type de nœud dans l'AST. Il est donc nécessaire de définir une nouvelle interface, propre à ce type de nœuds, dont le rôle est de donner accès au texte du commentaire. Afin d'intégrer ce nouveau type de nœud dans l'AST, il est nécessaire de définir une interface de conteneur permettant de lier

les nœuds de commentaires aux nœuds de composants. De façon plus précise, les nœuds représentant des composants doivent implanter une interface, appelée `CommentContainer`, qui permet la navigation vers les nœuds de commentaire.

6.2.2 Usine de nœuds spécialisable

Dans cette section, nous présentons un outil permettant aux concepteurs de développer des extensions de l'AST de façon aisée. Cet outil permet de générer des nœuds pour la construction d'AST, sans que les développeurs aient beaucoup de code à écrire à la main. Nous détaillons le fonctionnement de cet outil et discutons son apport concernant la facilité d'extensibilité de l'AST.

La figure 6.2 illustre le fonctionnement de l'*usine de nœuds* spécialisable [3]. Cette *usine* fournit une opération de création de nœuds qui prend en paramètre le nom du type de nœud à instancier, et retourne un objet correspondant à ce dernier. Les règles de construction des nœuds sont spécifiées dans un fichier de règles décrivant la grammaire de l'AST. Ces fichiers de description permettent en quelque sorte de spécialiser l'usine : il suffit d'étendre le fichier de règles pour spécialiser l'AST généré.

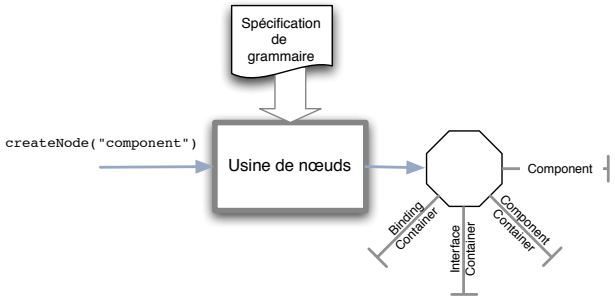


FIG. 6.2 – Illustration du fonctionnement de l'*usine de nœuds*.

Le langage de description de grammaires utilisé est basé sur le langage XML-DTD. La figure 6.3 illustre une version simplifiée des règles de description utilisées pour spécifier l'implantation d'un nœud de composant FRACTAL. La première ligne de cette description précise que tous nœuds de type `component` implante l'interface `Component` qui donne accès aux propriétés du composant modélisé. Les trois lignes suivantes définissent ces propriétés : un composant doit avoir un nom et, de façon optionnelle, une définition. A partir de la ligne 5, sont définis les fils que peut accepter un nœud `component`. Les trois premières lignes spécifient les types d'interfaces de conteneur qui doivent être implantés par un nœud `component`. La ligne 8 précise que ce type de noeud peut accepter un nombre quelconque de fils de types interface, liaison et sous-composant.

La figure 6.4 illustre deux exemples d'interface. L'interface `Component` fournit des opérations de consultation et de mise-à-jour des attributs d'un composant, alors que l'interface `ComponentContainer` définit des opérations permettant de consulter, d'ajouter ou de supprimer des fils de type `C`omponent. Notons qu'il serait possible de générer automatiquement le code de ces interfaces à partir des données spécifiés pas les constructions `ATTLIST` et `ELEMENT`. Cependant, cette fonctionnalité n'est pas supportée à ce jour. Par conséquent, les programmeurs doivent coder ces interfaces à la main en respectant des conventions de nommage.

Nous pouvons remarquer que le langage de spécification de grammaires utilisé pour programmer l'*usine de nœuds* est le même que celui qui est utilisé pour spécifier le langage FRACTALADL. En effet, comme la syntaxe de FRACTALADL est du XML, cette spécification permet de programmer à la fois

[3] L'implantation de cette usine est inspirée du *parseur* d'ADL de l'usine FRACTALADL.

```
1.  <?add ast="component" itf="adl.interfaces.Component"?>
2.  <!ATTLIST component
3.    name CDATA #REQUIRED
4.    definition CDATA #IMPLIED>

5.  <?add ast="component" itf="adl.interfaces.InterfaceContainer"?>
6.  <?add ast="component" itf="adl.interfaces.BindingContainer"?>
7.  <?add ast="component" itf="adl.interfaces.ComponentContainer"?>

8.  <!ELEMENT component (interface*, binding*, component*) >
```

FIG. 6.3 – Extrait de spécification de grammaire pour la génération des nœuds représentant des composants FRACTAL.

```
public interface Component {
  String getName() ;
  void setName(String name) ;
  String getDefinition() ;
  void setDefinition(Dtring definition) ;
}

public interface ComponentContainer {
  Component[] getComponents() ;
  void addComponent(Component c) ;
  void removeComponent(Component c) ;
}
```

FIG. 6.4 – Définition des interfaces `Component` et `ComponentContainer` implantées par les nœuds de l'AST représentant des composants.

le *parseur* et le mécanisme de construction d'AST. Ce n'est pas le cas dans d'autres langage comme l'IDL de THINK qui est basé sur une syntaxe semblable à Java. Dans ce cas, notre mécanisme est utilisé uniquement pour l'instanciation des nœuds d'AST. Il doit être dirigé par un *parseur*, généré par d'autres outils, comme, par exemple, le compilateur *JavaCC*.

6.2.3 Intégration de langages hétérogènes

Rappelons qu'une de nos motivations principales est de supporter l'utilisation de langages hétérogènes (e.g. ADL, IDL, DSL). Dans cette section, nous expliquons comment il est possible d'utiliser l'AST comme une plate-forme homogène permettant de fusionner les descriptions faites à l'aide de ces différents langages.

Nous présentons ces mécanismes de fusion de descriptions par l'exemple de deux langages : le langage de définition d'interfaces (IDL) de THINK et l'ADL de FRACTAL. La figure 6.5 reprend une partie de l'architecture du décodeur multimédia présenté précédemment. Est ajoutée la définition de la signature d'interface `VideDecoderInterface`. Les nœuds en blanc de l'AST représentent les éléments décrits en FRACTALADL, alors que les nœuds en gris représentent les éléments de définition d'interfaces.

Les ASTs de l'ADL et de l'IDL sont construits séparément par des composants appropriés du compilateur. Afin d'achever la construction de l'AST pour les définitions d'interfaces, nous mettons en place une deuxième instance d'*usine de nœuds* spécialisée par un fichier de spécification qui décrit la grammaire de l'IDL. Cette usine est utilisée par le *parseur d'IDL* pour construire les instances d'AST qui représentent les interfaces définies.

L'élément clé pour effectuer la fusion entre l'AST de l'ADL et celui de l'IDL consiste en la définition d'un point d'intersection qui lie ces deux types de description. Dans le cadre de notre exemple, cette

```
<component name="GUI">
 <interface name="MPEG2" role="client"
            signature="VideoDecoderInterface"/>
 <interface name="H264" role="client"
            signature="VideoDecoderInterface"/>
</component>

public interface VideoDecoderInterface {
  Stream     getStream() ;
  int        getLength() ;
  StreamType getType() ;
}
```

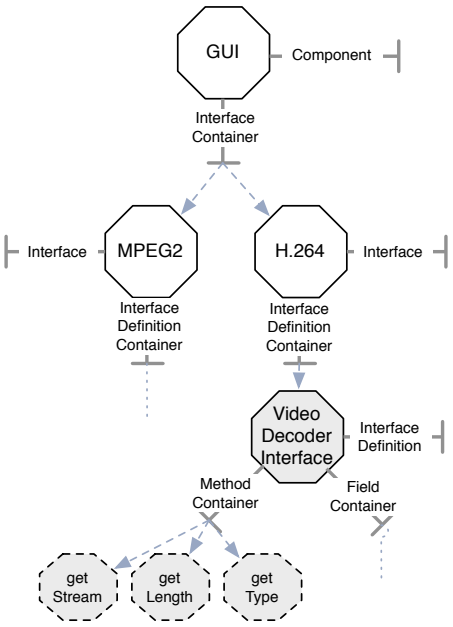


FIG. 6.5 – Extrait d'AST illustrant la fusion de FRACTALADL et de THINKIDL.

fusion consiste en l'ajout d'un sous-arbre d'AST décrivant la définition écrite en IDL dans l'AST de l'architecture (i.e. ADL) au niveau des nœuds correspondant aux interfaces. Afin d'adapter l'AST de l'ADL de manière à prendre en compte ce point d'intersection, une extension de la grammaire de ce dernier doit être fait par le concepteur. La figure 6.6 illustre les deux lignes qui doivent être ajoutées dans la grammaire de l'AST de l'ADL, afin de définir le point de fusion avec l'AST de l'IDL. La première ligne de cette extension spécifie l'interface conteneur qui permettra d'accepter la définition d'interface comme étant un fils d'un nœud de type *interface* (dans l'AST de l'ADL) ; la deuxième ligne précise que ce dernier ne peut accepter qu'un seul fils de type définition d'interface.

```
<?add ast="interface" itf="adl.ItfDefContainer"?>
<!ELEMENT interface (ItfDef?) >
```

FIG. 6.6 – Extension à porter dans la grammaire de l'AST de l'ADL pour la fusion de l'AST de l'IDL.

En résumé, notre approche pour fusionner des descriptions *hétérogènes* d'architectures au sein d'un même AST consiste en l'utilisation de différentes instances d'*usines de nœuds* spécialisées pour chacun des langages d'entrée. Ensuite, nous proposons de définir des points d'intersection, en étendant la grammaire de ces ASTs, qui permettent d'effectuer la fusion des AST de manière systématique.

6.3 Construction de l'arbre de syntaxe abstraite

Cette section décrit l'architecture du module de chargement introduit dans la section 6.1. Nous commençons par la présentation de l'architecture de ce composant. Nous présentons ensuite les patrons de programmation utilisés pour implanter les analyseurs syntaxiques et sémantiques.

6.3.1 Architecture du module de chargement

Le module de chargement (appelé *loader*) accepte en entrée des fichiers de descriptions d'architectures afin de produire un AST. Comme nous l'avons illustré sur la figure 6.7, le module de chargement est composé d'une chaîne de composants. Chacun de ces composants implante une fonction de transfor-

mation ou d'analyse spécifique. Le flot d'exécution de cette chaîne de composants débute par la requête de chargement d'un AST reçue par le composant se trouvant en fin de chaîne. Cette requête déclenche un appel de procédure en chaîne au cours de laquelle chaque composant de niveau intermédiaire demande à son prédecesseur le chargement d'un AST. Une fois qu'un composant reçoit l'AST retourné par son prédecesseur, il effectue sa fonction d'analyse ou de transformation et retourne le résultat à son successeur. Cette récursion s'arrête au niveau du premier composant de la chaîne qui est obligatoirement un *parseur* dont le rôle est de lire un fichier de description et de retourner l'AST correspondant.

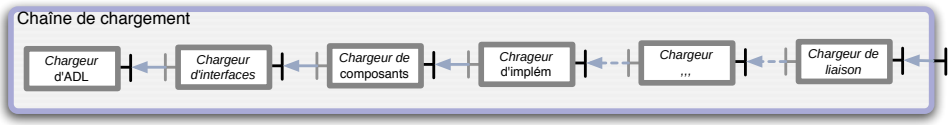


FIG. 6.7 – Architecture du composant *loader* qui est organisé comme une chaîne de composants à grain-fin.

Remarquons que bien que la chaîne de *loader* puisse charger des fichiers écrits dans des langages différents, le processus de chargement est toujours dirigé par un seul langage (celui qui est chargé par le premier composant de la chaîne). Dans le cadre de nos expérimentations, ce langage est FRACTALADL ; celui-ci est utilisé pour décrire l'organisation structurelle des composants.

L'organisation du module de chargement sous forme de chaîne de composants à grain fin permet de l'étendre facilement. En effet, il est facile d'identifier l'endroit de la chaîne auquel il faut effectuer des modifications afin d'ajouter des fonctionnalités. Ces modifications peuvent prendre la forme de remplacement, insertion, ou suppression de composants. Notons que le choix du niveau d'insertion est important : en effet, ce niveau définit les vérifications qui auront été faites sur l'AST avant que le composant inséré soit exécuté. Par exemple, un composant vérifiant que toutes les interfaces clientes sont correctement liées doit être inséré dans la chaîne à un point qui lui garantit que l'AST qu'il recevra comportera les informations adéquates sur l'architecture des composants et le type de leurs interfaces.

Enfin, remarquons que l'organisation en chaîne a des influences diverses sur le traitement des erreurs. Lorsqu'un composant détecte une erreur, il arrête le chargement de l'AST et lève une exception qui est propagée vers ses prédecesseurs en suivant le sens inverse des invocations. Cette architecture simplifie la programmation des composants *loader* puisqu'ils sont sûrs de recevoir un AST correct. Cela constitue un avantage. En revanche, le fait qu'un composant de bas niveau puisse arrêter l'exécution des composants de niveaux supérieurs rend difficile la poursuite du chargement en cas d'erreurs (il est notamment difficile de poursuivre le chargement pour pouvoir retourner des messages d'erreurs multiples).

Dans la suite de cette section, nous décrivons les deux types de composants d'analyse contenus dans le module de chargement : les analyseurs syntaxiques et les analyseurs sémantiques.

6.3.2 Composants d'analyse syntaxique

Les composants d'analyse syntaxique (appelés *parseurs*) sont en charge de lire un fichier en entrée pour construire un AST. Ils sont propres à un langage donné. Comme illustré sur la figure 6.8, ces composants sont des composites comprenant deux sous-composants : une *usine de nœuds* et le lecteur de fichiers d'entrée qui dirige la construction de l'AST.

Les composants de lecture de fichiers peuvent être programmés en utilisant des outils de haut niveau. Dans le cas des langages basés sur XML, il existe de nombreux outils, de type DOM [DOM]. Pour parser des fichiers dont la syntaxe est autre que XML, on peut utiliser des outils de génération de *parseurs* comme *JavaCC* [Jav], ou *lex/yacc* [Yac]. Ces outils permettent de générer automatiquement

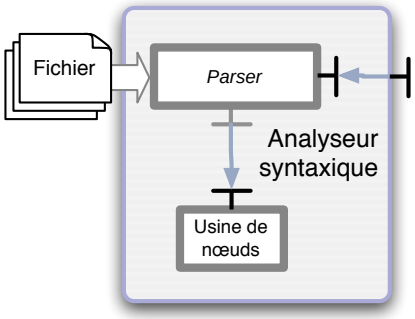


FIG. 6.8 – Architecture des composants d'analyse syntaxique.

l'implantation d'un *parseur* à partir des règles syntaxiques sous forme de LL(1).

A l'exception du *parseur* spécifique au langage qui dirige le chargement (i.e. le composant qui est au bout de la chaîne), tous les analyseurs syntaxiques reçoivent un AST en entrée et ont la responsabilité de fusionner l'AST obtenu par la lecture du fichier avec ce dernier. Ceci est par exemple le cas du *parseur* de THINKIDL qui construit des ASTs de définition d'interfaces et qui les insère dans l'AST d'architecture reçu en entrée (voir la figure 6.5).

6.3.3 Composants d'analyse sémantique

Les composants d'analyse sémantiques peuvent être considérés comme des filtres recevant un AST en entrée et le retournant après avoir effectué des vérifications (et éventuellement des modifications). Parmi les fonctions d'analyse sémantique de base implantées dans notre chaîne d'outils, citons la vérification de compatibilité entre les interfaces interconnectées, la vérification des propriétés de surcharge ou d'extension de définitions de composants pour le langage FRACTALADL, et la vérification de l'établissement de connections pour les interfaces clientes.

Un autre type d'analyse sémantique consiste en l'enrichissement ou en la modification de l'architecture de l'AST. Un exemple d'enrichissement est la mise en place d'adaptateurs de communication pour établir des liaisons entre des composants implantés dans des langages différents. Par exemple, un analyseur peut décider d'établir un pont JNI (*Java Native Interface*) entre deux composants implantés en C et Java respectivement. Ceci est effectué en insérant deux composants au niveau de la liaison entre ces composants. Ces composants vont respectivement jouer le rôle de *stub* et de *skeleton*. Cette modification est effectuée au niveau de l'AST. Une *usine de nœuds* d'AST est utilisée pour instancier de nouveaux nœuds lors de l'analyse. Nous détaillons cet exemple dans la section 7.3.1.

6.4 Traitement de l'arbre de syntaxe abstraite

Cette section présente la façon dont l'AST est traité afin de réaliser l'ensemble des tâches qui sont attendues de la chaîne d'outils. Nous commençons par introduire l'architecture générale du module de traitement de l'AST. Nous décrivons ensuite en détail les différents composants du module.

6.4.1 Architecture du module de traitement

Le rôle du module de traitement est de lire une description d'architecture exprimée sous forme d'AST est d'exécuter des opérations de traitement correspondant à la fonction supportée par le compilateur ADL. Le comportement implanté par le module de traitement s'exécute en deux temps. Le premier consiste en la génération d'un graphe de tâches modélisant l'organisation des opérations à exécuter.

Le deuxième consiste en l'exécution des opérations contenues dans le graphe de tâches dans un ordre correct. Afin de mieux comprendre la fonction implantée par le module de traitement, nous pouvons faire une analogie avec l'outil de compilation et de déploiement intitulé Ant [Ant]. Dans cette perspective, la première passe correspondrait à l'écriture des fichiers de *scripts* définissant les tâches à exécuter avec des règles de dépendances et la deuxième correspondrait à l'exécution des tâches par Ant.

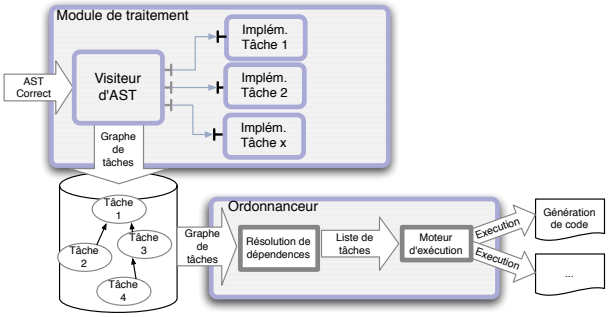


FIG. 6.9 – Flot d'exécution du module de traitement.

Par souci de modularité et d'extensibilité, le module de traitement est conçu comme une architecture à base de composants. La figure 6.9 présente les composants principaux de cette architecture. L'élément centrale du module de traitement est le graphe de tâches construit à partir de l'AST par le module de construction de tâches (i.e. visiteurs d'AST). Ce graphe permet de définir des dépendances unidirectionnelles entre les tâches ainsi que des flots de données qui doivent êtres échangés.

Illustrons ce modèle de graphe dans le cadre d'un générateur de code de composants Java. Considérons qu'un composant est implanté par une classe Java implantant ses interfaces serveurs et déclarant des champs privés pour référencer ses interfaces clientes. La figure 6.10 représente sous forme de grammaire les tâches de génération de code et leurs dépendances. Trois tâches sont identifiées : les deux dernières produisent le code pour les interface serveurs et clientes. Ces tâches prennent en entrée des données architecturales comme le nom (i.e. `ClientItfName`) ou le type (i.e. `ServerItfType`, `ClientItfType`) d'une interface. Ces données se trouvent dans l'AST. Par conséquent, ces tâches n'ont aucune dépendance envers d'autres tâches. La première tâche est responsable de la génération de code du type de composant. Elle utilise non-seulement une information architecturale pour le type du composant (i.e. `CompType`), mais également le code qui est généré par les deux tâches précédentes (dont elle dépend de fait).

CompDefinition	→	`public abstract class` CompType
		`implements` *ServerItfs* `{`
		ClientItfs
		`}`

ServerItfs	→	ServerItfType_1 {, ServerItfType_i}*
ClientItfs	→	{`private` ClientItfType ClientItfName `;`}*

FIG. 6.10 – Règles de génération de code pour l'implantation de composants Java. Les mots soulignés représentent les données obtenues à partir de l'AST, alors que les mots en italique représentent les données qui sont produites par l'exécution d'autres règles.

L'implantation concrète des tâches est fournie par les composants *backend*. Par analogie aux outils existants, les composants *backends* correspondent aux tâches Ant ou encore à JDT [JDT] qui implante des

fonctions de génération de code dans l'environnement Eclipse. Ces composants permettent d'implanter divers comportement pour une tâche donnée en fonction de l'environnement cible. Par exemple, nous montrons dans le chapitre suivant qu'un même graphe de tâches peut être utilisé pour la génération de code en C, C++ et Java en modifiant les composants *backends* utilisés.

Enfin, le graphe de tâches est exécuté par un composant *ordonnanceur*. Celui-ci exécute les tâches en respectant les dépendances entre celles-ci. Par ailleurs, il assure leurs échanges de données.

6.4.2 Canevas de tâches

Cette section est dédiée à la présentation du canevas de tâches utilisé dans le module de traitement. Ce canevas est utilisé pour l'organisation de l'exécution des opérations de traitement. Sa fonction peut être comparée à des outils de compilation tel que *Ant* ou *Make*. Il définit un modèle de programmation pour décrire un graphe de tâches et implante un moteur d'exécution, appelé ordonnanceur, pour exécuter les opérations contenues dans le graphe. Le modèle de programmation fourni permet de définir des types de tâches, similairement à *Ant*. Or, si ce dernier permet de dénoter uniquement des dépendances, notre canevas permet aussi de définir des flots de données typés qui seront échangés entre les tâches.

Nous commençons par présenter les différents éléments qu'inclut ce canevas. Ensuite, nous présentons la structure des tâches et nous détaillions les facilités fournies pour la programmation du graphe de tâches.

6.4.2.1 Elements du canevas de tâches

Le canevas de tâches est constitué de trois principaux éléments :

- **Les tâches** sont les éléments de base du canevas. Elles sont utilisées pour modéliser une action de traitement et sont connectées à des composants *backend* fournissant l'implantation concrète de l'action à réaliser. Chaque tâche implante une méthode `execute` qui permet de lancer de manière unifiée l'action implantée par le *backend* associé à la tâche. En plus de cette interface, les tâches peuvent implanter des interfaces spécifiques au type d'action qu'elles modélisent. Au travers de ces interfaces spécifiques, les tâches peuvent accéder aux informations se trouvant dans l'AST et aux résultat des tâches dont elles dépendent.
- **Le graphe de tâches** permet de modéliser l'ensemble des tâches à réaliser par le module de traitement. Il s'agit d'un graphe acyclique qui permet de représenter les tâches et leurs interdépendances. Il fournit des opérations d'ajout et de suppression de tâches, ainsi que de spécification des interdépendances. Remarquons que le graphe vérifie après chaque modification qu'aucun cycle n'est créé. En effet, cette propriété est cruciale pour garantir que le graphe pourra être exécuté.
- **L'ordonnanceur** est en charge d'exécuter les tâches du graphe en respectant leurs interdépendances. Pour ce faire, il implante une méthode, appelée `execute`, prenant en paramètre un graphe de tâches à exécuter. Notons qu'il est possible de réaliser diverses implantation de ce composant.

6.4.2.2 Définition de types de tâches

Chaque tâche est représentée par un objet dérivant d'une classe, appelée `Task`, qui implante diverses méthodes :

- Les méthodes `addDependancy(anotherTask,role)` et `removeDependancy(anotherTask)` permettent respectivement d'ajouter et de supprimer des dépendances vers une autre tâche. Le paramètre `anotherTask` identifie la tâche vers laquelle la dépendance doit être ajoutée/supprimée, alors que le paramètre `role` est un identifiant de rôle qui permet d'associer à la dépendance un type bien précis. Le méchanisme d'attribution de rôles peut être, par exemple, utilisé pour distinguer les codes sources fournis pour les interfaces clientes et serveurs dans l'exemple présenté dans la section 6.4.1.

- La méthode `execute(context)` permet de lancer l'exécution de la tâche. Le paramètre `context` permet de passer des informations sur le contexte d'exécution à la tâche.
- La méthode `getResult()` retourne le résultat d'exécution de la tâche. Le résultat de retour n'est pas typé au niveau de cette interface générique ; cependant il est possible de le *caster* dans le type de résultat attendu.

Pour spécialiser un type de tâches, il suffit de définir de nouvelles interfaces étendant le type de base présenté ci-dessus. Par exemple, dans le cadre de l'exemple présenté en section 6.4.1, il est nécessaire de définir deux type de tâches modélisant les actions de génération de code. La figure 6.11 présente la définition de ces types. La première interface modélise une tâche de production de code source. Cette tâche est utilisée pour produire les codes sources des interfaces clientes et serveurs. La deuxième interface, appelée `SourceCodeProviderConsumerTask`, modélise une tâche produisant et consommant du code source. Ce type de tâche est celui qui est utilisé pour créer les définitions de type de composants. L'interface définit tout d'abord un rôle qui permet de modéliser les interactions entre la tâche possédant l'interface et les tâches dont elle va dépendre. Plus précisément, il est spécifié qu'une tâche de type SourceCodeProviderConsumerTask peut dépendre de producteurs de codes (SourceCodeProviderTask). L'interface spécifie également deux méthodes qui permettent de rajouter et supprimer des dépendances envers ces producteurs de code.

```
public interface SourceCodeProviderTask extends Task{
  /**
   * Retourne le code source produit par cette tâche.
   */
  Object getSourceCode();
}

public interface SourceCodeProviderConsumerTask extends SourceCodeProviderTask{
  /**
   * Le paramètre 'role' est utilisé pour identifier une dépendance de code source.
   */
  Class SOURCE_CODE_PROVIDER_TASK_ROLE = SourceCodeProviderTask.class;

  /**
   * Ajout d'une nouvelle dépendance de code production de code
   * source en lui associant le nom  donnée par 'codeProviderName'.
   */
  void addSourceCodeProviderTask(String codeProviderName, TaskMap.PlaceHolder task);

  /**
   * Suppression de la tâche identifié par 'codeProviderName'.
   */
  void removeSourceCodeProviderTask(String codeProviderName);
}
```

FIG. 6.11 – Définition de types de tâches pour modéliser (1) la production de code et (2) la production et consommation de code.

Le canevas de tâche peut donc être étendu par les concepteurs du compilateur d'ADL afin de définir des types de tâches répondant à leurs besoins. Dans le cadre de nos travaux, nous avons, entre autres, étendu ce canevas afin de mettre en place un outil de traitement spécialisé pour la génération de code et la compilation. Celui-ci est décrit en détail dans le chapitre suivant.

6.4.2.3 Déclaration de dépendances entre tâches

Comme nous l'avons expliqué dans la section 6.4.1, divers composants peuvent participer à l'élaboration du graphe de tâches. Afin de déclarer des dépendances entre tâches, ces composants doivent

connaître leurs identifiants. Il n'est pas possible de prendre pour identifiant de tâches les références vers les objets les représentant. En effet, un tel choix imposerait que les tâches soient crées avant de pouvoir déclarer des dépendances vers elles. Ceci n'est pas possible dès lors que le graphe peut être construit par des composants distincts.

Afin de remédier à ce problème, nous proposons un mécanisme permettant de découpler l'instanciation des tâches de la déclaration de leurs dépendances. Ce mécanisme, appelé *mécanisme d'emplacement* (*PlaceHolder*), permet de référencer les différentes tâches du graphe sans connaître les objets qui les représentent. Pour ce faire, les nœuds du graphe de tâches ne sont pas des tâches, mais des "emplacements" référençant ces tâches. Ainsi, lors qu'un composant crée une tâche `T1` et veut déclarer une dépendence entre cette dernière et une tâche `T2`, il consulte le graphe de tâches pour connaître l'emplacement de la tâche `T2`. Si la tâche `T2` a préalablement été créée et enregistrée, l'emplacement retourné est déjà rempli. Sinon, un emplacement vide portant le nom `T2` est retourné. Cette emplacement sera rempli plus tard par un autre composant de construction du graphe de tâches. Notons que le module d'exécution du graphe ne débute son exécution que si l'ensemble des emplacements a été rempli.

6.4.3 Construction du graphe de tâches

La construction de graphes de tâches consiste en la définition des opérations à exécuter pour fournir les fonctions attendues du module de traitement. Si l'on reprend l'analogie avec les outils de compilation classiques comme *Ant*, il s'agit d'écrire un équivalent des fichiers de *script* décrivant le graphe de tâches à exécuter. Ce processus est effectué automatiquement de manière à générer le graphe de tâches à partir de l'interprétation des données architecturales contenues dans l'AST.

Dans cette section, nous décrivons la structure du module de construction qui prend en entrée l'AST produit par le module de chargement, et qui produit en sortie un graphe de tâches. Nous débutons notre présentation par la description de l'architecture du module de construction qui est basée sur le patron de programmation *visiteur*. Nous présentons ensuite le modèle de programmation générique employé pour implanter les visiteurs.

6.4.3.1 Architecture du module de construction du graphe de tâches

Étant donnée que le module de construction de graphe de tâches est en charge de définir les opérations de traitement implantées par un compilateur ADL donné, sa modularité et son extensibilité est un prérequis pour la capacité à intégrer de nouvelles fonctions de traitement. Dans cet objectif, nous optons pour une architecture à base de composants à grain fin où chaque composant implante une fonction de traitement spécifique au travers le patron de programmation *visiteur*. Par assurer la généricité d'une telle architecture, nous définissons trois types de composants nécessaires pour le parcours et l'interprétation de l'AST.

- **Les composants voyageurs** implantent une interface *Traveler* et possèdent une interface cliente de type *Visitor*. L'interface *Traveler* définit une méthode `travel(AST)`, dans laquelle le paramètre `AST` désigne la racine de l'AST qui doit être traversé par le composant. Le composant voyageur effectue un parcours en profondeur de l'AST et utilise son interface client de type *Visitor* pour déclencher un parcours de chaque nœud composant trouvé.
- **Les composant expéditeurs** permettent d'adapter une interface de type *Visitor* en n interfaces du même type. Ces composants permettent de diffuser une invocation en provenance d'un voyageur vers plusieurs composants visiteurs. Dans certains cas, les expéditeurs peuvent être utilisés pour sélectionner un composant visiteur particulier, à l'aide d'un critère de choix.
- **Les composants visiteurs** implantent l'interface *Visitor* qui est invoquée par les composants voyageurs. Cette interface définit une méthode `visit(component)` dont le rôle est de créer des tâches et d'enregistrer les dépendances de ces tâches vis-à-vis d'autres tâches. Les composants visiteurs

sont connectés à des composants *backend* qui fournissent l'implantation concrète des tâches crées.

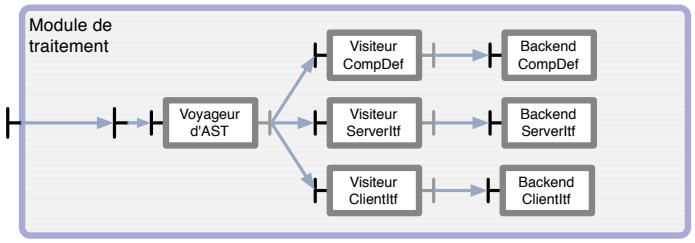


FIG. 6.12 – Architecture d'un module de traitement permettant de créer un graphe de tâches pour la fonction de génération de code présentée sur la figure 6.10.

La figure 6.12 illustre l'utilisation des différents composants du module de construction de graphe dans le cadre de l'exemple présenté sur la figure 6.10. Le module de traitement est un composant composite encapsulant un composant *voyageur*, connecté à un *expéditeur*, lui même connecté à trois *visiteurs*. Le composant *voyageur* effectue un parcours en profondeur de l'AST et invoque l'expéditeur à chaque fois qu'il rencontre un nœud composant. L'*expéditeur* diffuse les invocations du *voyageur* aux trois *visiteurs* auxquels il est connecté. Chaque *visiteur* implante une des trois règles de génération de code présentée dans la figure 6.10. Lorsqu'un *visiteur* est invoqué, il crée une tâche adéquate et déclare les dépendances de cette tâche envers les autres tâches. Sa connexion vers un composant *backend* lui permet d'associer les tâches créées à une implantation concrète.

Notons que l'ordre dans lequel les *visiteurs* sont connectés à l'*expéditeur* influence l'ordre dans lequel ils sont exécutés par ce dernier. Par conséquent, l'ordre de la création des tâches et de l'enregistrement de leurs dépendances est fonction de l'organisation des connexions des *visiteurs*. Néanmoins, cet ordre n'a aucune incidence du fait du mécanisme d'emplacement présenté dans la section 6.4.2.3.

Enfin, notons que le seul composant dont la présence est imposée est le composant *voyageur*, dont le rôle est de fournir une implantation de l'interface *Traveler*. Il est, en revanche, possible de bâtir des organisations d'expéditeurs/visiteurs arbitrairement complexes. Dans le chapitre suivant, nous décrivons plusieurs architectures que nous avons élaborées afin de décliner l'outil de traitement d'ADL spécialisé pour la génération de code en un générateur de code multi-cibles.

6.4.3.2 Programmation des visiteurs

Dans cette section, nous illustrons la programmation des *visiteurs* au travers de l'exemple de la figure 6.10. La figure 6.13 présente un extrait de pseudo-code du composant visiteur assurant la création de tâches pour la définition de type de composants. La méthode `visite` implanté par ce composant prend deux arguments en paramètres : le paramètre `componentNode` donne accès au nœud de l'AST pour lequel la méthode est invoquée ; le paramètre `taskGraph` donne accès au graphe de tâches en cours de construction. La méthode commence par créer une tâche correspondant à la définition d'un type de composant. Pour ce faire, elle récupère le type de composant dans l'AST (ligne 7). Ensuite, l'interface du composant *backend* est enregistrée auprès de la tâche en utilisant la connexion que le visiteur a vers ce composant. La méthode recherche ensuite les interfaces serveurs et clientes du composant, afin d'enregistrer des dépendances envers les tâches qui leurs sont associées. La ligne 13 récupère la liste des interfaces serveurs du composant. Ensuite, le graphe de tâches est consulté pour retrouver les emplacements des tâches associées à chacune des interfaces. Les lignes 21/22 correspondent à l'enregistrement des dépendances vers les tâches en charge des interfaces. Les mêmes opérations sont répétées pour enregistrer les dépendances vers le code produit pour les interfaces clientes. Une fois toutes les dépendances

enregistrées, la tâche créée par ce visiteur est rajoutée dans le graphe de tâches (ligne 30) en lui associant un nom et la référence du nœud d'AST pour lequel elle est créée.

```
1.  procedure compDefVisit(ComponentNode compNode, TaskGraph taskGraph) {
2.   // Creation d'une tâche de définition de type de composant
3.   CompDefTask compDefTask = new CompDefTask() ;
4.
5.   // Affectation du nom de composant en utilisant l'information
6.   // se trouvant sur l'AST
7.   compDefTask.setCompType(compNode.getType());
8.
9.   // Affectation de l'implantation concrète à la tâche créée
10.  // en utilisant la référence du backend qui est connectée
11.   compDefTask.setImplementationBackend(myBackendInterface) ;
10.
12.   // Trouver les interfaces serveurs du composant
13.   ServItf[] servItfs = compNode.getServerInterfaces();
14.
15.   foreach (servItf in servItfs) {
16.    // Trouver la tâche associé à cette interface serveur du composant
17.    servItfTaskPH = taskGraph.getTaskPlaceHolder(servItf.getName(),
18.                                                 compNode) ;
19.    // Enregistrer une dépendance envers la tâche trouvée pour obtenir
20.    // le code source qu'elle produit.
21.    compDefTask.addDependency(servItfTaskPH,
22.                              SrcCodeProvider) ;
23.   }
24.
25.   // De même pour l'enregistrement des dépendances envers le code
26.   // produit pour les interfaces clientes.
27.   ...
28.
29.   // Enregistrer la tâche créée dans le graphe des tâches
30.   taskGraph.registerTask("CompDef",
31.                          compNode,
32.                          compDefTask);
33.  }
```

FIG. 6.13 – Implantation en pseudo-code du composant visiteur qui organise la génération de code pour la définition du type de composant conformément à l'exemple 6.10.

6.4.4 Implantation des tâches

Dans cette section, nous décrivons la façon dont les tâches sont implantées au sein des composants *backend*. Rappelons que ces derniers fournissent les comportements concrets des opérations modélisés par des tâches. Naturellement, plusieurs composants *backend* peuvent êtres associés à une tâche afin de décliner cette dernière vers des opérations variées. Typiquement, deux jeux différents de composants *backend* peuvent êtres associés à un même jeu de tâches de génération de code afin de spporter la génération de code dans des langages de programmation différents, tels que le C et le C++.

Pour illustrer le modèle de programmation de tels composants, nous prenons l'exemple de la tâche de définition de type de composants Java spécifiée sur la figure 6.10. La figure 6.14 illustre un extrait de code du composant *backend* implantant cette tâche. La méthode `ComponentDefinitionJavaBackend` est invoquée par l'ordonnanceur afin de démarrer l'exécution de la tâche. Cette méthode admet deux paramètres : le paramètre `compType` correspond au type de composants enregistré par le visiteur ; le paramètre `codePieces` donne accès aux codes sources qui sont produits par les tâches dont celle-ci dépend. La méthode commence par créer un objet `CodeWriter` qui permet d'écrire du code source de manière formatée. La méthode écrit ensuite la définition de la classe Java (ligne 4), le mot clé pour énumérer les noms des interfaces serveurs (ligne 5) et les noms de ces interfaces (ligne 6). Une séquence similaire

d'opérations est effectuée pour écrire les champs correspondant aux interfaces clientes du composant. La méthode retourne le texte qui a été produit, de sorte qu'il pourra être utilisé par les tâches qui en dépendent.

Cet exemple a permis de constater que l'implantation d'un générateur de code ne nécessite que quelques lignes de Java. Ceci vient du fait qu'une architecture de composants à grain fin est opté pour la construction des compilateurs ADL. Cette granularité fine présente l'avantage de faciliter le reciblage des tâches dans le but de supporter de nouveaux comportements concrets de traitement.

```
1. public String ComponentDefinitionJavaBackend(String compType,
2.                                               Map<String,String> codePieces) {
3.   CodeWriter cw = new CodeWriter();
4.   cw.append("public␣abstract␣class␣").append(compType);
5.   cw.append("␣implements␣");
6.   cw.append(codePieces.get("server-interfaces"));
7.   cw.appendln("{");
8.   cw.appendln(codePieces.get("client-interfaces"));
9.   cw.appendln("}");
10.  return cw.toString() ;
11. }
```

FIG. 6.14 – Implantation du générateur de code pour la définition des composants.

6.5 Mécanismes d'extension des compilateurs ADL

Dans les sections précédentes, nous avons présenté l'architecture et les patrons de programmation utilisés pour la conception de notre canevas logiciel de construction de compilateur ADL. Cette dernière section est consacrée à la présentation des mécanismes d'extension que nous avons implantés afin de permettre aux concepteurs d'étendre de manière systématique les compilateurs élaborés. Le canevas a été développé à l'aide de la version Java du modèle de composants FRACTAL. L'architecture de chaque compilateur ADL mis au point est donc décrite à l'aide du langage FRACTALADL. Une façon systématique de modifier un compilateur est donc de modifier sa description ADL.

La modification via l'ADL présente un certain nombre de limitations. En particulier, dans le cadre de l'utilisation d'un support conjointement avec d'autres équipes de recherche, nous avons observé que la mise à jour des fichiers ADL était fastidieuse. En effet, il n'est pas envisageable (pour des raisons de maintenance) de construire un compilateur intégrant toutes les fonctionnalités. Il est donc continuellement nécessaire de modifier les fichiers ADL. Pour remédier à ce problème, nous avons mis au point un mécanisme de *plugins* permettant de charger dynamiquement et de façon paresseuse des composants dans les supports ADL. Les composants sont uniquement chargés quand la syntaxe des fichiers parsés n'est pas supportée par les composants déjà présents.

Un exemple typique d'utilisation des plugins dans le cadre d'un outil traitement d'ADL spécialiste de la génération de code est le chargement dynamique de composants prenant en charge des langages non supportés par l'outil. Lors de son instantiation, l'outil ne contient aucun module de génération de code. Dès lors que l'ADL chargé fait intervenir la nécessité de générer du code pour de langages donnés, les composants de génération adéquats sont chargés. Par exemple, si l'ensemble d'un logiciel est écrit à l'aide de THINK, le support chargera uniquement le module de génération de code ciblant les logiciels THINK. Si, en revanche, d'autres langages (e.g. C++) sont utilisés, alors d'autres modules de génération (e.g. ciblant C++) seront chargés.

Le mécanisme de plugins est implanté par deux types de composants :

– **Les chargeurs de plugins** sont des sortes d'usines de déploiement ADL qui permettent de déployer dynamiquement les plugins implantés sous forme de composants FRACTAL. La méthode

de chargement mise en œuvre par un chargeur de plugins prend en paramètre le nom de la description ADL décrivant l'architecture d'un plugin. Elle instancie un composant de ce type et retourne son interface serveur appelée `plugin`[4]. Notons que le chargeur de plugins possède un cache afin d'optimiser le temps de chargement des plugins lorsque ceux-ci ont déjà été chargés.

– **Les sélecteurs de plugins** prennent l'initiative de charger des plugins en utilisant un chargeur de plugins. Leur rôle est de rendre transparents au reste du système la présence de plugins. Les sélecteurs peuvent être considérés comme des intercepteurs implantant une interface fonctionnelle identique à celle qui aurait été implantée par le plugin s'il avait été présent. Dès lors que le plugin est chargé, le sélecteur lui transmet les appels interceptés. Par exemple, dans le cas d'un plugin responsable du chargement des contrôleurs, le sélecteur est un composant interceptant l'opération `load` et chargeant le plugin dès que cette méthode est invoquée. Par ailleurs, notons que nous avons implanté deux sélecteurs génériques : le premier est responsable des plugins de la chaîne de chargement ; le second se charge des plugins de traitement de l'AST. Le comportement de ces sélecteurs peut être programmé par le biais d'attributs.

La figure 6.15 illustre l'utilisation du mécanisme de plugins dans le cadre du chargement dynamique de membranes de composants. Un composant *sélecteur de membrane* est inséré dans la partie traitement de l'AST comme étant un composant *visiteur* devant être invoqué pour chaque nœud de type composant. Le sélecteur est connecté à un chargeur de plugins (lié, dans ce cas à une usine de déploiement d'ADL standard). Initialement (instant t_0), une implantation de *membrane* est déjà chargée. A l'instant t_1, le *sélecteur de membrane* est invoqué pour un composant dont la *membrane* est d'un type spécifique, appelé `loggingController`. Le sélecteur procède au chargement du plugin fournissant le visiteur nécessaire à la génération de cette *membrane*. Pour ce faire, le sélecteur transmet au chargeur de plugins l'ADL du visiteur à charger. A l'instant t_2, le chargeur utilise l'usine de déploiement pour charger le visiteur demandé et retourne son interface serveur `plugin`. A l'instant t_3, le sélecteur se lie à l'interface retournée et invoque son opération `visit` pour accomplir l'opération qu'il avait interceptée.

L'utilisation de plugins présente plusieurs avantages. En premier lieu, dans la mesure où les plugins sont chargés de façon paresseuse, il est possible de concevoir une version "exo" de l'outil, uniquement constituée des sélecteurs, des chargeurs de plugins et de quelques composants de base. L'outil est ensuite enrichi au cours des opérations de traitement ultérieures. D'autre part, étant donné que la sélection des plugins est effectuée dynamiquement à l'aide de règles de sélection bien précises, il est possible de déterminer l'usage de plugins particuliers en les plaçant à dessein dans les paquetages où les sélecteurs par convention effectuent leurs recherches. Par ce biais très simple, l'extension de l'outil devient possible sans que le moindre fichier de description de l'outil de base ne soit modifié. Cela permet, entre autres, de développer des extensions dans des équipes de développements distinctes et de les utiliser complètement séparément. Enfin, le mécanisme de plugins que nous avons présenté est généraliste : différentes implantations de chargeurs et de sélecteurs sont permises. Ainsi, les plugins peuvent être utilisés à divers endroits de la chaîne d'outils.

6.6 Conclusion

Dans ce chapitre, nous avons présenté le canevas logiciel que nous avons conçu pour la construction d'outils de traitement d'ADL. Ce canevas définit principalement une architecture extensible à base de composants et des modèles de programmation adaptés aux divers types de traitements qui doivent être effectués par une chaîne d'outils de traitement d'ADL. Par ailleurs, il fournit des mécanismes de base qui peuvent être réutilisés pour la construction de divers types d'outils de traitement d'ADL. De plus, il est doté d'un mécanisme de *plugin* permettant aux tierces-parties d'intégrer des extensions de manière à adapter la chaîne de traitement à leurs besoins.

[4]En effet, chaque plugin doit obéir à une convention de nommage, et implanter une interface serveur appelée `plugin`.

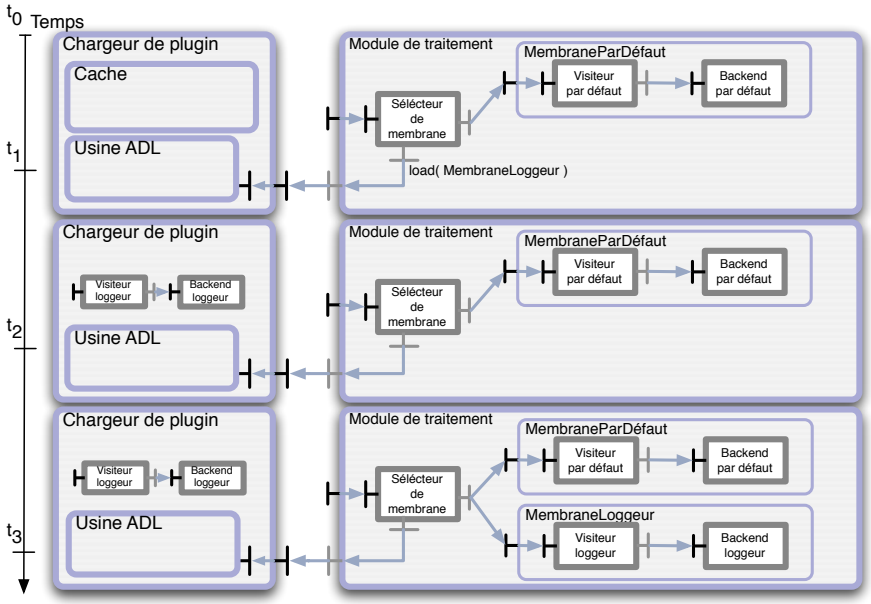


FIG. 6.15 – Flot d'exécution du chargement et de l'utilisation d'un plugin.

Les contributions principales de ce canevas par rapport aux outils existants sont sa capacité de fusion de descriptions d'architectures hétérogènes, sa capacité d'intégration de différents composants d'analyse effectuants des opérations de vérification et d'enrichissement d'architectures, et sa flexibilité en ce qui concerne l'intégration de divers modules de traitement. Ce dernier point est obtenu grâce à l'utilisation d'un canevas de tâches permettant une gestion hiérarchique des opérations à réaliser et grâce à une architecture à base de composants à grain fin permettant de mettre en œuvre des modules de traitements de façon modulaire. Nous pensons qu'une telle infrastructure présente un apport intéressant pour la construction d'outils de programmation spécifiques dans le cadre du processus de développement de logiciels à base de composants.

Chapitre 7

Construction d'un outil de traitement d'ADL pour la génération de code

Sommaire

Ce chapitre est consacré à la présentation d'un outil de traitement d'ADL dédié à la génération de code, conçu en utilisant le canevas logiciel présenté dans le chapitre précédent. Plus précisément, le compilateur ADL en question est une personnalisation des modules de chargement et de traitement génériques du canevas dont le but est de générer une partie du code des composants FRACTAL. Une particularité de cette personnalité est sont aspect multi-cibles, c'est à dire sa capacité à supporter la programmation des composants dans divers langages de programmation tels que le C, le C++ et le Java. Elle permet, par ailleurs, la compilation de l'ensemble du code généré et des codes sources écrits par les programmeurs.

7.1 Objectifs

Le rôle de la personnalité de génération de code est de fournir deux fonctions fondamentales :

- **L'assemblage d'un système à base de composants** permet aux programmeurs de construire et de configurer une application à partir des descriptions d'architectures qui lui sont liées, faisant référence à une ou plusieurs bibliothèques de composants. L'assemblage nécessite la génération de code *glue* qui permet d'assembler les composants. Dans la plupart des cas, l'opération d'assemblage comprend aussi la compilation de l'ensemble des codes, générés et implantés, afin de compiler le résultat d'assemblage sous une forme binaire.
- **La simplification de l'implantation des composants** vise à fournir des guides de programmation offrant aux programmeurs un ensemble de macro-opérations (e.g. invocation d'interface cliente, accès aux attributs, etc). De tels guides de programmation s'appuient sur la génération effective de ces macro-opérations par le compilateur ADL selon l'architecture de chaque composant. Il

s'appuie, de même, sur la génération de portions de codes dépendant de l'architecture interne des composants comme des structures de données et des aspects de contrôles. Le support développé pour le canevas THINK décrit un exemple d'aide à l'implantation de composants écrit en C. Les lecteurs intéressés trouveront une description détaillée de ce support dans l'annexe A.1.

À ces objectifs d'aide à la programmation et à l'assemblage d'architectures logicielles à base de composants, se rajoutent la nécessité de supporter les trois caractéristiques suivantes du modèle de composants FRACTAL :

1. **La possibilité d'intégrer différents langages d'entrée :** il est nécessaire de permettre d'intégrer différents langages de description d'architectures (e.g. FRACTALADL, THINKIDL), mais aussi différents langages de programmation comme C, C++ et Java. Cela implique en particulier le support de différents guides de programmation.
2. **La possibilité d'assurer différents types d'implantation de** *membranes* **:** il est nécessaire de fournir un mécanisme d'extensions permettant aux tierces-parties d'étendre l'outil de génération de code afin d'y intégrer de nouveaux tisseurs de *membranes*.
3. **La possibilité de produire différents types de résultats :** il est nécessaire de savoir générer différents types de résultats : exécutable binaire monolithique, librairie destinée au chargement dynamique, image binaire de noyaux de système d'exploitation, etc. Le fait que différents langages de programmation et différentes plates-formes d'exécution puissent être utilisées implique la nécessité d'être compatible avec différentes chaînes de compilation existantes (e.g. gcc, JDK, etc.). Par ailleurs, nous pouvons citer la nécessité de la génération des adaptateurs de communication en fonction des description d'architecture, qui constitue un autre type de résultat intéressant pour les systèmes répartis et pour les logiciels hétérogènes écrits dans différents langages de programmation (e.g. C, Java).

Afin de répondre aux besoins suscitées, l'outil de génération de code doit avoir une architecture modulaire et extensible. Notons que l'extensibilité est également requise du fait du caractère expérimentale des projets de développement réalisés par les utilisateurs de ce compilateur ADL. En effet, le compilateur doit pouvoir être étendu pour répondre aux changements fréquents des besoins de ces projets.

7.2 Architecture de l'outil de génération de code

L'implantation de la personnalité de génération de code nous a amené à réaliser un certain nombre de spécialisations de l'architecture présentée dans le chapitre précédent. Cette section est consacrée à la présentation de ces spécialisations. Nous commençons par décrire l'architecture du module de chargement. Ensuite, nous exposons une version spécialisée du canevas de tâches pour la génération de code. Enfin, nous présentons l'architecture du module de traitement qui permet d'organiser les tâches de génération de code.

7.2.1 Architecture du module de chargement

Un assemblage particulier de la chaine de construction d'AST à été mis au point afin d'obtenir un module de chargement adapté aux caractéristiques mentionnées dans la section précédente. Cette chaîne de construction d'AST est dirigée par des fichiers ADL, écrits en FRACTALADL. En plus de ces derniers, elle charge des fichiers IDL et des fichiers d'implantation liés à la description d'architecture lue. Le chargement de ces fichiers annexes à la description d'architecture a pour objectif la construction d'un AST suffisamment riche pour être utilisé à des fins de génération de code. De plus, cette chaîne procède au chargement des tisseurs de code en charge de la génération des *membranes* des composants (i.e. implantation des interfaces de contrôle). Enfin, la chaîne effectue un certain nombre d'analyses sémantiques sur l'architecture à générer.

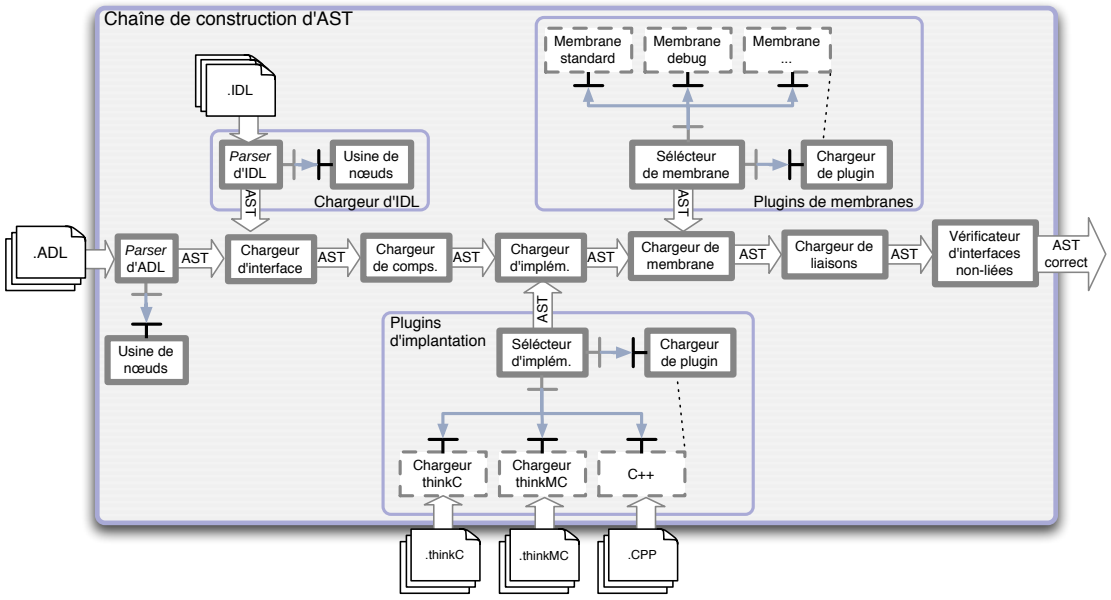


FIG. 7.1 – Architecture de la chaîne de chargement de l'outil de génération de code.

La figure 7.1 illustre à la fois l'architecture et le flot d'exécution de la chaîne de construction de l'AST. Rappelons que cette chaîne de composants fonctionne en mode « aspiration », dans le sens où l'on aspire un AST depuis le bas de la chaîne vers le composant le plus haut (i.e. depuis le *parser d'ADL* vers le *vérificateur d'interfaces non liées*). Les flèches fines représentent le sens d'invocation (i.e. client vers serveur) alors que les flèches épaisses représentent le flot de données allant dans le sens inverse du sens d'invocation (i.e. de serveur vers client).

Les principaux composants faisants partie du module de construction de l'AST sont les suivants :

- **Le chargeur d'ADL** a pour rôle de lire un fichier ADL et de retourner l'AST lui correspondant. Ce composant est lui-même constitué de deux sous-composants. Le *parser* d'ADL lit le fichier et dirige la construction de l'AST en utilisant l'usine de nœuds. L'usine de nœuds est spécialisée en utilisant la spécification de grammaire de FRACTALADL. Étant donné que la syntaxe de FRACTALADL est basée sur XML, l'implantation du *parser d'ADL* est réalisée en spécialisant un *parser* SAX générique à l'aide de la même spécification de grammaire.
- **Le chargeur d'interfaces** a pour fonction d'enrichir l'AST reçu en entrée. Plus précisément, le chargeur d'interfaces lit les fichier de description des interfaces (IDL) afin de créer des AST représentant les interfaces et de fusionner ces AST avec l'AST reçu en entrée. Cette fusion est effectuée en utilisant les mécanismes expliqués dans la section 6.2.3. Le chargeur d'interfaces utilise un chargeur d'IDL dont l'architecture est identique à celle du chargeur d'ADL. Notons simplement que l'implantation de l'usine de nœuds utilisée pour la construction des AST représentant les interfaces est spécialisée en utilisant la grammaire propre à ce langage.
- **Le chargeur de composants** est en charge de parcourir en profondeur l'AST reçu en entrée et de le compléter avec les descriptions manquantes réalisées dans différentes fichiers ADL. En effet, FRACTALADL autorise la description d'architectures à l'aide de plusieurs fichiers. Le chargeur de composants produit un AST représentant l'architecture complète du logiciel à construire.
- **Le chargeur d'implantations** est responsable du chargement des fichiers d'implantation. L'interprétation des fichiers d'implantation dépend du langage d'entrée. Par exemple, dans le cas de Java, certaines vérifications sémantiques sont effectuées lors du chargement des composants primitifs,

comme la compatibilité des interfaces fournies par la classe d'implantation avec les interfaces serveurs du composant. Comparativement, aucune vérification de ce type n'est effectuée dans le cas du C ou du C++. Afin de permettre cette différence de traitement pour chaque langage d'implantation, le chargement est délégué aux chargeurs spécifiques à chaque langage par le chargeur principal. Les chargeurs spécifiques sont implantés sous forme de plugins et sont chargés dynamiquement lorsqu'ils sont requis. Le choix du chargement de plugin et le routage du traitement vers le bon délégué sont assurés par le sélecteur d'implantation.

- **Le chargeur de membranes** assume l'enrichissement de l'AST avec les informations nécessaires pour le tissage de l'implantation des contrôleurs de chaque composant. La chaîne d'outils peut générer plusieurs types de membranes. Par ailleurs, le choix de tisseur de membrane peut dépendre du langage d'implantation des composants. Afin de permettre, là encore, cette diversité de traitement, les tisseurs de membranes sont écrits comme dans le cas du chargeur d'implantation sous forme de plugins chargés dynamiquement. Le chargement et le choix du tisseur à utiliser sont assurés par le **sélécteur de membranes**. Ce dernier interprète la description de chaque composant pour en déduire quel composant doit être utilisé pour tisser l'implantation des interfaces de contrôle de chaque composant.
- **Le chargeur de liaisons** parcourt l'AST en profondeur et effectue certaines optimisations concernant les liaisons. Par exemple, les liaisons traversant plusieurs composites sont simplifiées en des liaisons directes afin d'améliorer les performances d'exécution.
- **Le vérificateur d'interfaces non-liées** parcourt l'AST pour s'assurer que toutes les interfaces client sont liées à une interface serveur. Dans le cas contraire, l'assemblage manque de cohérence. Une exception est alors levée afin de stopper le chargement et d'afficher une erreur de compilation.

Remarquons que la chaîne de construction de l'AST présentée ci-dessus est doublement extensible. En premier lieu, cette chaîne peut être facilement étendue, par le biais de l'écriture de nouveaux *plugins*, afin d'intégrer de nouveaux langages de programmation et de nouveaux tisseurs de membranes. Ce type d'extension ne nécessite pas la modification de la chaîne ; leur prise en compte est assurée par les sélecteurs de *plugins*. En second lieu, il est possible d'étendre la chaîne en y intégrant de nouveaux composants d'analyse. Ce type d'extension nécessite la modification de l'architecture de la chaîne en étendant le fichier ADL correspondant. Lorsqu'un concepteur entreprend ce type d'extensions, il doit être attentif à l'emplacement où le nouveau composant est inséré. En effet, la logique de précédence des composants de chargement qui assure l'ordre d'exécution des fonctions d'analyse doit être systématiquement respectée[1].

7.2.2 Spécialisation du canevas de tâches

Afin de simplifier la programmation du module de traitement de l'outil de génération de code, une personnalité spéciale du canevas de tâches a été définie. Cette personnalité fournit des types de tâches appropriés à la génération et à la compilation de code source, permettant ainsi de mieux modéliser les flots de données échangés entre les opérations concernées.

Nous consacrons cette sous-section à la présentation de cette version du canevas de tâches. Pour ce faire, nous débutons par l'introduction des caractéristiques des types de tâches définis. Nous analysons ensuite un extrait de graphe de tâches au travers d'un exemple de graphe de tâches pour la génération de code.

[1]Rappelons qu'un composant de chargement de niveau n utilise l'AST résultant du niveau $n-1$. L'emplacement correct d'un composant C dans la chaîne dépend alors des traitements qui sont supposés être effectués avant l'intervention de celui-ci.

7.2.2.1 Types de tâches pour la génération de code

Le tableau 7.1 présente les types de tâches définis pour assurer la génération et la compilation de code. Sont par ailleurs illustrés dans ce tableau les flots de données échangés entre les tâches en fonction de leurs types respectifs.

Tâche		Nom de type	Nom d'instance	Code Source	Fichier
`TypeProvider`	`TP`	↑	-	-	-
`InstanceProvider`	`IP`	↓	↑	-	-
`SourceCodeProvider`	`SCP`	↓	↓	↑	-
`SourceCodeConsumerProvider`	`SCCP`	↓	↓	↓↑	-
`SourceFileProvider`	`SFP`	↓	↓	↓	↑
`FileProvider`	`FP`	↓	↓	-	↑
`FileConsumerProvider`	`FCP`	↓	↓	-	↓↑

TAB. 7.1 – Présentation des types de tâches ainsi que de leurs flot de données. Les flèches vers le bas représentent les flots entrants alors que les flèches vers le haut représentent les flots sortants.

Nous analysons par la suite les caractéristiques associées à chacun des types de tâches.

- Les tâches de type `TypeProvider` produisent une référence unique pour chaque type de composants. Leur entrée n'est basée que sur les informations venant de l'AST. Par conséquent, elles n'ont aucunes dépendances envers d'autres tâches.
- Les tâches de type `InstanceProvider` produisent une référence unique pour chaque instance de composant. Etant donné que chaque instance possède un type spécifique, ces tâches dépendent non seulement des informations venant de l'AST, mais aussi d'une tâche de type `TypeProvider` qui est en charge de lui associer un type de composant.
- Les tâches de type `SourceCodeProvider` produisent du code source. Ces tâches peuvent être spécifiques à un type ou à une instance de composant. Pour cette raison, elles dépendent formellement des tâches qui fournissent ces références. De plus, elles peuvent utiliser les informations venant de l'AST pour obtenir des informations plus sophistiquées en fonction du code qu'elles vont produire. Remarquons que le code produit par ce type de tâches peut prendre des formes variées comme un segment de texte (i.e. un *buffer* de *string*), un ensemble de segments textes (i.e. des ensemble *buffers* de *string*) ou bien un AST correspondant au format intermédiaire du code à générer.
- Les tâches de type `SourceCodeConsumerProvider` sont similaires aux tâches de type `SourceCodeProvider` excepté qu'elles font usage des portions de code source produites par d'autres tâches. Ce type de tâches correspond aux générateurs de code qui utilisent les résultats d'autres générateurs de code pour modifier ou fusionner leurs résultats.
- Les tâches de type `SourceFileProvider` sont en charge de fournir des fichiers de code source. Pour ce faire, elles utilisent les codes sources produits par des tâches appartenant aux deux types précédents.
- Le type de tâches `FileProvider` peut être considéré comme une version plus générique du précédent. Les tâches de ce type sont en charge de fournir des fichiers pour un type ou une instance de composant donné. Par exemple, une tâche fournissant directement le fichier d'implantation d'un composant primitif à destination d'autres tâches comme des tâches de compilation.
- Les tâches de type `FileConsumerProvider` utilisent un certain nombre de fichiers produits par des tâches appartenants aux deux types précédents pour en produire d'autres. Ce type de tâches

permet par exemple d'effectuer les opérations de compilation qui prennent un ensemble de fichiers en entrée et qui produisent le ou les fichiers binaires résultant de la compilation proprement dite.

7.2.2.2 Analyse d'un graphe de tâches

Afin d'illustrer l'utilisation de cette personnalité du canevas de tâches, nous proposons par la suite l'analyse d'un extrait de graphe de tâches de génération de code. Pour ce faire, nous reprenons l'exemple de génération de code effectué pour l'architecture de composant décrite sur la figure 7.2. Dans cette exemple, le composant réalise des opérations arithmétiques. Le composant possède deux interfaces fonctionnelles : une interface serveur appelée `arithmetic` qui fournit des opérations arithmétiques et une interface client appelée `console` qui fournit des opérations d'affichage. Le composant `ArithmeticComp` est implanté en C dans un fichier appelé `ArithmeticImpl.c`. La *membrane* du composant est la version *standard* proposée par le compilateur ADL. Cette *membrane* fournit une interface *component identity* et une interface *binding controller* (resp. *attribute controller*) lorsque le composant a des interfaces client (resp. des attributs). Enfin, ce composant est instancié au sein d'une architecture logicielle sous le nom *comp1*.

```
<!-- Définition du composant d'opération arithmétique -->
<definition name=''ArithmeticComp''>
  <interface name=''arithmetic'' role=''server'' signature=''Arithmetic'' />
  <interface name=''console''    role=''client'' signature=''Console'' />
  <implementation class=''ArithmethicImpl'' language=''C'' />
  <controller desc=''standard''/>
</definition>

<!-- Instanciation du composant ArithmeticComp au sein d'une architecture -->
...
<component name=''comp1'' definition=''ArithmeticComp''/>
...
```

FIG. 7.2 – Description de l'architecture d'un composant d'opération arithmétique en FRACTALADL.

La figure 7.3 illustre un extrait du graphe de tâches obtenu lors de l'exécution du module de traitement[2] pour l'architecture décrite ci-dessus. Les nœuds du graphe représentent les instances de tâches. Pour chaque instance est noté son nom, son type et l'élément pour laquelle elle est utilisée. Les flèches illustrent les dépendances entre les tâches. Les dépendances concretes (i.e. dépendances de code source, de fichier, etc.) sont illustrées avec des lignes continues, alors que les dépendances conceptuelles (i.e. type, instance) sont illustrées avec des lignes pleines.

Pour analyser le graphe de tâches, nous effectuons une lecture de gauche à droite, ce qui donne une analyse commençant par le résultat et progressant vers les sources de ce résultat. À gauche du graphe est représentée la tâche de compilation d'un fichier source (N°23). Cette tâche de type *file consumer provider* dépend de l'ensemble des tâches fournissant les fichiers sources générés (N°22) pour le composant ainsi que des fichiers entêtes (N°18 à N°21). La génération des fichiers entêtes est nécessaire pour la définition des types d'interfaces, d'attributs, etc. Ces tâches ont des dépendances qui ne sont pas représentées dans cet extrait simplifié. La tâche N°22 est de type *source file provider* et a pour rôle de regrouper l'ensemble des codes sources produits par les tâches de définition (N°17) et d'instanciation de composants (N°18). Selon la spécification de génération de code considérée, un fichier source doit être produit pour chaque type de composant. Ainsi, la tâche N°22 dépend, entre autres, du type de composants défini par la tâche N°1, afin d'accéder au nom du fichier dans lequel devrait être généré l'ensemble des codes sources produits pour le composant traité.

[2]Le flot d'exécution du module de traitement pour la génération de code est présenté dans la sous-section suivante.

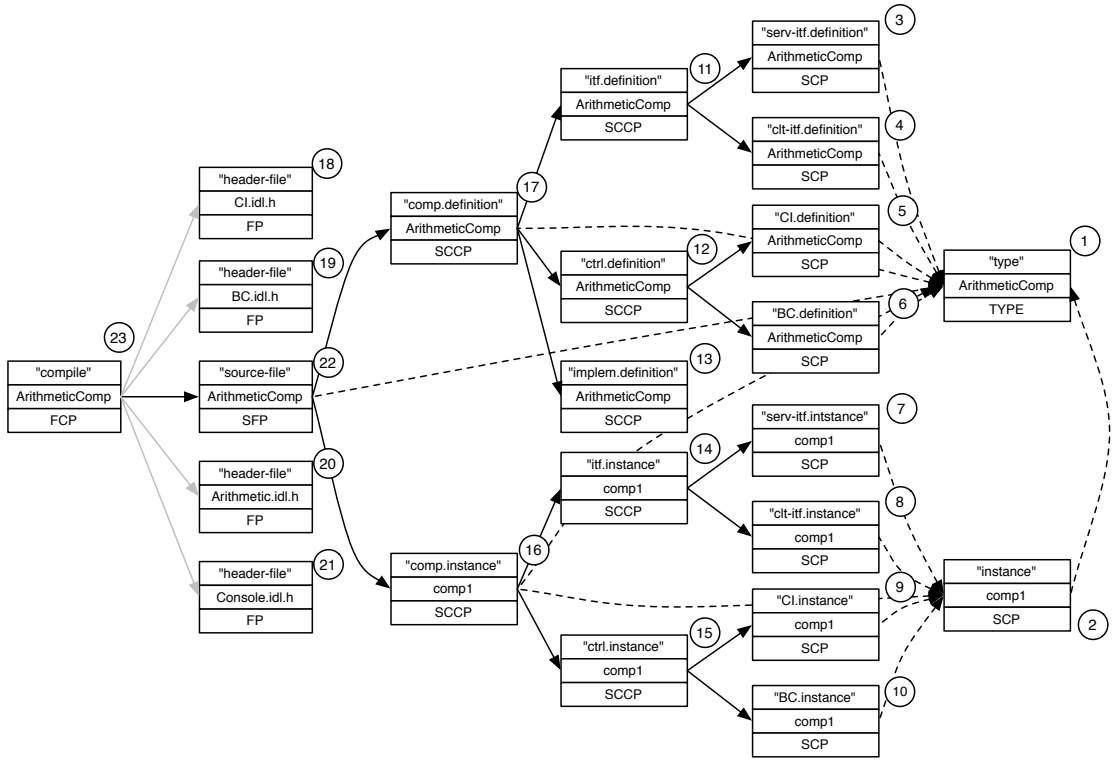


FIG. 7.3 – Extrait du graphe de tâches construit par le module de traitement pour la génération de code.

La tâche N°17 est en charge de fournir l'ensemble des codes sources produits pour la définition d'un type de composants. Elle dépend donc elle aussi de la tâche produisant le type du composant (N°1) et des tâches produisant les codes sources de définition (la moitié haute du graphe). Le code source collecté pour la définition d'un type de composants est généré par trois tâches de haut niveau assurant la définition de la partie fonctionnelle, de la partie contrôle et de l'implantation du composant. La partie fonctionnelle (N°11) concerne la définition de la structure de données liée aux interfaces fonctionnelles du composant. Elle utilise le code source produit par deux tâches produisant respectivement le code des interfaces serveur et celui des interfaces client. La partie contrôle (N°12) utilise l'ensemble du code généré par les tisseurs de *membranes*. Etant donné que dans notre exemple il s'agit d'une membrane standard, nous retrouvons la définition des interfaces *component identity* (N°5) et *binding controller* (N°6). Parce que le composant `ArithmeticComp` ne possède pas d'attributs, il n'y a pas de tâche produite dans le graphe pour la génération d'une interface de type *attribute controller*. Enfin, la tâche N°13 fournit le code source de l'implantation.

La génération du code source du composant `ArithmeticComp` est modélisée par la tâche N°18. Dans la mesure où cette tâche est spécifique à une instance donnée, celle-ci dépend aussi bien de la tâche définissant le type que de celle définissant l'instance. Pour réaliser sa fonction, cette tâche réutilise le code produit pour l'instanciation des parties fonctionnelle et de contrôle. De façon similaire à la tâche N°11, la tâche N°14 génère le code d'instanciation des structures fonctionnelles. De même, l'instanciation des structures de la *membrane* est produite par la tâche N°15 et par les tâches dont elle dépend (N°9 et N°10).

Notons que cet extrait de graphe constitue une sous-partie du graphe modélisant la génération de code pour l'architecture au sein de laquelle est instancié le composant *comp1*. Comme cette analyse le

montre, les opérations de génération de code sont modélisées à un niveau très fin. Bien que compliqué à analyser sur un tel document, cette structuration à grain fin permet de rendre la génération de code très modulaire, simplifiant ainsi les extensions ultérieures. À titre d'exemple, s'il était nécessaire de modifier le code généré pour la *membrane* des composants, seules les tâches 5, 6, 9, 10, 12 et 15 devraient être modifiées.

7.2.3 Architecture du module de traitement

Après le module de chargement et le canevas de tâches, le module de traitement constitue le troisième élément qui a du être personnalisé afin de mettre au point l'outil de traitement d'ADL dédié à la génération de code. Rappelons que le rôle de ce module est d'interpréter les informations architecturales contenues dans l'AST afin d'organiser les opérations à exécuter pour réaliser la génération et la compilation de code. Il est principalement composé de deux types de composants qui construisent le graphe de tâches et qui fournissent l'implantation concrète des tâches.

Compte tenu du caractère multi-cibles de la génération de code envisagée, nous avons opté pour une approche privilégiant la minimisation des efforts nécessaires pour le support de langages de programmation cibles différents. Dans ce cadre, nous avons tout d'abord conçu une architecture de *plugin* réutilisable qui permet de séparer les modules de traitement propres à chacun des langages considérés. Ensuite, afin de réutiliser un maximum de composants de construction de graphe de tâches, nous avons défini des règles de construction ayant la capacité de modéliser la génération de code pour l'ensemble de ces langages. Ainsi, nous avons réussi à réduire l'effort de portage uniquement à l'implantation des composants *backends* propres à chacun des langages (C, C++ et Java dans le cadre de cette thèse). Les lecteurs intéressés trouveront les caractéristiques des codes générés pour ces langages dans l'annexe A. Dans le reste de cette section, nous allons nous concentrer sur l'architecture de traitement mise au point.

L'architecture du module de traitement est structurée en plusieurs niveaux hiérarchiques. Pour des raisons de place, nous allons illustrer chacun de ces niveaux en utilisant des figures séparées. La figure 7.4 présente le composant de traitement de plus haut niveau (i.e. le composant réifiant le module de traitement). Ce composant admet en entrée un AST correct (représenté par la flèche épaisse) produit par le module de chargement. Cet AST est donné à un composant *voyageur* qui est en charge de le parcourir en profondeur et d'invoquer le composant *routeur* pour chaque nœud de type composant trouvé. Le *routeur* est un composant de type *expéditeur* qui a pour rôle de réexpédier les nœuds composant reçus en entrée vers l'un de ses deux clients en fonction de la nature primitive ou composite du composant traité. Cette sélection s'impose compte tenu de la différence des tâches de génération de code à créer pour ces deux natures de composants.

La suite du traitement dépend directement du langage de programmation utilisé pour implanter le composant traité. Pour cette raison, les composants de traitement pour primitifs et composites contiennent rien de plus que des sélecteurs de *plugins* attentifs au langage de programmation spécifié dans la description d'architecture. À partir de ce niveau de *plugins*, les concepteurs peuvent intégrer des composants de traitement spécifiques pour différents langages cibles.

La figure 7.5 illustre l'architecture d'un *plugin* de génération de code spécifique à un langage de programmation donné. Il s'agit précisément des *plugins* chargés par les sélecteurs présentés dans la figure 7.4. Ces plugins contiennent trois sous-composants de manière à distinguer la génération de code pour :

1. la définition des structures de données des composants,
2. l'instanciation de ces structures de données pour chaque instance de composant,
3. la compilation des codes générés.

Pour mieux comprendre à quoi correspondent les opérations effectuées par ces étages, considérons

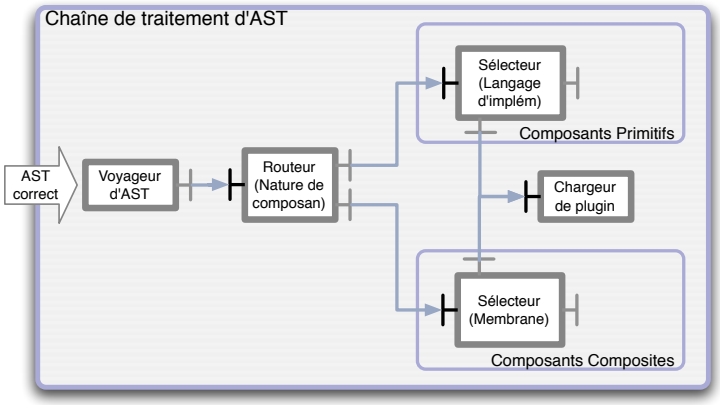


FIG. 7.4 – Architecture initiale de la chaîne de traitement.

un exemple de génération de code pour des composants primitifs écrits en C++ (voir la section A.2 pour une plus ample description de l'intégration du langage C++ dans la chaîne). Le premier étage de la liste ci-dessus correspond à la génération de trois types de classes qui servent (i) pour la définition des interfaces, (ii) pour l'implantation des opérations de contrôle fournies par la *membrane* du composant, et finalement (iii) pour la définition du type du composant, ce dernier étant défini par une classe héritant des deux premières. Le deuxième étage correspond à la création d'une instance (i.e. opération *new*) de la classe définissant le type du composant. Enfin, le dernier étage correspond à la compilation de l'ensemble des classes mentionnées précédemment, ainsi que de la classe d'implantation fournie par le programmeur.

Un composant *expéditeur*, placé en entrée du composant illustré sur la figure 7.5, est en charge de faire suivre les invocations reçues vers les trois sous-composants réifiant les étages mentionnés. Le composant réifiant le premier étage contient en entrée un *expéditeur* sélectif. Le rôle de ce dernier est de capturer les nouveaux types de composants trouvés dans l'architecture traitée. En effet, cette détection est requise pour le cas spécifique du langage FRACTALADL qui ne demande pas la définition explicite de types de composants dans les description d'architecture. Ce détecteur invoque la suite des opérations de définition de types si le nœud traité en définit un nouveau et retourne sans traitement dans le cas contraire. Derrière ce détecteur se trouve un expéditeur générique qui diffuse les invocations au reste des composants. Ces composants sont au nombre de quatre. Ils permettent la génération des structures de données pour définir un nouveau type de composant. Nous les décrivons dans la suite.

Le premier composant traite la génération des structures associées aux interfaces et aux attributs des composants. Les composants blancs sont les visiteurs qui organisent les tâches, alors que les composant gris sont les composants *backends* fournissant l'implantation des tâches. Le deuxième composant s'occupe de la définition des structures de données concernées par la partie fonctionnelle des composants. Le code généré dépend du format des structures de données qui sont définies pour l'implantation de la membrane du composant. Pour cette raison, le composant de *définition des structures fonctionnelles* est constitué d'un sélecteur de *plugins* attentif au type de *membrane* spécifié dans la description d'architecture du composant. L'architecture typique d'un *plugin* qui peut être chargé à ce niveau est présentée sur la figure 7.6.a. Un tel *plugin* est composé des *visiteurs* assurant la génération du type fonctionnel du composant traité, ce qui correspond à la définition de ses interfaces et de ses attributs. Selon l'architecture que nous avons conçue pour ce type de *plugins*, la création des tâches dédiées à la génération du code pour la définition des interfaces est réalisée par trois *visiteurs* distincts. Les deux premiers sont en charge de générer la définition des interfaces client et serveur, et le troisième a pour rôle de définir une structure

de donnée agrégeant le résultat de ces derniers. Un autre *visiteur* est dédié à la génération des structures de données nécessaires à la définition des attributs. Enfin, un dernier *visiteur* définit une tâche qui a pour rôle de collecter le code généré par les tâches précédentes, afin de générer la définition complète de la partie fonctionnelle du composant traité.

Comme son nom l'indique, le composant de *définition des structures de membrane* présenté dans la figure 7.5 a pour rôle de générer la définition de la *membrane* des composants. Similairement à la partie fonctionnelle, la définition des structures de données générées par ce composant dépend du type de *membrane* à implanter. Des *plugins* sont utilisés afin de supporter différentes implantations de ce composant de traitement. La figure 7.6.b illustre l'architecture d'un tel *plugin*. Sont présents dans ce type de *plugins* des couples *visiteurs*/*backends* dédiés à la génération de l'implantation de diverses interfaces de *contrôle*. Le nombre de ces couples dépend du nombre et de la structure des interfaces de contrôle implantées par la *membrane* gérée par le *plugin*.

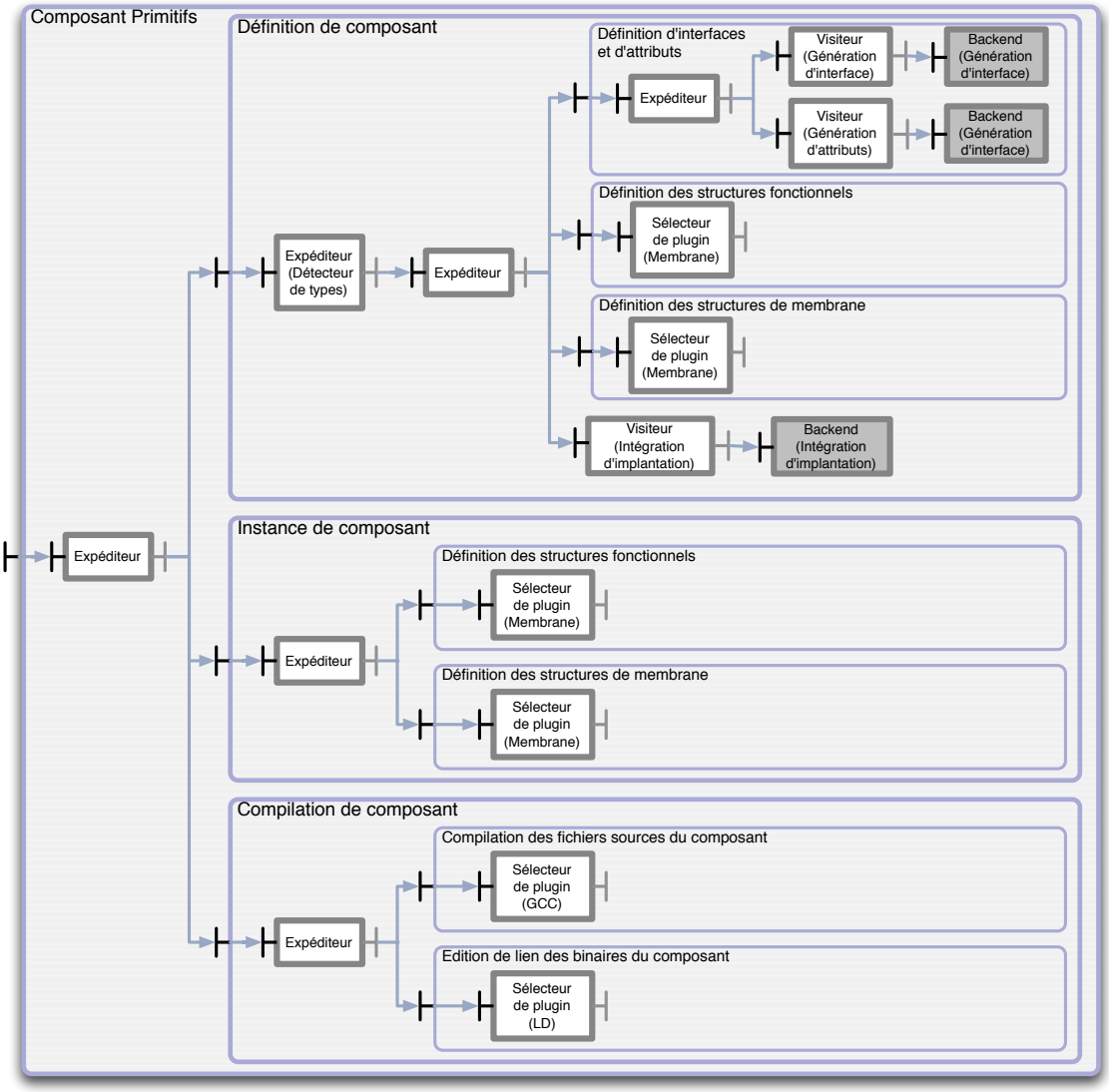


FIG. 7.5 – Architecture d'un module de traitement pour composants ptimitifs.

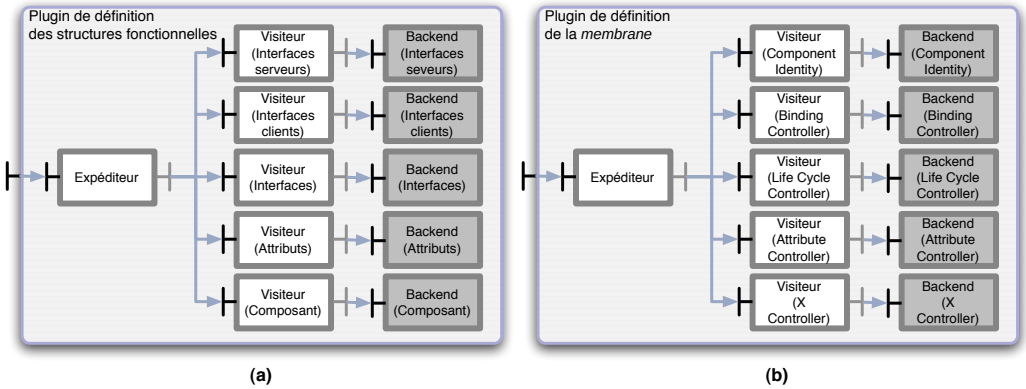


FIG. 7.6 – Architectures des plugins de définition de la partie fonctionnelle et de la *membrane* des composants.

Pour compléter la définition du type des composants, un dernier couple de *visiteur* et *backend* organise l'intégration des morceaux de code générés pour la définition du type de composants avec le (ou les) fichiers d'implantation fournis par les programmeurs. L'implantation du *visiteur d'intégration* est générique, au moins pour les trois langages cibles supportés actuellement.

À la suite du composant de *définition de composant* illustré dans la figure 7.5, se trouve un composant s'occupant de l'instanciation des composants. Le rôle de ce composant est d'effectuer la génération du code permettant d'instancier les structures de données définies pour le composant traité. Les sous-composants implantant ce composant sont au nombre de deux : le premier organise les tâches d'instanciation de la partie fonctionnelle alors que le deuxième organise les tâches d'instanciation de la *membrane*. Comme les structures de données associées à ces deux parties des composants dépendent de l'implantation de la *membrane* choisie, ces composants de traitement sont constitués de sélecteurs de *plugins* attentifs au type de *membrane* spécifié pour le composant traité. Les *plugins* s'intégrant à ce niveau ont des architectures similaires à celles présentées dans la figure 7.6. Cependant, l'implantation des couples *visiteur/backend* sont spécifiques à la génération de code d'instanciation.

L'exécution des tâches de définition et d'instanciation suscitées résulte en la génération d'un certain nombre de fichiers sources[3]. Le rôle du troisième composant de traitement illustré dans la figure 7.5 est de compiler ces fichiers et potentiellement de lier les modules binaires générés. Des plugins sont encore utilisés à ce niveau afin de pouvoir sélectionner parmi plusieurs choix de compilateurs et d'éditeurs de lien, ces derniers étant potentiellement différents pour chaque composant de l'application. Deux critères président à la sélection d'un plugin : le format d'objet binaire attendu à l'issu de la compilation (i.e. binaire statique, exécutable, binaire dynamiquement relogeable, etc.) — qui influence l'organisation des tâches de compilation —, et la plateforme matérielle cible — qui influence la version du compilateur et de l'éditeur de liens à utiliser.

En résumé, les patrons de programmation et les guides architecturaux définis dans le chapitre précédent ont permi la définition d'une architecture générique et hiérarchique pour l'implantation d'un module de traitement dédié à la génération de code à partir de descriptions d'architecture. Par ailleurs, le module de traitement ainsi défini remplit différents besoins de flexibilité concernant :

1. le support pour divers langages de programmations cibles,

[3]Bien que nous n'ayons pas détaillé l'organisation de toutes les tâches de génération de code en raison de leur nombre, notons simplement qu'elles génèrent le code source de manière hiérarchique de façon similaire au cas que nous avons traité dans l'exemple 6.10 dans le chapitre précédent.

2. le support pour divers implantations de *membranes*,
3. le support pour divers outils de compilation.

En effet, l'ensemble de ces extensions peuvent être implantées par des concepteurs de tierces-parties sous forme de *plugins*. Il nous semble que l'architecture des composants de traitement définis pourrait satisfaire la génération de code pour d'autres langages de programmation. Seuls les composants *gris* correspondant aux *backends* devraient être réécrits. Cependant, si cette architecture ne convient pas pour un langage donné, les concepteurs ont la liberté de définir une autre architecture de *plugin*, de manière à remplacer celle que nous avons présentée dans la figure 7.5. Par ailleurs, les concepteurs peuvent aller jusqu'à étendre le module de chargement ainsi que le canevas de tâches si ces derniers présentent des propriétés non-adaptées aux extensions envisagées.

7.3 Génération d'adaptateurs de communication

L'outil de génération de code présenté précédemment intègre une fonction dédiée à la génération automatique des adaptateurs de communication. Cette fonction est très utile pour automatiser la connexion des composants dont la communication nécessite la mise en place de liaisons complexes au sens défini par le modèle FRACTAL[4]. Parmi des exemples de telles liaison, citons les liaisons inter-processus (i.e. IPC), les liaisons inter-processeurs (i.e. RPC), les liaisons systèmes (i.e. *syscall*), ou encore les liaisons de type JNI entre des composants écrits en Java et en C/C++.

Notre démarche pour l'adaptation des liaisons peut se comparer à la proposition concernant les connecteurs logiciels des concepteurs d'ArchJava[ASCN03]. Ce dernier propose un modèle de programmation générique pour permettre aux programmeurs de construire des connecteurs implantant des protocoles variés. La différence majeure entre cette approche et la notre est que la première est basée sur la programmation réflexive alors que la deuxième est basée sur la génération statique de code. Si la programmation réflexive présente des avantages indéniables, elle se base sur un support à l'exécution fourni par l'environnement Java. Or compte tenu des contraintes que nous considérons pour la programmation des plates-formes systèmes sur puce, notre solution doit fonctionner avec des langages de bas niveau (e.g. C). De plus, la génération statique des adaptateurs de communication permet d'obtenir de meilleures performances, compte tenu du coût à l'exécution du support pour la programmation réflexive.

Un autre aspect qui distingue notre proposition de celle des concepteurs d'ArchJava est que nous visons l'automatisation, dans la limite du possible, de l'insertion des adaptateurs de communication par le biais d'une analyse architecturale. En effet, nous proposons d'intégrer, dans le module de chargement d'ADL, des analyseurs sémantiques capable d'insérer automatiquement des composants de liaison dès lors que certains *patterns* architecturaux sont détectés. De tels mécanismes intelligents peuvent être construits dans la plupart des cas suscités, pourvu que les informations architecturales nécessaires soient intégrées dans L'ADL. Par exemple, si l'ADL permet de spécifier sur quels processeurs seront exécutés les composants, un analyseur sémantique peut automatiquement insérer des composants de liaison de type RPC.

La suite de cette section est consacrée à la présentation de notre proposition pour la génération d'adaptateurs de communication. Nous commençons par décrire le module de chargement dédié à cette fonction. Ensuite, nous décrivons la partie qui effectue la génération effective du code d'implantation des composants de liaison. Afin d'illustrer le fonctionnement de ces deux modules sur des bases concrètes, nous considérons un exemple de support dédié à l'adaptation des interactions être les composants écrits en Java et en C.

[4]Une liaison complexe au sens FRACTAL correspond à une chaîne de composants implantant un protocole de communication arbitrairement complexe.

7.3.1 Insertion automatique d'adaptateurs de communication

L'insertion automatique d'adaptateurs de communication est effectuée au sein du module de chargement. A cette fin, un composant d'analyse sémantique est inséré afin de détecter les liaisons dont la modification est nécessaire. Ce composant a également la charge de modifier l'AST représentant la description d'architecture traitée. La figure 7.7.a illustre le cas d'une applications faisant intervenir des composants écrits en Java et en C. Afin qu'ils puissent communiquer, il est nécessaire d'établir entre ces composants une liaison de type JNI[5] (*Java Native Interface* [JNI]). Ce protocole requiert l'insertion de deux composants de communication, l'un du coté client (appelé *stub*) et l'autre du coté serveur (appelé *skeleton*). Le rôle du *stub* est de fournir, dans le monde Java, une interface équivalente à celle qui est fournie par le composant serveur, de manière à faire suivre les invocations du composant client en utilisant le format d'appel défini par le protocole JNI. De manière symétrique, le rôle du *skeleton* est de recevoir des invocations en provenance du protocole JNI pour les faire suivre vers le composant serveur écrit en C.

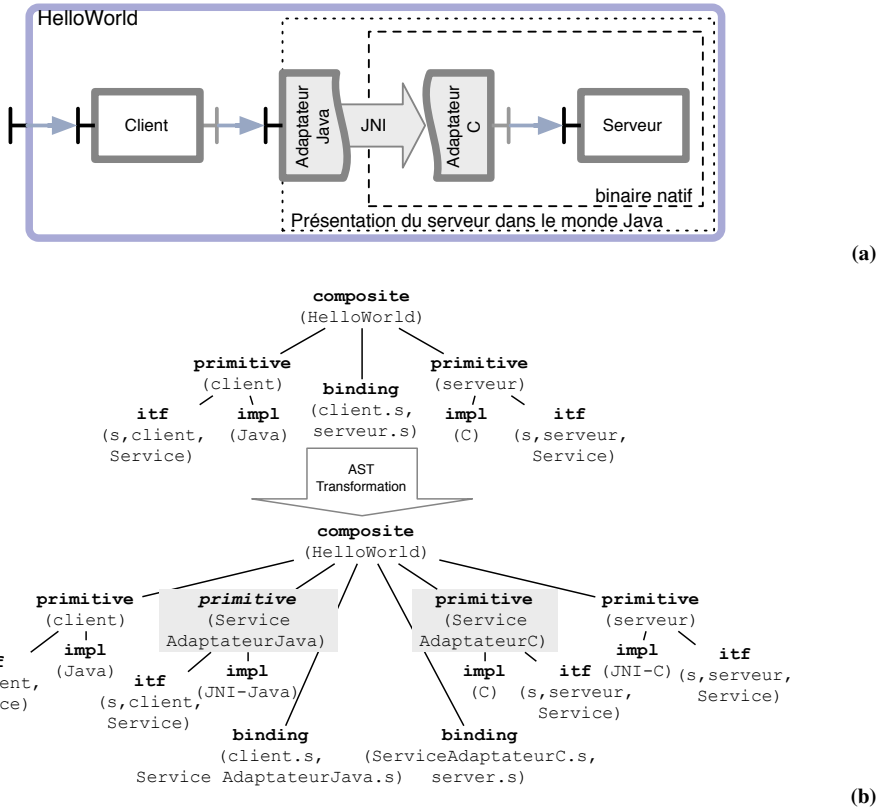


FIG. 7.7 – Architecture de liaison des composants inter-plates-formes via des adaptateurs de communication et la transformation d'AST associée.

L'analyseur sémantique mis en place pour adapter la communication entre les composants Java et C parcourt l'AST et détecte les liaisons effectuées entre des composants écrits dans ces deux langages. Une fois ces liaisons détectées, elles doivent être modifiées pour implanter le schéma de communication suscité. Cette modification est effectuée directement au niveau de l'AST. La figure 7.7.b illustre la modification correspondante dans le cadre de notre exemple. Nous y voyons que de nouveaux nœuds de type

[5]Le protocole JNI permet des interactions bidirectionnelles entre les composants logiciels écrits en C et en Java. Par souci de simplicité, nous traitons, dans cet exemple, un exemple restreint basé sur un schéma de communication possible.

composant sont crées pour réifier le *stub* et le *skeleton*. Par ailleurs, la liaison entre les composants client et serveur est modifiée, de manière à utiliser ces composants d'adaptation. Notons que la liaison entre les composants *stub* et *skeleton* n'est pas explicitement représentée dans l'architecture, étant donnée que celle-ci est implanté par le protocole JNI. Par ailleurs, des valeurs spéciales (e.g. JNI-C, JNI-Java) sont affectées au niveau du langage d'implantation des composants adaptateur, de manière à ce que le module de traitement soit capable de les distinguer ultérieurement.

7.3.2 Génération du code d'implantation des adaptateurs de communication

Après avoir modifié l'architecture traitée de manière à insérer des adaptateurs de communication, vient l'étape de la génération du code d'implantation de ces derniers. Les composants adaptateurs sont des composants normaux, hormis le fait que leur contenu doit être généré par le module de traitement. Le code à générer dépend de deux facteurs : le type d'interface et le protocole de communication à implanter. Ces facteurs sont figés par l'analyseur sémantique effectuant la modification d'architecture. Le type d'interface est défini par le nœud représentant l'interface des composants adaptateurs, alors que le protocole implanté est encodé dans le langage d'implantation de ces derniers.

Le support nécessaire pour la génération de ces composants est intégré dans le module de traitement sous forme d'un *plugin* de génération de composant primitif implantant l'architecture présentée dans la figure 7.5. En effet, l'encodage du nom du protocole de communication au niveau du langage d'implantation des composants permet au sélecteur de *plugin* présenté dans la figure 7.4 de faire suivre le traitement de ces composants vers le *plugin* concerné. Seul un composant doit être modifié dans l'architecture du composant illustrée dans la figure 7.5 pour supporter la génération d'implantation pour un protocole donné[6]. Il s'agit du *backend d'intégration d'implantation*. En effet, ce dernier doit, non pas intégrer un fichier source fourni par le programmeur, mais générer son propre fichier d'implantation en se basant sur la définition d'interface reçue en entrée.

Afin de simplifier la programmation de tels générateurs d'implantation, nous avons effectué un découpage du composant *backend* qui doit être fourni par les concepteurs, selon le patron de programmation *visiteur*. Un premier composant générique implante la tâche de génération de code réifié par le *backend d'intégration d'implantation*. Ce composant parcours la définition d'interface trouvée dans la partie de l'AST pour laquelle il est invoqué et appelle un visiteur implantant l'interface `InterfaceVisitor`, présenté dans la figure 7.8. Ce *visiteur* réifie le générateur de code effectif qui doit être implanté par les concepteurs de manière à générer le code correspondant au protocole de communication supporté. Ainsi, l'effort nécessaire pour le support de nouveaux protocoles de communication est réduit à la programmation d'un seul composant *backend* au niveau du module de traitement.

```
public interface InterfaceVisitor {
  /** Génère le code d'entrée pour une définition d'interface. */
  void enterInterface(InterfaceDefinition itf) throws Exception;

  /** Génère le code de sortie d'une définition d'interface. */
  void leaveInterface(InterfaceDefinition itf) throws Exception;

  /** Visite d'une définition de méthode. */
  void visitMethod(Method method) throws Exception;
}
```

FIG. 7.8 – Interface de *visiteur* de définition d'interfaces.

Afin d'illustrer le résultat de génération de code effectué par de tels *plugins*, nous proposons de poursuivre notre exemple de client-serveur présenté dans la section précédente. Les figures 7.9 et 7.10

[6]Pour chaque type de protocole de communication supporté, doit être implanté un *plugin* différent.

illustrent les codes d'implantation des *stub* et *skeleton* générés selon le protocole JNI.

```
///// [enterInterface] : entrée //////
// Inclusion nécessaire pour accéder au macros de programmation à composants
#include <think.h>
#include <jni.h>
///// [enterInterface] : sortie //////

///// [visitMethod(afficher)] : entrée //////
JNIEXPORT void JNICALL Java_ServiceAdaptateurJava_method_AdaptateurAfficher(JNIEnv *env,
                                                                            jobject this,
                                                                            jstring arg) {
  /* Invocation de l'interface client connécté au composant serveur */
  CALL(serveur, afficher, arg);

}
///// [visitMethod(afficher)] : sortie //////
///// [leaveInterface] : entrée //////
///// [leaveInterface] : sortie //////
```

FIG. 7.9 – Implantation générée du composant *skeleton* `ServiceAdaptateurC`.

Le *stub* est implanté en Java de manière à fournir l'interface `Service` requise par le composant *client*. L'interface `Service` fournit uniquement une méthode qui permet d'afficher une chaîne de caractères. Le comportement associé à l'implantation de cette méthode par le *stub* correspond à l'invocation d'une fonction native fournie par l'intermédiaire de l'environnement JNI. Les commentaires tissés avant et après les morceaux de code permettent de distinguer l'opération de *visite* qui les a générés.

```
///// [enterInterface] : entrée //////
import java.util.*;
public class ServiceAdaptateurJava implements Service {
static {
        // Chargement statique du coté skeleton
        System.loadLibrary("ServiceAdaptateurC");
  }
///// [enterInterface] : sortie //////

///// [visitMethod(afficher)] : entrée //////
  // Référence de la méthode native implantée par le skeleton
  native void adaptateurAfficher (String arg) ;
  // Implantation de la méthode d'adaptation
  public void afficher(String arg) {
    adaptateurAfficher (arg) ;
  }
///// [visitMethod(afficher)] : sortie //////

///// [leaveInterface] : entrée //////
}
///// [leaveInterface] : sortie //////
```

FIG. 7.10 – Implantation générée du composant *stub* `ServiceAdaptateurJava`.

Le code d'implantation du *skeleton* est généré en C de manière symétrique. Ce composant implante une fonction réifiant la méthode `afficher` dans l'environnement JNI. Lorsque celle-ci est invoquée, elle transmet cet appel au composant serveur via son interface cliente qui lui est connectée.

7.4 Conclusion

Ce chapitre a présenté un outil de génération de code que nous avons développé en utilisant le canevas logiciel de construction de compilateurs ADL introduit dans le chapitre précédent. Cet outil de génération de code est conçu pour automatiser l'assemblage des architectures logicielles et pour simplifier la programmation des composants. Il supporte actuellement trois langages de programmation. Par ailleurs, le caractère extensible de l'architecture de cet outil lui permet d'être facilement étendu pour supporter de nouveaux langages de programmation. Par ailleurs, l'outil peut être étendu par des tierces-parties pour intégrer de nouveaux tisseurs de membranes ou pour produire différents types de résultats que ceux supportés actuellement.

Une autre caractéristique importante de l'outil de génération de code est sa capacité à analyser et à enrichir l'architecture du logiciel traité de manière à insérer automatiquement des adaptateurs de communication. Pour ce faire, un mécanisme générique s'intégrant dans le module de chargement est conçu de manière à pouvoir adapter les communications entre les composants qui le nécessitent. Par ailleurs, un patron de conception est fourni afin de simplifier la génération de code pour l'implantation des adaptateurs de communication. Cet outil d'adaptation de communications est actuellement utilisé dans divers projets de recherches, aussi bien pour assembler des logiciels hétérogènes (écrits en Java et en C) que pour développer des applications et intergiciels répartis à destination de systèmes multiprocesseurs sur puce.

Troisième partie

Applications

Chapitre 8

Construction de noyaux de systèmes d'exploitation

Sommaire

Cette dernière partie du manuscrit est dédiée à la présentation de deux applications misent en œuvre en utilisant le compilateur ADL présenté dans la partie précédente. La première application que nous présentons dans ce chapitre utilise cette infrastructure pour la construction de systèmes d'exploitation spécialisés. Dans ce cadre, nous réutilisons le canevas logiciel THINK que nous avons légèrement amélioré (i) en définissant un nouveau modèle de programmation[1], et (ii) en enrichissant l'outil de génération de code avec la capacité de générer des usines de composants permettant d'instancier et de détruire des composants dynamiquement. Cette dernière amélioration permet de réifier des ressources virtuelles à l'aide de composants.

Ce chapitre est organisé en deux sections. La première section dresse un état de l'art des travaux visant à la spécialisation de systèmes d'exploitation et positionne l'approche introduite par le canevas THINK dans cette perspective. La deuxième section présente la conception et l'évaluation d'une personnalité de noyau de système d'exploitation que nous avons mis en œuvre dans le cadre de nos travaux. Ce noyau implante la spécification de micro-noyaux L4 avec une architecture à base de composants à grain fin dont la conception privilégie les aspects architecturaux vis-à-vis des aspects liés à la performance.

8.1 Construction de systèmes d'exploitation spécialisées

Cette section a pour objectif de mettre l'approche THINK en perspective vis-à-vis des autres systèmes permettant de personnaliser ou d'étendre des systèmes d'exploitation. Nous débutons la section par la présentation des motivations pour la spécialisation des systèmes d'exploitation et par un survol rapide des différentes approches qui ont été proposées à ce jour. Nous décrivons ensuite la façon dont THINK permet de construire des systèmes par assemblage de composants toute en restant compatible avec les propositions précédentes. Enfin, nous présentons une extension de notre support ADL qui permet de réifier l'évolution dynamique de l'architecture des systèmes à base de composants.

[1] Ce modèle est décrit dans l'annexe A.1.

8.1.1 Spécialisation de systèmes d'exploitation

Le système d'exploitation constitue la brique logicielle de base se situant entre la plate-forme matérielle et les applications. Il a deux rôles principaux [Tan01] : le système d'exploitation doit tout d'abord fournir une vue virtuelle des ressources matérielles sous-jacentes pour simplifier la programmation des applications. Cela comprend d'une part l'abstraction des complexités liées à la configuration de la plate-forme matérielle (l'accès aux périphériques, la gestion de DMA[2], etc.) et d'autre part le multiplexage des ressources virtuelles sur des ressources physiques (gestion des processus, de la mémoire, etc.). Par ailleurs, le système d'exploitation doit assurer l'utilisation efficace et protégée des ressources matérielles.

La notion d'efficacité fait appel à la spécialisation des services implantés au sein du système d'exploitation pour mieux servir les programmes utilisateurs. Typiquement, dans le cas d'une application temps-réel, l'ordonnanceur du système doit répondre aux contraintes temporelles de cette dernière. Dans la plupart des cas, ceci n'est possible que si l'ordonnanceur du système est spécialisé ou configuré spécifiquement pour l'application en question. Notons que ce type de spécialisation est d'autant plus important pour assurer l'efficacité des processeurs spécialisés que l'on trouve dans la partie média (c'est-à-dire pour le traitement de données multimédia) des plates-formes multi-processeurs sur puce.

Si les systèmes d'exploitation à usage général comme Linux ou FreeBSD ont proposé des mécanismes pour étendre ou modifier certains de leurs modules, leur architecture monolithique ne permet pas d'effectuer des spécialisations à grain fin. De plus la taille qu'occupe ce type de systèmes en mémoire les rend difficilement compatible avec la partie média des MPSoC. Pour ces raisons, nous pensons qu'il est intéressant d'analyser les différentes solutions proposées dans la littérature pour la spécialisation de systèmes d'exploitation.

Les micro-noyaux de première génération comme Amoeba [TM81], Mach [RJO+89] et Chorus [RAA+92] ont introduit l'idée de minimiser les services systèmes s'exécutant en mode noyau afin de permettre aux programmeurs de mettre en place des serveurs spécialisés. La protection du reste du système vis-à-vis de ces serveurs écrits par les utilisateurs est garantie par l'isolation de ces derniers au sein d'espaces d'adressage différents. Le mécanisme d'appels inter-processus (IPC pour *Inter-Process Communication*) est utilisé pour permettre aux serveurs de communiquer. Ce mode de communication étant très coûteux, les micro-noyaux de première génération se sont avérés peu efficaces. Ce problème à été en grande partie résolue par les micro-noyaux de deuxième génération. En effet, il a été démontré qu'un certain nombre de règles de conception permettaient de réduire significativement le coût des IPCs [HHL+97] . Parmi les noyaux utilisant ce type de mécanismes, citons L4 [LES+97], QNX [Hil92] et Choices [CIRM93]. Plus tard, cette philosophie de minimisation des services systèmes a été encore plus accentuée avec l'approche des exo-noyaux [EK95]. L'idée est de réduire le noyau à un mécanisme de multiplexage et de protection permettant d'implanter le reste des services systèmes sous forme de bibliothèques utilisateurs. Aegis [Eng98, KEG+97] a été l'unique implantation de cette approche. Si les concepteurs de ce dernier ont démontré que des gains importants en performances pouvaient être obtenus grâce à l'approche exo-noyau, ils ont aussi montré que la mise en œvre d'un tel système était très complexe.

D'autres travaux se sont concentrés sur des approches de type langage pour la spécialisation de systèmes d'exploitation. Spin [BSP+95] et, plus récemment, Singularity[HLA+05] sont deux exemples de systèmes qui s'inscrivent dans ce cadre. Ces travaux proposent des langages sûrs pour la programmation des services systèmes spécialisés. L'objectif recherché est de permettre la construction de serveurs sûrs, ne nécessitant plus l'utilisation de mécanismes de protection de la mémoire. Cela permet de concevoir des systèmes à espace d'adressage unique, ce qui rend inutile l'utilisation de mécanismes de communication coûteux (e.g. IPC).

[2] DMA pour *Direct Access Memory*.

Un autre ensemble de travaux se sont intéressés à la reconfiguration dynamique des systèmes pour effectuer des spécialisations à la volée. VINO [SS94], Synthetix [PAB+95], MMLite [HF98], K42 [BHA+05, SAH+03] et YNVM [POF01, FPS+01] proposent des mécanismes pour la reconfiguration dynamique de certains composants d'un système d'exploitation. La plupart de ces travaux nécessitent l'utilisation de mécanismes de synchronisation ou d'interception sophistiqués tout au long de l'exécution du système pour capturer un état stable avant de procéder à la reconfiguration. Ces mécanismes engendrent des pertes de performances. Seul K42 propose un mécanisme qui permet de réagir uniquement après une requête de reconfiguration, améliorant ainsi les performances tout au long de l'exécution du système. En contrepartie, ce dernier fait un certain nombre d'hypothèses sur les propriétés des *threads* systèmes (notamment au niveau de leur durée de vie). Cela rend l'implantation de K42 non-triviale et réduit son champ d'application.

Enfin, une dernière classe de travaux visant à la spécialisation des systèmes d'exploitation comprend les approches de programmation à base de composants. Orthogonalement aux approches suscitées qui proposent des architectures pour l'implantation des noyaux de systèmes, celles-ci proposent des méthodologies de conception pour rendre les systèmes d'exploitation plus modulaires, et par conséquent facilement configurables. Scout [MP96], Click [MKJK99], OSKit [FBB+97], eCos[eco] et THINK [FSLM02] font partie de cette classe. Alors que les deux premiers systèmes se concentrent sur la construction de piles de protocoles de réseaux spécialisées, les trois derniers proposent des approches plus généralistes. Parmi ces derniers, OSKit a démontré qu'il était possible de construire des systèmes d'exploitation à partir d'une bibliothèque de composants en utilisant un outil d'assemblage comme Knit[RFS+00]. Plus tard, THINK a proposé un modèle de composants plus léger et plus flexible, permettant la mise en œuvre de composants à grain plus fin et pouvant interagir via un nombre arbitraire de protocoles de communications.

8.1.2 Approche THINK

Comme nous l'avons expliqué précédemment, THINK propose une approche de programmation à base de composants pour la construction de systèmes d'exploitation spécialisés. Think est complémentaire aux architectures de construction de systèmes mentionnées ci-dessus. En effet, [FSLM02] a démontré que THINK permettait l'implantation de diverses architectures, allant des micro-noyaux à des noyaux spécialisés. Par ailleurs, [POS06] a montré que THINK était aussi compatible avec différents mécanismes de reconfiguration dynamique (e.g. les mécanismes implantés dans MMLite et K42). Enfin, nous allons présenter dans la section suivante une expérimentation de réingénierie d'un micro-noyau L4 à l'aide du canevas THINK, d'une part pour démontrer la capacité de ce dernier à être utilisé pour le développement d'un noyau de taille réaliste, et d'autre pour discuter des intérêts d'une architecture à base de composants à grain fin pour la construction de tels systèmes complexes et critiques en robustesse. Mais avant d'aller plus loin, nous proposons ci-dessous de regrouper les différents éléments de ce canevas, afin de souligner les particularités qui le rendent adapté à la construction de systèmes d'exploitation.

8.1.2.1 Modèle de composants

Le canevas THINK fournit une implantation légère du modèle de composants FRACTAL. Dans une perspective de construction de systèmes d'exploitation, les avantages de ce modèle sont les suivants :

- La **composition hiérarchique** permet de réifier des parties différentes du système d'exploitation à des niveaux de complexité arbitraires. Par exemple, un gestionnaire de mémoire fournissant des services d'allocation et de désallocation peut être réifié par différentes architectures sous-jacentes (e.g. mémoire à plat ou paginée) alors qu'il est utilisé de la même manière par le reste du système.
- La **réflexivité des composants à l'exécution** permet de prendre en compte l'évolution dynamique de l'architecture du système. Par exemple, la création d'un nouveau processus, réifié par un com-

posant composite, peut impliquer la création de nouvelles instances de ses sous-composants et la modification de certaines liaisons au sein du système (e.g. liaison des *threads* du processus à l'ordonnanceur). Cette évolution peut être contrôlée via les mécanismes d'introspection et d'interception supportés par les composants.
- La **programmation des composants en C et en assembleur** permet l'implantation de tout type de service système.
- L'**indépendance vis-à-vis de la plate-forme d'exécution sous-jacente** permet l'utilisation de la programmation à base de composants jusqu'au plus bas niveau du système (e.g. gestion de mémoire et d'ordonnancement, etc.). Ceci est un des aspects distinctif entre THINK et OSKit, car ce dernier nécessite la présence d'une plate-forme de virtualisation pour exécuter les composants.

8.1.2.2 Bibliothèque de composants Kortex

Kortex est une bibliothèque de composants de systèmes d'exploitation fournie dans le canevas THINK. Elle contient de nombreux composants à grain fin fonctionnant sur des architectures matérielles variées. Sont brièvement décrits ci-dessous les différents éléments constituant la bibliothèque Kortex :

- La **couche d'abstraction de la plate-forme matérielle** (**HAL** pour *Hardware Abstraction Layer*) contient des composants qui sont spécifiques aux architectures matérielles sous-jacentes. Les composants qui font partie de cette couche sont des traitants d'événements, des pilotes de périphériques, des mécanismes d'ordonnancement et de gestion de mémoire de bas niveau, etc. Les plates-formes matérielles qui sont actuellement supportées incluent l'architecture PowerPC, l'architecture ARM (5, 7, 9) et l'architecture ST200.
- Les **composants de système d'exploitation** fournissent des implantations alternatives pour être assemblées dans des personnalités différentes de systèmes. Les composants faisant partie de cette bibliothèque comprennent des gestionnaires de mémoire (e.g. paginée, hiérarchique, etc.), des ordonnanceurs (e.g. à base de priorités, tourniquet), des piles de protocoles réseau (par exemple TCP/IP ou Bluetooth), des systèmes de fichiers (tels que VFS et Ext3), etc.
- Les **services spécifiques aux plates-formes multi-processeurs** fournissent les mécanismes de base pour construire des systèmes d'exploitation répartis. Cela comprend différentes implantations de composants de synchronisation et d'échange de données sur des plates-formes à mémoire partagée ou répartie.
- Enfin, les **services de surveillance** permettent d'instrumenter le système avec des sondes de performance. Parmi ces composants, citons les compteurs de cycles, les chronomètres, ainsi que les intercepteurs d'interfaces qui permettent de tracer les communication entre composants.

Remarquons que la bibliothèque Kortex est extensible. Cela permet aux concepteurs de définir de nouvelles interfaces et de rajouter de nouveaux composants en fonction de leurs besoins spécifiques. De plus, d'autres bibliothèques de composants peuvent être définies et intégrées dans le canevas THINK.

8.1.2.3 Support ADL

Le support ADL qui accompagne le canevas THINK dans sa version actuelle est la personnalité de génération de code que nous avons présentée dans le chapitre 7. Le rôle premier de ce support est d'assembler automatiquement un système à partir d'une description ADL. Ce mécanisme d'automatisation a un apport indéniable pour la spécialisation de systèmes d'exploitation. Outre la facilité d'instanciation d'une architecture à partir de descriptions ADL, ce support permet de garantir la cohérence entre la spécification architecturale du système d'exploitation et son implantation. Cet aspect est crucial pour garantir la maîtrise et faciliter la documentation des systèmes complexes.

En plus de cette fonction d'assembleur, le support ADL permet d'automatiser d'autres fonctions. Un des exemples d'extensions que nous avons intégrées au générateur de code permet de créer des usines de

composants à partir d'une description ADL et de l'implantation du composant. Une usine de composants C, au sens FRACTAL, est un composant qui fournit une interface *Factory*, permettant d'instancier et de détruire des composants de type C. Ce mécanisme simplifie grandement la programmation et la configuration des aspects dynamiques d'un système. Nous verrons par exemple dans la section suivante que les processus et les chaînes de communication IPC d'un système peuvent être instanciés par des usines dédiées, ce qui réduit la manipulation de ces structures complexes à quelques lignes de programme.

Remarquons que d'autres extensions de la chaîne de génération de code ont été faites dans le but d'automatiser la prise en charge de la reconfiguration dynamique. [POS06] présente le façon dont l'ADL peut être utilisé pour tisser différents mécanismes de reconfiguration dynamique au sein d'un système. Un travail complémentaire décrit dans [Maz06] a proposé l'exploitation des langages FScript/FPath [DL06] pour générer le code de reconfiguration des composants à partir de descriptions abstraites.

8.2 THINK4L : Une personnalité micro-noyau à base de composants

Cette section est consacrée à la présentation d'une personnalité de système d'exploitation que nous avons mise au point à l'aide du canevas logiciel THINK. Il s'agit d'une implantation d'un micro-noyau L4. Nous débutons la section par la présentation de L4. Ensuite, nous décrivons les choix d'implantation qui ont étés faits. Par la suite, nous présentons l'architecture à base de composants que nous avons mise en œuvre. Enfin, nous comparons la version à base de composants à la version originale afin d'évaluer notre approche sur des bases qualitatives et quantitatives.

8.2.1 Présentation de L4

L4 [LES+97], le successeur de L3 [Lie93], est un micro-noyau de deuxième génération. Les travaux dans le cadre de L4 ont pour but la définition d'une architecture de système d'exploitation flexible et sécurisée. La spécification de L4 [L4K06] est issue d'une réflexion visant à minimiser les services systèmes exécutés en mode noyau. Le critère de choix pour placer un composant dans l'espace du noyau est la fonctionnalité implantée par ce composant et non ses performances. L4 stipule que seuls les composants dont l'exécution s'avère impossible en mode utilisateur devraient être accueillis dans le mode superviseur. En l'occurrence, L4 définit trois concepts qui doivent rester dans le noyau : la gestion des espaces d'adressage, la gestion des fils d'exécution et la gestion des communications inter-processus.

8.2.1.1 Gestion d'espaces d'adressages

Un espace d'adressage spécifie une association entre une plage d'adresses virtuelles et une plage d'adresses physiques. Ce mécanisme a deux utilités. D'une part, il permet de localiser les programmes utilisateurs de façon transparente par rapport à leur adresse physique. D'autre part, il permet d'isoler les programmes dans des espaces distincts afin de les protéger des programmes malveillants.

Une partie du support pour les espaces d'adressage doit être fournie par la plate-forme matérielle. Il s'agit précisément du TLB (*Translation Lookaside Buffer*) qui permet de convertir les adresses virtuelles en des adresses physiques. Ce mécanisme est programmé au niveau logiciel pour mettre à jour les tables de traduction du TLB en fonction des pages qui sont allouées aux applications.

L4 suggère de mettre dans le noyau une couche logicielle minimale fournissant une abstraction du support matériel décrit ci-dessus. Cette couche minimale est nécessaire pour protéger la modification des entrées du TLB. Trois opérations sont fournies par cette couche d'abstraction. Premièrement, l'opération *map* permet de créer une nouvelle association dans une table de traduction. Deuxièmement, l'opération *grant* permet de donner une association déjà effectuée à une autre table de traduction. Cette opération est particulièrement utile pour concevoir des gestionnaires de mémoire hiérarchique sans causer de pertes de performances. Enfin, l'opération *flush* permet de vider les associations préétablies.

Au dessus de cette couche d'abstraction est construite une gestion récursive de l'allocation de mémoire. La base de cette gestion récursive est assurée par un serveur, appelé σ_0, qui est en charge d'allouer la totalité de la mémoire disponible lors de l'amorçage du système. Ce serveur est utilisé par des gestionnaires de mémoire de plus haut niveau pour effectuer des allocations de mémoire dynamiques.

8.2.1.2 Gestion de fils d'exécution

Un fil d'exécution, autrement dit un *thread*, est une activité qui s'exécute au sein d'un espace d'adressage. Un *thread* est défini par un ensemble de registres (y compris un pointeur d'instruction et un pointeur de pile), un registre d'état (e.g. privilège d'exécution, etc.) et un identifiant d'espace d'adressage. Parce que la modification de l'état et/ou de l'identifiant d'espace d'adressage d'un *thread* peut altérer l'état du système, une partie des informations relatives aux *threads* doivent être représentés et manipulés en mode superviseur. Alors qu'un certain nombre d'opérations comme la création ou l'ordonnancement de *threads* peuvent être implantées en mode utilisateur, certaines opérations critiques comme la mise à jour du contexte du processeur ou la modification de l'espace d'adressage associé au *thread* courant doivent être prise en charge par des composants encapsulés dans le noyau.

8.2.1.3 Communications inter-processus

Un mécanisme de communication inter-espaces, autrement dit inter-processus (IPC), doit être fourni par le noyau pour assurer l'interaction des *threads* exécutés dans des espaces d'adressage distincts. Plus précisément, ce mécanisme est nécessaire pour effectuer des échanges de messages entre différents espace d'adressage. D'autres sortes d'interactions comme l'appel de procédure à distance (RPC) peuvent être pris en charge en se basant sur ce mécanisme d'échange de messages.

L'IPC, tel que spécifié par L4, est une forme de communication point-à-point. Afin d'assurer l'intégrité des communications, un échange de données via IPC ne peut avoir lieu que s'il y a une connexion préétablie entre un expéditeur et un récepteur. L'opération d'établissement de connexion peut déclencher une suite de vérifications de droits. En effet, la communication entre espaces d'adressages peut être limitée, filtrée ou même chiffrée pour améliorer la sécurité du système.

Pour démarrer un échange de message, l'expéditeur doit définir l'endroit et la taille du message à transmettre, et le récepteur doit spécifier l'endroit et la taille des données qu'il est prêt à recevoir. Pour des raisons d'optimisation, les zones de mémoire qui peuvent être utilisées pour la communication par IPC sont préallouées à la création de chaque *thread*. Les *threads* ne peuvent modifier l'emplacement de cette zone. Afin de garantir la sécurité des communications, l'opération de copie de données entre la zone d'expédition et la zone de réception doit être effectuée par le noyau. Enfin, les opérations d'expédition et de réception peuvent être bloquantes si l'autre extrémité n'est pas prête. Dans ce cas, le noyau est en charge de geler l'exécution du *thread* courant pour donner la main à un autre thread.

8.2.2 Architecture de THINK4L

L'objectif principal de l'expérimentation présentée dans cette section était d'étudier l'architecture d'implantation d'un noyau de système d'exploitation conformément à la spécification fonctionnelle de L4. Cette expérimentation à été motivée par la volonté de remise à plat des considérations de conceptions d'une implantation industrielle de L4, Pistachio, développée par la société STMicroelectronics en collaboration avec l'Université de Dresden, dans le perspectif d'explorer différentes possibilités d'architectures d'implantation alignées sur le compromis de propreté architecturale et de l'efficacité de performances. Notre démarche a été d'analyser et de réorganiser cette implantation de L4 pour obtenir une architecture à base de composants. Cette implantation industrielle était conçue pour améliorer les performances. En contraste à cette approche, nous avons privilégié les considérations architecturales lors de la mise en œuvre d'une première version à base de composants afin de disposer des éléments de discussion

sur les avantages et les inconvénients d'une architecture propre à base de composants par rapport à une implantation monolithique privilégiant les performances.

Nous allons présenter par la suite l'architecture à base de composants du noyau L4 que nous avons construit. Notons que notre implantation n'est pas encore complète mais fournit tous les mécanismes nécessaires pour évaluer la communication IPC, le service système le plus critique d'un micro-noyau L4. Nous allons débuter cette sous-section par une illustration globale de l'architecture mise au point. Ensuite, nous allons décrire l'architecture des principaux composants du noyau.

8.2.2.1 Vue d'ensemble

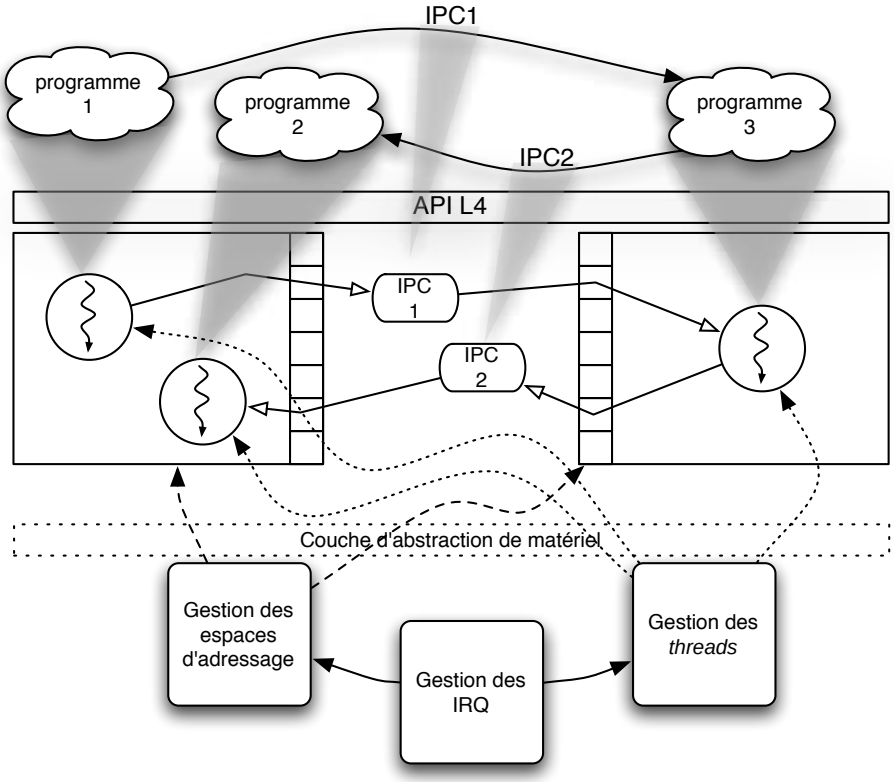


FIG. 8.1 – Vue d'ensemble de l'architecture de THINK4L.

La figure 8.1 présente un exemple d'architecture à l'exécution du système THINK4L. Cette architecture est composée de trois couches. La couche la plus haute représente les programmes utilisateurs. Dans le cas de l'exemple considéré, il existe trois programmes utilisateurs. Les *programmes 1* et *2* s'exécutent dans le même espace d'adressage, alors que le *programme 3* s'exécute dans un autre espace d'adressage. Ces programmes interagissent entre eux. Étant donné que le *programme 3* s'exécute dans un espace d'adressage distinct, il utilise des IPC pour communiquer avec les autres programmes.

En dessous de cette couche se trouve le noyau du système. La séparation entre les couches utilisateur et noyau est assurée par les interfaces de programmation (API pour *Application Programming Interface*) spécifiées par L4 [L4K06]. Le seul moyen de communication inter-espaces dans L4 étant des IPC, cette API définit des opérations de communication IPC (e.g. création, expédition, réception, destruction, etc.) ainsi que des appels systèmes (i.e. *syscall*). La couche noyau est décomposée en deux couches. La couche basse réifie les ressources matérielles, alors que la couche intermédiaire réifie les ressources virtuelles.

L'interaction entre ces deux couches est assurée par l'interface d'abstraction de la plate-forme matérielle.

Conformément au modèle décrit par les auteurs de L4 [LES+97], les composants d'abstraction des ressources matérielles sont au nombre de trois. Le gestionnaire des IRQ réifie le traitement des événements matériels. Le gestionnaire d'espaces d'adressage réifie le TLB en fournissant des opérations de programmation associées (e.g. *map*, *unmap*, *flush*, etc.). Enfin le gestionnaire de *threads* réifie le contexte d'exécution du processeur. Remarquons que ce composant implante le mécanisme de changement de contexte, ainsi qu'une politique d'ordonnancement de base permettant de supporter d'autres politiques via des serveurs s'exécutant dans la couche utilisateur.

Les composants présentés dans la couche intermédiaire réifient les ressources virtuelles allouées pour les programmes utilisateurs. En d'autre termes, ce sont des représentants des programmes utilisateurs. Parmi ces composants, citons les *threads*, les espaces d'adressage et les IPC. Ces composants peuvent directement accéder aux interfaces fournies par les composants de réification du matériel. En revanche, l'interaction entre les programmes utilisateurs et leurs représentants noyaux ne peuvent se faire directement pour des raisons de sécurité. En effet, l'ensemble des opérations faisant partie de l'API L4 déclenchent une exception de type *syscall* pour appeler leurs représentants. Ceci résulte en la modification de l'état du processeur pour passer en mode superviseur, afin de sécuriser les accès aux composants du noyau. Enfin, parce que l'architecture de la couche intermédiaire peut être modifiée en fonction de l'évolution des programmes utilisateurs (e.g. création de nouveaux programmes, fermeture de canaux IPC, etc.), les composants associés à cette couche peuvent être crées et détruits par des usines de composants[3].

8.2.2.2 Traitement des événements

Le traitement des événements matériels et logiciels est une tâche centrale dans un noyau L4. C'est notamment au travers de ce mécanisme qu'ont lieu toutes les interactions entre le noyau et les programmes utilisateurs. Pour cette raison, nous débutons notre présentation par la description des événements traités dans l'architecture à base de composants que nous avons mise au point.

Dans un noyau de système d'exploitation classique (e.g Linux, L4 :Pistachio), le traitement des événements est pris en charge par une unité centrale. Pour chaque événement produit, est exécutée une unité de traitement centrale qui est en charge de diriger ce dernier vers le traitant adéquat. Le traitement d'événements est alors implanté de manière hiérarchique (de l'unité la plus généraliste aux unités spécifiques). Ceci est une façon intuitive et classique pour implanter le traitement des événements dans un système.

L'architecture initiale que nous avons conçue pour THINK4L implantait le traitement des événements de cette manière. Très vite, nous nous sommes rendu compte que cette architecture présentait de faibles performances pour les événements de type *syscall* (qui sont en général utilisés pour effectuer des IPC) et pour les événements de gestion des défauts de TLB. Étant donné que ce sont les événements les plus fréquents et les plus critiques en termes de performances, nous avons proposé une autre architecture dans laquelle la hiérarchie de traitement est inversée.

La figure 8.2 illustre l'architecture que nous avons mise au point dans THINK4L. Dans cette architecture, est implantée une instance de traitant d'événements par *thread*. Le flot de traitement est organisé ainsi : lorsqu'un événement se produit, l'exécution est branchée à l'interface de traitement du composant *trap* du *thread* courant (au lieu d'être branchée à un composant central du noyau). Ce composant est en charge de diriger l'événement vers le traitant associé. En effet, ce composant de traitement général dispose d'une interface client de collection à laquelle sont connectés tous les traitants d'événements spécifiques. Ces composants spécifiques incluent les traitants de *syscall*, de défauts de *TLB*, d'exceptions logicielles et d'événements matériels. Notons que les événements matériels sont pris en charge par un composant central qui traite les interruptions (IRQ) du processeur.

[3]Les usines de composants ne sont pas présentes sur la figure 8.1 pour des raisons de lisibilité.

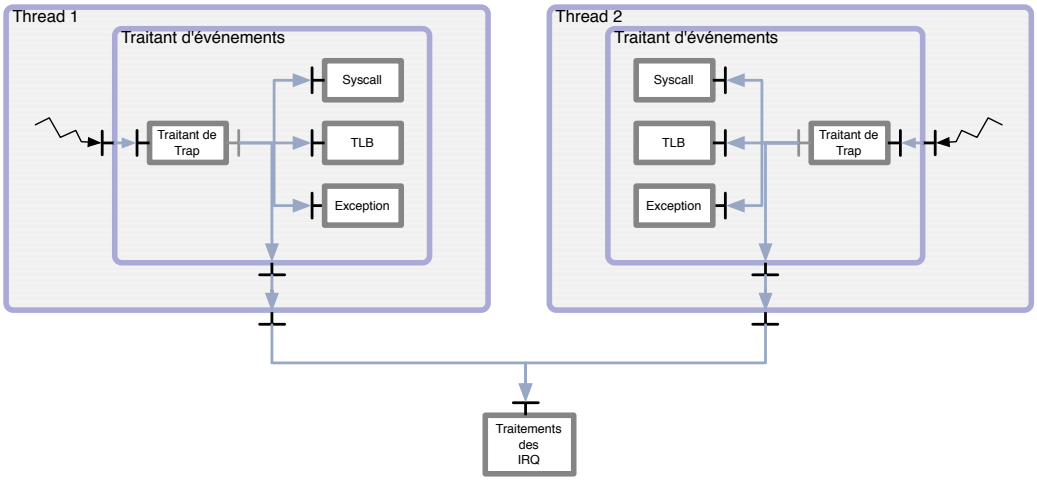


FIG. 8.2 – Architecture hiérarchique de traitement d'événements dans THINK4L.

L'architecture présentée dans la figure 8.2 permet de traiter au plus vite les exceptions liées à la communication et aux fautes d'accès à la mémoire. Cela permet d'atteindre de meilleures performances qu'une architecture de traitement classique, tout en restant généraliste[4]. Notons que pour améliorer les performances de traitement des IPC, une variation architecturale peut être envisagée pour connecter le composant de traitement du *thread* directement au traitant d'IPC. Bien que cette variation ne soit pas implantée à l'heure actuelle, nous estimons que ce compromis architectural peut engendrer des gains en performance de près de 12 %.

8.2.2.3 Gestion des espaces d'adressage

Les espaces d'adressage sont les unités d'isolation dans L4. Plusieurs *threads* peuvent être associés à un espace d'adressage, et ainsi partager leur mémoire. Pour cette raison, les mécanismes de gestion de la mémoire des *threads* situés dans un même espace d'adressage doivent êtres partagés entre ces derniers.

Pour répondre à ces besoins, nous avons réifié l'espace d'adressage par un composant qui encapsule des *threads* et qui leur fournit les mécanismes de gestion de mémoire associés. La figure 8.3 illustre l'architecture mise au point. Comme on le voit dans la figure, le traitant d'exceptions du TLB associé à chaque *thread* est connecté aux deux composants. Ces composants réifient respectivement la table de pages associée à l'espace, et le traitant des fautes de pages. En effet, lorsqu'une exception de type TLB est produite, le traitant d'exceptions consulte la table de pages pour vérifier si une association est déjà définie pour la plage de mémoire virtuelle concernée. Si oui, le gestionnaire central des espaces d'adressage est appelé pour enregistrer une nouvelle entrée au niveau du TLB. Pour ce faire, l'opération *map* fournie par le gestionnaire d'espaces d'adressage est utilisée en spécifiant la taille et l'adresse initiale des plages d'adresse virtuelles et physiques qui sont sujettes à l'association. Si aucune association n'est trouvée dans la table, la main est donnée au traitant de défauts de pages. Ce composant peut soit allouer une nouvelle page dans la table de pages pour augmenter la taille mémoire associée à l'espace courant, soit dérouter l'exécution vers une unité centrale qui va arrêter l'exécution du *thread* courant[5]. Enfin, le gestionnaire de *threads* est connecté au gestionnaire des espaces d'adressage pour mettre à jour le TLB lors d'un

[4]En effet, notre architecture est généraliste comparée aux traitants d'événements classiques qui privilégient certains traitements.

[5]Ces opérations non illustrées sur la figure sont réalisées par des requête IPC à destination du serveur d'allocation de mémoire et du gestionnaire de *threads*.

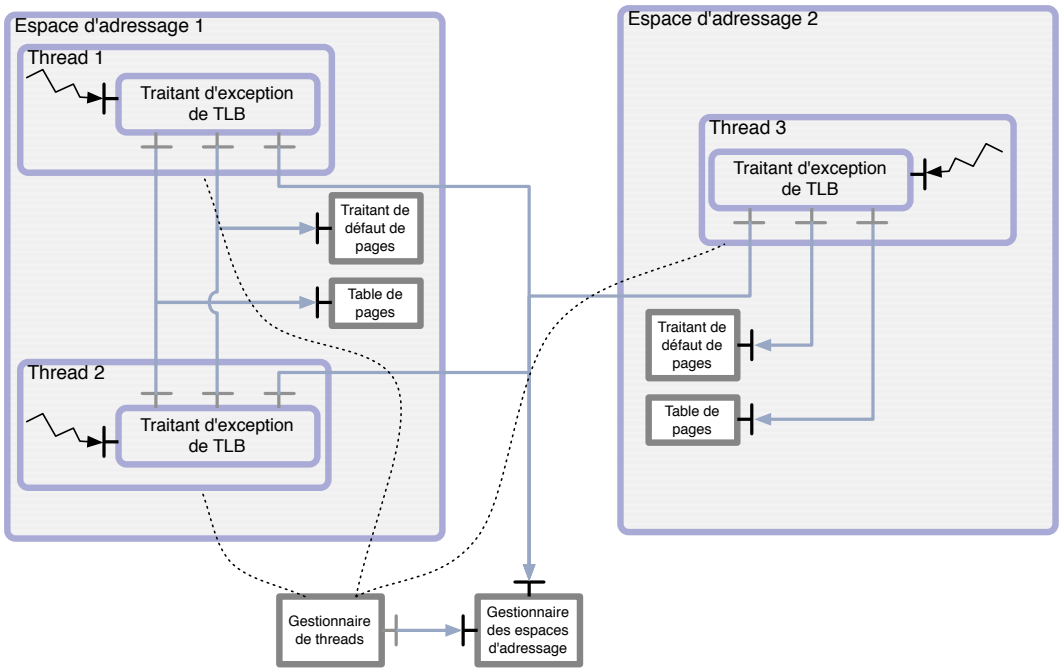


FIG. 8.3 – Architecture de gestion des espaces d'adressages dans THINK4L.

changement de contexte entre deux *threads* appartenant à des espaces distincts.

Remarquons que les composants d'espace d'adressage ne contiennent pas de mécanismes d'allocation de mémoire. Ceci est du au fait que ce type de mécanismes nécessite la prise en compte de politiques qui doivent être gérées au niveau de la couche utilisateur. Du point de vue du noyau, les requêtes d'allocation de mémoire sont des communications IPC en provenance d'un *thread* réifiant un programme utilisateur à destination d'un autre *thread* implantant le service d'allocation de mémoire.

Cette architecture de gestion d'espaces d'adressage nous semble satisfaisante, aussi bien au niveau architectural qu'au niveau des performances. D'une part, la structuration de l'espace à base de composants permet d'identifier les différentes interactions qui peuvent avoir lieu entre les éléments le constituant. Cela permet, entre autres, d'améliorer la maîtrise de la portée des différents éléments d'un tel sous-système, ce qui est nécessaire pour des raisons de sécurité. De plus, grâce au mécanisme d'usines de composants, les espaces sont créées et détruits sans avoir à se préoccuper de détails d'architecture interne. Une fois un espace instancié, de nouveaux *threads* peuvent êtres créés et rajoutés dans l'espace à l'aide d'opérations de configuration définies par FRACTAL (ajout/suppresion de composants, connexion/-déconnexion, etc.). Enfin, le fait d'implanter des composants de traitement de TLB propres à chaque *thread* permet de raccourcir le chemin de contrôle traversé lors du traitement des fautes de TLB.

8.2.2.4 Gestion des *threads*

Les *threads* sont des ressources d'exécution virtuelles qui permettent l'exécution des programmes utilisateurs. A chaque programme utilisateur qui a une activité propre correspond un *thread* le représentant au niveau du noyau (voir figure 8.1).

La figure 8.4 présente l'architecture d'implantation des *threads* dans THINK4L. Comme illustrée dans la figure, les *threads* sont des sous composants des espaces d'adressage. Cela représente la relation d'encapsulation entre des *threads* au sein des espaces d'adressage. Un composant *thread* est composé d'un traitant d'événement, d'un composant d'amorçage qui permet de démarrer l'exécution d'un *thread*

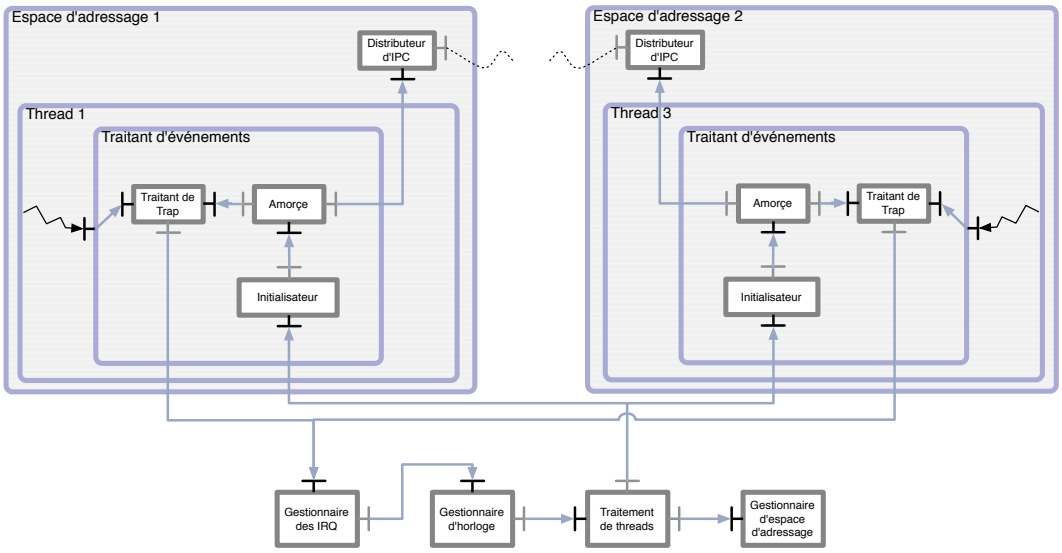


FIG. 8.4 – Architecture de gestion des *threads* dans THINK4L.

et enfin d'un composant *Job* réifiant l'état du *thread*.

Les *threads* peuvent être sujets à deux classes d'opérations. Il s'agit de leur création/destruction et de leur blocage/déblocage (sur des conditions de synchronisation). Sont expliqués par la suite les flots d'exécution de ces deux types d'opération au travers de l'architecture proposée.

Création/destruction : Les opérations de création/destruction sont fournies par des usines de *threads* (non illustrées dans la figure 8.4). Une fois qu'un nouveau *thread* est instancié par l'usine associée, il est inséré dans l'espace d'adressage correspondant et lié aux composants avec lesquels il doit interagir. Parmi ces connexions, celle qui le lie vers le composant central de traitement de *thread* a pour effet d'enregistrer le nouveau *thread* au sein de la liste d'ordonnancement. À ce moment, le point d'entrée du programme utilisateur n'est pas connu ; le point d'entrée enregistré au niveau du noyau est l'opération *run* fournie par le sous-composant *initialisateur* du *thread*. Lorsque le nouveau *thread* sera ordonnancé pour la première fois, son exécution commencera alors par le composant *initialisateur* qui invoquera le composant d'*amorçe* après avoir effectué certaines opérations d'initialisation. Le composant d'amorçe a pour rôle de récupérer via une communication IPC le point d'entrée du programme utilisateur. Une fois que ce point d'entrée est communiqué par le serveur correspondant, l'exécution est branchée au programme utilisateur et le cycle de vie normal du programme utilisateur commençe. Quant à l'opération de destruction,elle est effectuée par l'usine responsable de l'instanciation du composant *thread*. Naturellement, cette opération de destruction est effectuée après avoir mis à jour la configuration de tous les composants qui ont des liaisons vers le composant *thread* à détruire.

Blocage/deblocage : La réordonnancement des *threads* peut être causé soit par le blocage du *thread* courant, soit par un appel périodique de l'ordonnanceur. Le premier cas peut uniquement arriver dans le mode noyau. Dans ce cas, le composant de synchronisation demande au composant de *traitement de threads* d'être bloqué. Étant donné la nature non-préemtif du noyau L4, le deuxième cas peut arriver uniquement dans le mode utilisateur. Dans ce cas, l'exécution est déroutée vers le traitant d'événements du *thread* courant puis dirigée vers le traitant d'IRQ. Ce dernier appelle le decrémenteur d'horloge qui a pour fonction de demander la réordonnacement au composant de *traitement de threads*. Dans le cas où le réordonnacement implique le blocage du *thread* courant pour donner la main à un autre, ce dernier composant met en place le contexte gelé du *thread* à débloquer et déclenche ainsi son exécution.

8.2.2.5 Communication inter-processus

Le moyen de communication inter-processus constitue l'élément le plus critique d'un noyau L4. Les IPC assurent en effet toutes les interactions qui ont lieu entre des programmes se trouvant dans des espaces d'adressage distincts. Notons que le nombre de communications IPC est significativement plus important dans un micro-noyau que dans un système monolithique. Ceci est dû au fait que la plupart des services du système d'exploitation (e.g. allocation de mémoire, accès au réseau, etc.) se trouvent implantés dans le monde utilisateur, et par conséquent sont contactés via des IPC.

L'implantation des IPC n'a rien de trivial. Premièrement, la plupart des composants du noyau sont impliqués dans une communication IPC (e.g. gestionnaire de *threads* et d'espaces d'adressage, etc.). Il en découle que de nombreuses interactions doivent être prises en compte, notamment le fait que les IPC sont des ressources virtuelles partagés (entre espace d'adressages) qui peuvent être créées et détruites à la volée ou encore la gestion de leur cycle de vie ainsi que celle des ressources avec lesquelles ils sont liées. Par conséquent, il est important de les implanter avec une architecture propre et compréhensible afin de maîtriser leur cycle de vie et leurs interactions avec les autres composants du noyau. Troisièmement, la spécification de L4 impose le support d'une opération *recv_any* qui permet à un *thread* d'écouter sur tous les canaux d'IPC qu'il possède[6]. Cela rend l'implantation de l'IPC plus difficile (en termes d'efficacité) que celle d'une chaîne de communication simplement point-à-point. À ces deux aspects se rajoute la notion de *capacité de communication* qui définit des restrictions sur les droit d'expédition ou de réception des deux paires communicantes. Afin d'améliorer les performances, il est par exemple préférable de pouvoir simplifier le code de contrôle des IPC en fonction des restrictions de droits indiquées à leur création.

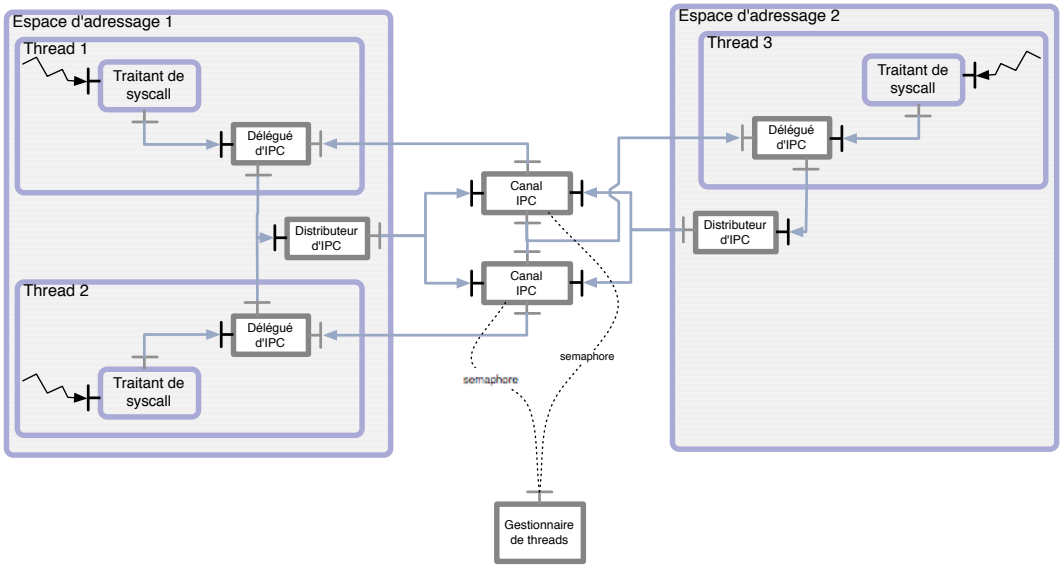


FIG. 8.5 – Architecture de gestion des IPC dans THINK4L.

Pour attaquer l'ensemble des problèmes cités ci-dessus, nous avons mis au point l'implantation des IPC à l'aide d'une architecture à composants à grain fin. Ce faisant, nous avons visé à bien distinguer les différents étages d'une communication IPC afin d'isoler les différentes ressources manipulées dans des composants séparés. L'architecture obtenue est illustrée dans la figure 8.5.

Nous pouvons remarquer que chaque canal de communication IPC est réifié par un composant diffé-

[6]L'opération *recv_any* permet de fournir une fonction équivalente à l'opération *select* de la norme POSIX.

rent. Les composants *canal IPC* implantent deux interfaces serveurs pour donner accès aux deux paires communiquantes. Cette interface serveur implante les opérations *send* et *recv* pour effectuer les opération d'expédition et de réception, respectivement. En plus de ses deux interfaces serveurs qui permettent au extrémités d'initier des communications, les *canaux IPC* sont connectés à des *déléguées d'IPC* qui se trouvent dans chaque *thread* pour effectuer le transfert effectif des données. En effet, les composants *délégués* réifient les zones de mémoire que les programmes peuvent utiliser pour les IPC. De plus, ces derniers fournissent une opération de consultation des communications en cours pour implanter l'opération *recv_all*.

Enfin, au sein de chaque espace se trouve un *distributeur* d'IPC. Le rôle de ce composant partagé par les *threads* d'un espace mémoire est de donner à tous les *threads* de l'espace la possibilité d'accéder à tous les *canaux d'IPC* associé à l'espace. Bien que cela rajoute une indirection dans le flot d'exécution des IPC, cette fonction nous semble très utile. Par exemple, via ce moyen, un programme peut ouvrir des canaux vers différents services systèmes et partager ces accès avec les autres programmes se trouvant dans son espace.

Un avantage indéniable de cette architecture est l'isolation de l'exécution concurrente au sein des canaux d'IPC. En effet, étant donné qu'un seul *thread* peut être exécuté à la fois dans le noyau [7] et que les seuls composants de la chaîne de communication IPC possédant des moyens de synchronisation sont les canaux IPC, seuls ces dernier peuvent être exécutés par deux *threads* à un moment donné. Ce type de cas peut uniquement arriver si une opération d'expédition est initiée alors que le récepteur n'est pas encore prête (ou dans le cas symétrique). Dans ce cas, le *thread* expéditeur reste bloqué dans le canal IPC jusqu'à ce que le récepteur vienne le débloquer. Cette isolation permet de simplifier le code de la chaîne d'IPC qui a pour résultat d'améliorer les performances et de réduire les sources d'erreur.

Enfin, une autre optimisation mise en place consiste en la spécialisation des canaux d'IPC en fonction des droits de communication. En effet, la spécification du micro-noyau L4 permet de définir des contraintes sur les droits d'expédition et de réception à la création de chaque canal IPC. Les implantations existantes de L4 proposent une seule implantation générique de canal IPC. La vérification des contraintes de communication est alors effectué à chaque communication en se basant sur les propriétés du canal utilisé. Dans notre implantation à base de composants, nous avons mis en place des types de canal IPC différents pour chaque modèle de contraintes possibles[8], ce qui nous a permis de supprimer les vérifications suscitées. Ainsi, à chaque ouverture de canal IPC est instancié un composant dédié au contraintes de communication sollicitées. Cette spécialisation a permis de diviser par deux le nombre de tests effectués à chaque opération d'expédition et de réception au travers des canaux IPC, par rapport à la version générique des implantations existantes.

8.2.3 Évaluation

Nous consacrons cette section à l'évaluation de THINK4L. Nous séparons notre analyse en deux parties pour distinguer les aspects qualitatifs des aspects quantitatifs.

8.2.3.1 Aspects qualitatifs

Bien qu'elle constitue un paramètre d'évaluation important, la qualité d'un programme est souvent négligée. Or, ceci est d'autant plus important dans le cas des logiciels critiques tels que les systèmes d'exploitation. Nous dressons ci-dessous notre retour d'expérience sous différents volets allant de la documentation jusqu'à la validation du logiciel mis en œuvre.

[7] Le micro-noyau L4 est non-préemptif.

[8] Les trois modèles de contraintes possibles sont expédition seule, réception seule, et expédition/réception.

Documentation : Un des aspects qui a initialement motivé le projet THINK4L était la documentation du noyau L4 développé au sein de la société STMicroelectronics. L'expérience a montré que l'expression du noyau en termes de composants, interfaces et connexions a dramatiquement simplifié la documentation. En effet, le fait de bénéficier d'un outillage de génération de code qui assemble les composants à partir des descriptions d'architecture permet d'assurer la correspondance entre l'architecture et l'implantation. Les schémas d'architecture présentés dans ce manuscrit sont dessinés à la main en se basant sur les fichier ADL du THINK4L et reflètent (à quelques simplification près) l'architecture d'implantation actuelle. Dans ce cadre, l'une des perspectives envisageables est de mettre en œuvre une extension de notre support ADL dédiée à la génération de représentation graphique des composants. Un tel outil simplifierait encore plus la tâche de documentation.

Maintenance et évolution : Ces deux aspects sont cruciaux dans le cadre de la programmation à grande échelle (ou haut niveau d'abstraction)[9]. En effet, bien que la taille d'un noyau L4 en termes de ligne de code soit raisonnable (15000 lignes de code), la maîtrise du système reste non triviale. Ceci est en particulier dû à la durée développement et de maintenance du projet qui rend les programmeurs (même les plus expérimentés) étrangers au code qu'ils ont développé. Après avoir discuté avec les programmeurs de L4 :Pistachio, nous constatons ensemble que l'organisation à base de composant mise au point simplifie la maintenance grâce au découpage du logiciel en composants à grain fin[10] et à l'identification des interactions à l'aide des liaisons explicites.

Validation : La validation du logiciel mis en œuvre est une étape cruciale du processus de développement, en particulier dans le cadre des applications critiques comme des systèmes d'exploitation. Un des moyens de validation le plus fiable, qui est aisément déployé dans la conception de circuits électroniques, consiste en tester les blocs d'un système séparément, en les intégrant dans des environnements spécifiques. La réussite de ce type de tests unitaires est lié à deux aspects. Premièrement, les blocs doivent permettre d'identifier aisément leurs points d'interaction avec le mode extérieur afin de déduire les vecteurs de tests à exécuter. Cette démarche qui semble assez difficile dans des programmes monolithiques écrits en C ou en C++ est considérablement simplifiée par l'approche à composants qui permet précisément d'identifier les points d'interaction de chaque module à l'aide des interfaces serveurs et clients. Deuxièmement, il doit être relativement facile de mettre en place des environnements de vérification pour tester les composants. Le compilateur ADL que nous avons présenté dans ce manuscrit répond tout à fait à ce besoin en permettant de générer le code d'une application pour un assemblage exprimé en ADL. Par ce biais, les concepteurs peuvent mettre en place des suites de tests unitaires en décrivant de multiples environnements permettant de vérifier le comportement d'un même composant dans des contextes différents.

Fiabilité : Le noyau d'un système d'exploitation doit être fiable. Bien que la fiabilité d'un logiciel soit liée à plusieurs aspects, nous pouvons citer l'importance d'une bonne architecture qui a pour effet d'améliorer la maîtrise et la validation du logiciel. Le niveau de fiabilité peut être encore augmenté en automatisant le processus de développement et de validation. L'approche à base de composants et les outils de génération de code que nous avons utilisés dans le développement de THINK4L nous ont aidé à aller dans ce sens. Par exemple, la génération automatique des usines de composants à partir de descriptions d'architecture nous ont permis de réduire la création/destruction des structures complexes telles que des espaces mémoire ou des IPC en un appel de fonction.

Portage : Le découpage du noyau à l'aide des composants permet d'identifier très clairement l'interface entre les composants dépendants d'une architecture et ceux qui sont indépendants. Notons que notre

[9] *Programming in the large*.

[10] Le nombre de lignes de code écrites pour implanter un composant varie entre 100 et 400.

approche est légèrement différente de l'approche classique consistant en la définition d'un API de couche d'abstraction de la plate-forme matérielle. En effet, la bibliothèque Kortex propose non pas *une interface* pour abstraire la plate-forme matérielle mais une *collection d'interfaces* permettant d'abstraire différents éléments de cette dernière (e.g. TLB, IRQ, etc.). Cela permet aux concepteurs d'exploiter au mieux les ressources physiques à disposition en ajustant certaines interfaces et en supportant les modifications apportées via des composants appropriés.

Réutilisation : Enfin, nous constatons qu'un nombre important des composants de la bibliothèque Kortex ont été réutilisés pour démarrer le projet THINK4L. Une fois une implantation fonctionnelle préliminaire mise en place, nous avons enrichi Kortex avec des interfaces et des composants plus adaptés aux besoins du noyau L4. Ces composants restent naturellement à disposition pour être réutilisé dans des projet de développement futurs.

8.2.3.2 Aspects quantitatifs

Comme nous l'avions déjà précisé, l'implantation actuelle de THINK4L ne satisfait pas l'ensemble de l'API du noyau L4. En revanche, l'ensemble des fonctions qui sont nécessaires à l'évaluation des performances des échanges IPC sont en place. Dans la suite, nous présentons les micro analyses effectuées au niveau de la communication et de la création des IPC. Nous discutons ensuite des perspectives d'optimisations envisagées. Enfin, nous argumentons sur la taille du noyau mise en œuvre actuellement, et les améliorations envisagées dans le futur.

Environnement d'évaluation :

Nos évaluations sont effectuées sur un simulateur de plate-forme embarquée à base d'un processeur ST200. ST200 est un processeur de type VLIW (*Very Long Instruction Word*), conçu par la société STMicroelectronics. Son architecture permet d'exploiter un parallélisme au niveau instructions afin d'effectuer des calculs massifs pour des traitements de type multimédia. Il est typiquement déployé dans la partie média d'un système multiprocesseur sur puce. En effet, le choix de cette plate-forme est justifié par deux aspects. Premièrement, celle-ci est la plate-forme utilisée pour le développement actuel du projet L4 au sein de STMicroelectronics, ce qui nous permet de considérer une base de comparaison pour évaluer les résultats de THINK4L. Deuxièmement, l'utilisation du simulateur de ST200 nous permet de tracer le flot d'exécution à un grain très fin, sans perturber le système exécuté.

Coût de la communication IPC :

Afin d'évaluer le coût d'une transmission de donnée via le mécanisme d'IPC, nous choisissons d'analyser le flot d'exécution d'un appel *send/recv*. Cette opération, souvent utilisée pour implanter des appels de procédures à distance, permet en un appel système d'envoyer un message IPC et d'attendre une réponse. La figure 8.6 représente le flot d'exécution d'un tel appel. La partie gauche de la figure illustre les fonctions qui sont appelées tout au long de l'exécution de cette opération. La partie centrale présente le flot d'exécution en illustrant le nombre de cycles consommés à chaque étage. Enfin, la partie droite présente le nombre de cycles totaux utilisés pour l'exécution de chaque fonction appelée, en extrayant le temps d'exécution des sous-fonctions. Ceci nous permet de connaître le coût de chaque étage, indépendamment des étages sous-jacents.

Selon le scénario d'exécution mis en place, une extrémité invoque l'opération IPC *send/recv* alors que l'autre extrémité se trouve déjà dans l'attente de réception. L'invocation de l'opération déclenche un événement système. Ceci branche l'exécution vers le traitant d'événement du *thread* courant. Les trois premières lignes de la figure représentent les opération de sélection de traitant qui ont lieu dans les différents étages du traitement d'événements. Une fois que l'événement est arrivé au délégué des IPC, le traitement effectif de la communication démarre. L'opération *send* est transmise du délégué jusqu'au

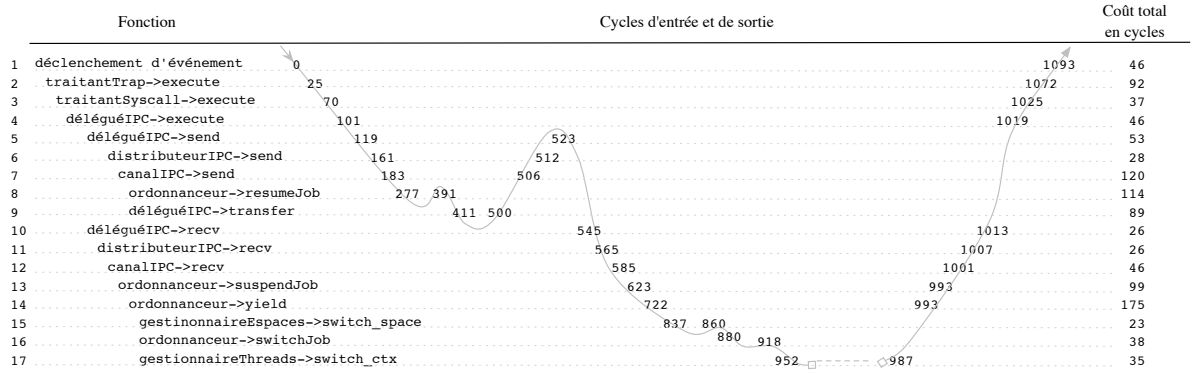


FIG. 8.6 – Analyse en nombre de cycles du flot d'exécution d'un appel IPC send/recv.

canal concerné en passant par le distributeur d'IPC (voir la figure 8.5). Ce dernier débloque l'autre extrémité de la communication et effectue le transfert de données. Une fois les données transférées, le délégué d'IPC déclenche l'opération *recv* qui a pour effet d'enregistrer cet appel auprès du canal concerné et de le suspendre lui-même dans l'attente de la réponse. La ligne pointillée qui commence à cet instant représente le temps passé dans cette attente passive. Pour simplifier l'analyse, nous avons soustrait des chiffres présentés le temps d'attente réellement consommé à ce niveau dont le temps passé dépend du temps d'exécution de l'autre extrémité de la communication. Par conséquent, le *thread* analysé peut être considéré comme immédiatement débloqué après la commutation de contexte. A partir de ce moment, le *thread* remonte dans sa pile d'exécution afin de ressortir du traitement d'événement de manière à brancher le flot d'exécution au niveau du programme utilisateur.

Au total, cette opération prend 1093 cycles (voir la figure 8.6) contre 276 cycles pour une opération identique dans le cas du noyau L4 monolithique. Ceci correspond à un rapport de 3,96 en faveur de l'implantation monolithique. Or, les raisons de cette perte de performance sont partagées entre la généricité de l'architecture à base de composants que nous avons mis au point et les coûts intrinsèques au format d'implantation des composants selon le canevas THINK. Afin d'identifier les pistes d'optimisations envisageables, nous avons comparé le flot d'exécution présenté dans la figure 8.6 avec celui produit par l'implantation monolithique[11]. Cette analyse nous a montré que les optimisations et les compromis architecturaux discutés ci-dessous pourraient résulter en des optimisations de performances importantes du noyau THINK4L.

Optimisation du chemin de distribution d'événements Notre architecture implante un mécanisme de distribution d'événements générique où tous les événements sont considérés au même niveau de priorité. Or, les IPC constituent le type d'événement le plus produit, et le plus critique en termes de performances. Pour cette raison, leur traitement pourrait être géré de façon prioritaire. Ainsi, le premier étage de traitement d'événements peut expédier les IPC directement au traitant concerné, de manière à court-circuiter les opérations de routage illustrées sur les lignes 2 et 3 de la figure 8.6. Bien que nous ayons pas implanté cette variation, nous estimons, grâce à une analyse au niveau du code assembleur concerné, que celle-ci peut se traduire en un gain de 129 cycles, soit un gain de 12%.

[11]Le flot d'exécution détaillé de la version monolithique n'est pas présenté dans ce document en raison de son caractère industrielle.

Élimination des frontières de composants Comme nous l'avions précisé précédemment, les composants THINK sont présents à l'exécution et que le format binaire des interfaces est implantée de façon similaire au format binaire des composants COM, c'est à dire en utilisant des tables de fonction virtuelles. Ce format d'implantation permet la modification dynamique des liaisons mais ajoute des indirections lors des invocations. Ainsi, le coût d'un appel d'interface est évalué à 10 cycles en moyenne[12]. Or, l'architecture des composants impliqués dans la communication de type IPC n'est plus modifiée après le déploiement d'un canal. Par conséquent, une technologie d'optimisation permettant d'entrelacer (au sens *inlining*) les composants peut être employée. À ce jour, nous ne disposons pas d'une telle technologie. Notre analyse du nombre d'appels d'interfaces qui ont lieu au cours d'une opération *send/recv*[13] montre qu'une telle optimisation peut permettre un gain de près de 30% en temps d'exécution.

Implantation d'une opération send/recv optimisée Notre implantation de l'opération *send/recv* est basée sur des appels successifs aux opérations *send* et *recv*. Or, cette opération est implantée spécifiquement dans le noyau monolithique auquel on compare le noyau THINK4L. Ainsi, l'opération *send/recv* du noyau monolithique suit un chemin d'exécution plus court. De plus, selon ce dernier, le flot d'exécution est amené directement à l'extrémité réceptrice de manière à court-circuiter le mécanisme d'élection de l'ordonnanceur. L'implantation d'un tel comportement plus spécifique permettra d'éliminer l'exécution des lignes 8 et 15 et ainsi de réduire le temps d'exécution total de près de 25%.

Enfin, d'autres types d'optimisations de plus bas niveau expliquent l'efficacité du noyau monolithique. Parmi ces optimisations, citons la modification de la pile d'appel d'un *thread* dans le cas d'une erreur survenue lors de la communication IPC de façon à exécuter le traitement des cas d'erreurs. Ceci permet d'optimiser les tests de valeurs de retour à chaque exécution en implantant un mécanisme d'exception ad-hoc. Bien qu'elle permette des améliorations nettes en performances, l'utilisation de tels mécanismes d'optimisation rend la maîtrise du code plus difficile, ce qui peut par exemple résulter en l'ouverture de *trous de sécurité*.

Coût de déploiement d'une chaîne de communication IPC

Le coût de déploiement d'un canal de communication IPC dans le noyau THINK4L est près de 10 fois plus lent que l'opération identique dans le noyau monolithique. Ceci s'explique par l'utilisation massive des opérations réflexives lors de l'instanciation et de la liaison des composants constituant le canal IPC. Nos analyses détaillées montrent que plus de 70% du temps est passé dans ces opérations permettant de retrouver et lier des interfaces contre 10% du temps passé pour l'instanciation de l'ensemble de composants impliqués. L'explication d'un tel coût en ce qui concerne les opérations réflexives est l'utilisation des représentations textuelles des noms d'interfaces et de composants. En effet, la plupart du temps de déploiement correspond à comparaisons de chaînes de caractères. Or, d'autres types d'implantations peuvent êtres envisagés pour réduire ces coûts. Typiquement, des références uniques peuvent être générées par le compilateur ADL de manière à transformer ces opérations en des comparaisons d'entiers. Nous croyons que le support pour une telle optimisation peut permettre des améliorations dramatiques, de près de 70%.

[12] Il s'agit du temps d'invocation d'une fonction avec 3 paramètres formels, sans prise en compte du temps d'accès à la mémoire. Lorsqu'un modèle de mémoire réaliste est pris en compte, les défauts de cache qui peuvent être causées par les indirections peuvent augmenter le nombre de cycles d'une quantité très importante, en fonction de la latence de l'architecture de mémoire.

[13] Notons que la figure 8.6 n'illustre pas la totalité des invocations d'interfaces. Typiquement, les appels aux sémaphores utilisés pour la synchronisation des canaux d'IPC ne sont pas représentés.

Taille en mémoire

À ce jour, la taille du noyau en mémoire correspond à 274 kilo octets. Or, seulement un tiers de l'API L4 est actuellement supportée. Nous estimons que la taille du noyau s'élèvera à près de 330 kilo octets lorsque l'ensemble de l'API sera supportée. Ceci correspond à une taille supérieure de 43% comparée au noyau monolithique. Or, près de 66% de l'image du noyau est occupée par des données à lecture unique, et près de 80% de ces données correspondent aux chaînes de caractères liés aux opérations réflexives. Dans ce cadre, le remplacement de ces chaînes de caractères par des valeurs numériques suggéré ci-dessus, résulterait en un gain de près de 150 octets. Par conséquent, nous croyons que la taille du noyau peut descendre aux alentours de 200 kilo octets, soit une taille inférieure à celle du noyau monolithique. Ce gain peut être expliqué par les deux aspects suivants :

- L'utilisation du langage C au lieu du langage C++ (utilisé pour l'implantation du noyau monolithique).
- La conception à base de composants à grain fin qui permet de contextualiser le code, et par conséquent de diminuer le nombre des cas à prendre en compte. Ceci a pour effet direct de réduire la taille du code.

Enfin, notons que ces valeurs sont approximatives. La stratégie d'optimisation mentionnée devrait être implantée au niveau de l'outil de génération de code pour la validation de ces chiffres.

8.3 Conclusion

Ce chapitre a présenté la mise en œuvre d'un micro-noyau de système d'exploitation respectant la spécification fonctionnelle de L4. L'objectif principal de ce travail était d'étudier une implantation existante de ce type de micro-noyau, d'analyser son architecture interne, et de proposer une architecture modulaire, à base de composants, afin d'évaluer la pertinence de cette approche dans le cadre de la programmation de systèmes d'exploitation embarqué. Nos travaux de développement se sont plutôt concentré sur les mécanismes d'isolation d'un micro-noyau L4, c'est-à-dire les espaces d'adressages et les canaux de communication inter-processus (IPC).

Après avoir présenté les détails de l'architecture proposée, nous avons dressé dans ce chapitre, une évaluation quantitative et qualitative de notre implantation. Les aspects quantitatifs soulignent des pertes de performances relativement importantes. Ceci est dû d'une part au format binaire des composants (e.g. indirections au niveau des interfaces, coût de structure de données, etc.), d'autre part à l'architecture très modulaire que nous avons proposé (e.g. adoption d'une granularité très fine de composants, etc.). Ainsi, nous avons discuté d'un certain nombre d'optimisations qui pourraient être mise au point dans le futur. Parmi ces optimisations, certaines pourraient être intégrées dans la chaîne de génération de code sous forme de *plugin*, d'autres nécessiteraient des compromis de conception entre la clarté architecturale et les performances. Enfin, concernant l'évaluation quantitative, la programmation à base de composants, supportée par une chaîne de compilation basée sur des descriptions d'architecture, semble très satisfaisante pour la maîtrise de l'implantation d'un tel micro-noyau en termes de la documentation, de la validation, de la maintenance et de fiabilité.

Chapitre 9

Construction d'applications de streaming réparties

Sommaire

Les applications multimédia constituent l'un des domaines applicatifs majeurs de systèmes embarqués tels que téléphones portables ou assistants personnels électroniques (PDA). Les applications multimédia ayant en général affaire à des flux d'entrée continus, elles sont considérées comme des applications de type *streaming*. Une application de *streaming* au sens large, est un logiciel qui applique une fonction de transformation à un flux d'entrée continu pour produire un flux en sortie qui est lui aussi continu. Un exemple représentatif de cette classe d'applications est un logiciel de décodage de vidéo qui reçoit en entrée un flux MPEG-2 et qui l'affiche au fur et à mesure le résultat sur un écran graphique après avoir réalisé l'opération de décodage adéquate.

Les applications de *streaming* sont en général modélisées sous forme de *pipelines*. Par exemple, un égaliseur de flux audio peut être assemblé à partir d'un ensemble de filtres passe-bande mis en parallèle, qui reçoivent leur flux d'entrée par un composant de diffusion multipliant le signal en entrée et qui envoient le résultat vers un composant mixant les résultats de chacun pour obtenir à nouveau un signal unique. Dans une telle architecture, chaque composant (les filtres, le mixeur, etc.) constitue à lui seul une application de streaming étant donné qu'il applique une fonction de transformation à un flux continu. Cela montre que la programmation à base de composants s'applique naturellement au développement des application de *streaming*. Par ce biais, une méthodologie d'assemblage peut être aisément adoptée pour composer de telles applications à partir d'autres de taille plus petite, de manière récursive.

Nous consacrons ce chapitre aux techniques de programmation pour les applications de *streaming*. Nous commençons par présenter une première expérience visant à évaluer les avantages et les inconvénients d'un processus de développement à base de composants pour mettre au point un décodeur de flux H.264. Cette réalisation nous a permis d'identifier un certain nombre perspectives afin d'améliorer la programmation d'applications multimédia. Dans une seconde partie, nous décrivons les extensions que nous avons intégré au niveau de la chaîne d'outils afin d'améliorer la construction des applications multimédia parallèles à destination des plates-formes multiprocesseurs embarquées. Enfin, nous présentons une autre expérimentation que nous avons réalisée en utilisant les extensions du compilateur ADL et évaluons les gains obtenus.

9.1 H.264 - Une expérimentation de mise en œuvre d'application multimédia

Afin d'évaluer l'efficacité de la programmation à base de composant dans le contexte des applications multimédia, nous nous sommes proposés de concevoir une application réelle et de l'évaluer sur des plates-formes d'exécution de type MPSoC. L'application que nous avons considérée dans ce cadre est un décodeur vidéo qui fonctionne selon la norme H.264 [H2603]. H.264, connue aussi sous le nom de couche 10 de MPEG-4 [MPE98], est une spécification de compression de flux vidéo qui s'intègre dans la norme MPEG-4 pour remplacer le standard précédent intitulé H.263. Par rapport à ce dernier, H.264 fournit une meilleur qualité d'image avec une définition plus haute tout en assurant des taux de compression sensiblement plus élevés. Cette norme est de plus en plus utilisée dans la transmission de la télévision en haute définition ainsi que dans les applications nomades de type téléphones portables en raison de ses performances élevées.

Le code que nous avons pris en main est l'implantation de référence de la norme H.264 [JM]. Ce logiciel qui contient près de 26000 lignes de code en C est produit par une équipe intitulée *Joint Video Team*, qui réunit de nombreux programmeurs du monde académique et de l'industrie.

Par le biais de notre expérience, nous avons cherché à répondre aux questions suivantes :

- Est-il possible et raisonnable de transformer une application monolithique écrite en C en une application à base de composants ?
- Quels sont les avantages d'un processus de développement basé sur l'architecture logicielle en ce qui concerne les applications multimédia, et en particulier à destination des plates-formes multiprocesseurs sur puce ?
- Une version à base de composants à grain fin d'une application multimédia est-elle moins performante qu'une version monolithique classique ?

Nous présentons dans la suite de cette section, les différentes étapes de notre expérimentation. Nous y dresserons, entre autres, une méthodologie de *componentisation* que nous avons mise au point, ainsi que les différentes configurations architecturales que nous avons pu réaliser à partir d'une même application grâce à sa structure à composants. Enfin, nous présenterons les conclusion que nous avons tirées de cette expérimentation.

9.1.1 Méthode de restructuration d'une application monolithique en composants

Une méthodologie a été mise au point pour effectuer la transformation du code originel en une version à base de composants. Cette méthode permet de partir d'un code monolithique et de le découper en plusieurs composants de manière incrémentale. Elle se résume à l'itération des étapes illustrées en figure 9.1. Nous présentons ci-dessous chacune des étapes de ce schéma :

1. Créer un composant qui encapsule l'ensemble des fichiers d'implantation de l'application. A ce stade, on obtient un composant dont l'implantation est monolithique.
2. Identifier un fichier C (ou un module) à transformer en un composant. Déterminer les fonctions qu'il fournit et qu'il appelle afin d'identifier les interfaces qu'il faudrait mettre en place afin d'expliciter ses interactions avec son environnement. Définir en IDL, les interfaces telles ainsi identifiées. Faire un effort pour définir des interfaces réutilisables qui regroupent des opérations logiquement proches.
3. Définir en ADL l'architecture du composants à séparer. À ce stade, il s'agit simplement de décrire les interfaces clientes et serveurs du composant identifié, ainsi que ses attributs s'il y en a.
4. Transformer le fichier d'implantation en une version à base de composants en respectant le guide de programmation associé au langage d'implantation. Dans le cadre de cette expérimentation, il s'agit d'utiliser le guide de programmation fourni pour le canevas THINK.
5. La réécriture des fonctions implantées sous forme des interfaces fait qu'elles ne sont plus accessibles directement par la partie monolithique de l'application. Écrire des composants adaptateurs qui redéfinissent ses fonctions de manière à établir un pont entre la partie monolithique et la partie à base de composants. Noter que ces composants adaptateurs servent uniquement à tester le bon fonctionnement de l'application à un stade intermédiaire. Ils seront retirés au fur et à mesure que le nombre de composants augmente. Il n'en restera plus aucun lorsque l'application sera entièrement écrite à base de composants.
6. Intégration du nouveau composant ainsi que de ses adaptateurs dans l'architecture. Établir les liaisons avec les autres composants. Retirer les composants adaptateurs qui ne sont plus requis après l'établissement des liaisons entre composants. Tester l'équivalence fonctionnelle de la version obtenue avec la version originale.
7. Vérifier si l'application est entièrement transformée en composants de granularité désirée. Si oui, terminer l'opération. Dans le cas contraire, retourner à l'étape 2 pour continuer à transformer l'application.

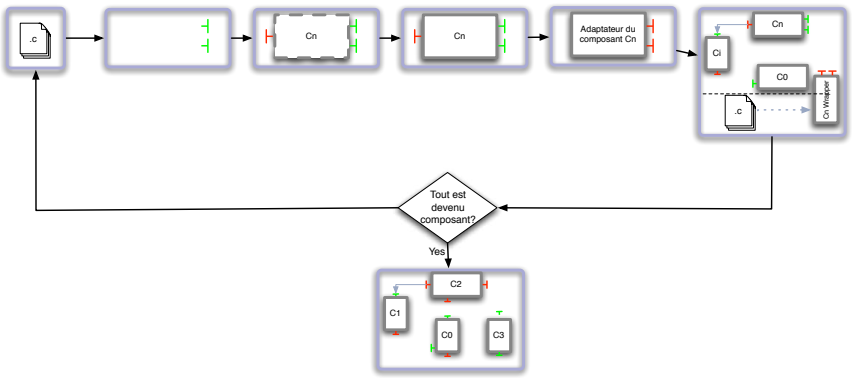


FIG. 9.1 – Flot de transformation d'une application monolithique en une version à base de composants.

Les avantages de cette méthodologie sont multiples. Premièrement, elle permet de clarifier et de simplifier le travail d'ingénierie nécessaire à la restructuration d'une application. De plus, grâce à la transformation incrémentale proposée, il est possible de compiler et tester le code en cours de transformation à la fin de chaque itération. Cela permet de détecter les erreurs introduites au fur et à mesure.

La transformation du décodeur H.264 a nécessité un homme-mois. Au cours de cette opération, le code monolithique de 26000 lignes de code a été transformé en 35 composants. La difficulté majeure fut relative à l'élimination des variables globales en les encapsulant au niveau des composants. Nous avons pu tester l'équivalence fonctionnelle de la version courante à la fin de chaque itération en utilisant des tests de non-régression. Bien que la taille moyenne de chaque composant soit d'environ 700 lignes, nous qualifions la version obtenue d'architecture à grain fin. En effet, les composants d'une application multimédia sont gourmands en nombre de lignes de code car la plupart de l'application correspond des opératiosn arithmétiques complexes manipulant de nombreux tableaux de valeurs constants.

9.1.2 Architecture du décodeur à base de composants

Le résultat de la transformation du décodeur H.264 en une version à base de composants à permis d'avoir une idée sur l'architecture initiale du logiciel. Remarquons que, nous avons tout d'abord suivi un processus de transformation sans effort particulier pour obtenir une architecture propre. Nous avons alors explicité l'architecture inhérente au décodeur monolithique. La figure 9.2 présente le schéma architectural que nous avons générée à partir de la description ADL de la version obtenue. Force est de constater que cette architecture est très compliquée et contient des composants qui sont presque tous interdépendants. Cela ne constitue évidemment pas un bon exemple d'architecture logicielle. Néanmoins, il faut considérer ce schéma d'architecture comme un exemple représentatif des logiciels monolithique tels qu'ils sont conçus couramment dans l'industrie informatique.

En partant de cette architecture, nous avons essayé de mettre en place une architecture plus simple et modulaire. La figure 9.3 reflète les blocs majeurs d'une architecture telle que l'on voudrait la concevoir dans un cadre idéal. Ce schéma identifie trois composants principaux : le composant d'entrée qui reçoit le flux au travers un moyen de communication (réseau, fichier, mémoire) et le transmet vers le composant de traitement. Ce composant décode le flux reçu en entrée et envoie le résultat à un composant de sortie. Ce dernier s'occupe d'afficher le résultat sur un écran ou de l'envoyer sur le réseau à destination d'un autre dispositif.

Chacun de ces composants contient des modules dont l'utilisation est contextuelle. Par exemple, le composant d'entrée peut être adapté en fonction du domaine d'utilisation pour contenir uniquement les composants réseau ou bien de manipulation de fichiers en fonction de la source du flux. De même, le composant de sortie peut être spécialisé pour un mode d'affichage donné. Quant au module de traitement, celui-ci peut être composé de certains blocs fonctionnels qui assurerait le décodage d'un certain type de flux. Par exemple, l'utilisation des blocs CABAC ou CAVLC pour la décompression dépend du format d'encodage du flux vidéo reçu en entrée. Le module de traitement peut être spécialisé pour n'en contenir qu'un seul si le dispositif ne reçoit qu'un de ces types de flux.

L'architecture présentée dans la figure 9.3 est obtenue en analysant le code à base de composants et la spécification de la norme H.264. Or, son implantation à partir de la version à base de composants du décodeur nous a paru trop difficile. En effet, le processus de *componentisation* présenté dans la section précédente était peu coûteux en ingénierie car il s'agissait uniquement de transformations de fichiers au niveau de la définition des interfaces, des noms de fonction, etc. Bien entendu, ce processus a uniquement permis de rendre l'architecture explicite. Contrairement à ce dernier, le processus de restructuration nécessite une réingénierie en profondeur du code d'implantation. L'effort requis pour cette réalisation nous a semblé comparable à celui qui serait nécessaire à la réécriture complète de l'application en utilisant une approche à composants. Pour ces raisons, nous avons effectué uniquement une modification légère

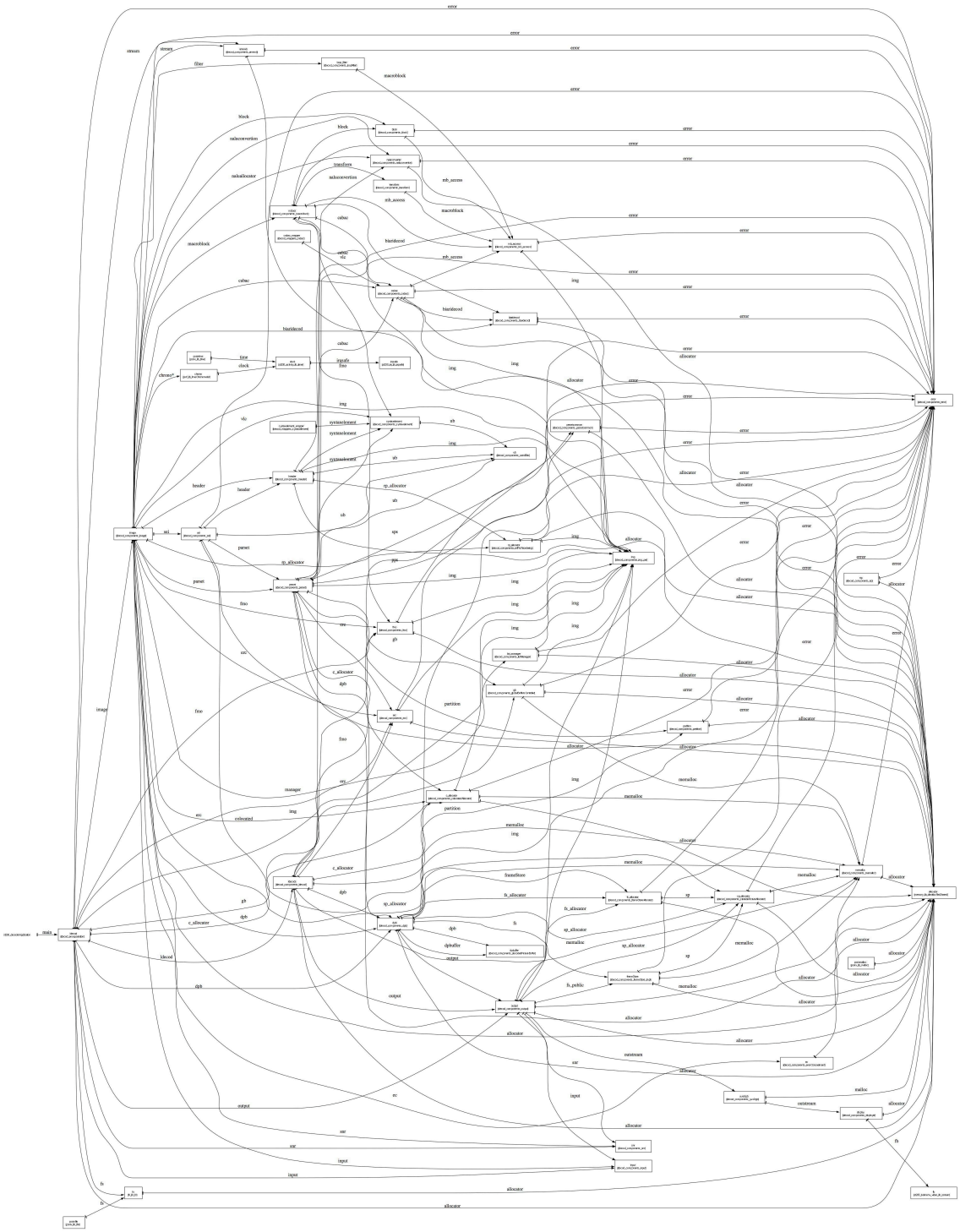


FIG. 9.2 – Organisation du décodeur H.264 monolithique.

qui consiste à définir les trois composants majeurs pour séparer les modules d'entrée, de traitement et de sortie.

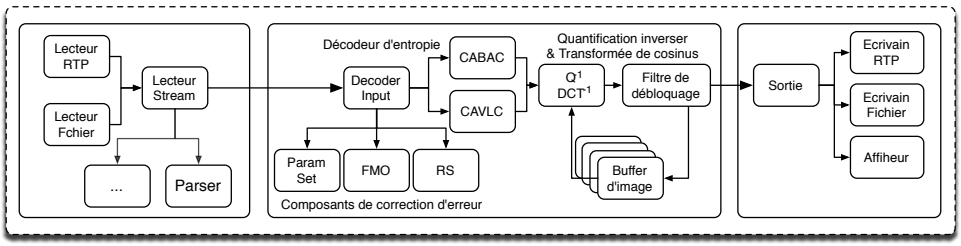


FIG. 9.3 – Architecture idéale d'un décodeur H.264 à base de composants.

9.1.3 Mise au point de différentes versions du décodeur

Comme expliqué dans la sous-section précédente, la *componentisation* du décodeur H.264 nous a permis d'identifier trois composants principaux. L'architecture de ces trois composants est vue sous la forme d'un *pipeline* où le composant d'entrée et de sortie constituent respectivement la source et le puit, et le composant de traitement est un filtre qui implante un algorithme de transformation de flux vidéo. Cette sous section est dédiée à la présentation de différentes variations de ce *pipeline* vidéo qui sont mises en œuvre pour être exécutées sur différentes architectures de plates-formes embarquées.

9.1.3.1 Version séquentielle

La première version que nous avons crée adopte un modèle d'exécution séquentiel. Cette version s'exécute d'une manière identique à celle de la version originale : le composant de sortie demande le décodage d'une image au composant de traitement, le composant de traitement récupère une image encodée auprès du composant d'entrée, effectue l'opération de décodage et envoie l'image décodé au composant de sortie. Le décodage d'un flux s'effectue par itération sur l'ensemble de ces étapes.

Le but de cette première expérimentation était en particulier de concevoir un système d'exploitation minimale qui permet d'exécuter ce décodeur H.264 sur un processeur ST230. ST230 est un processeur de type traitement de signal conçu pour être utilisé dans des solutions de type système sur puce fournis par la société STMicroelectronics. Il s'agit d'un processeur de type *very long instruction word* (VLIW) qui permet d'exploiter un parallélisme au niveau instructions afin d'augmenter l'utilisation des ressources de calcul.

Afin de mettre au point l'instance du système d'exploitation décrit ci-dessus, nous avons d'abord configuré les composants d'entrée et de sortie pour obtenir une version du décodeur qui lit le flux en entrée à partir d'un fichier et qui affiche la sortie sur un écran. Par la suite, nous avons fait en sorte d'expliciter les services système requis par cette application. Pour ce faire, nous avons localisé les appels à des services système tels que l'ouverture et la lecture de fichiers ou l'allocation de mémoire. Ces appels ont été transformés en des appels à des interfaces clientes qui identifient explicitement les services systèmes requis pour le bon fonctionnement de l'application. À la fin de cette opération, nous avons constaté que trois blocs système étaient nécessaires pour exécuter le décodeur sur une machine nue. Il s'agit d'un gestionnaire dynamique de mémoire, d'un système de fichiers et d'un pilote de dispositif d'affichage graphique.

Une fois ces modules systèmes identifiés, nous avons utilisé la bibliothèque de composants du canevas THINK avec son portage sur le processeur ST230 pour assembler un système d'exploitation qui fournit ces services. Cette opération d'assemblage a été effectuée uniquement en rédigeant des fichiers ADL. L'ensemble des fichiers ADL décrivant l'architecture du décodeur séquentiel et l'instance du système d'exploitation pour ce dernier contient 225 lignes[1]. Le noyau de système d'exploitation et l'appli-

[1]Le nombre de lignes a été compté avec le syntaxe du langage THINKADL. Ceci peut être légèrement plus si la même

cation décodeur ont alors été assemblés et compilés par le compilateur ADL. Remarquons que, grâce au niveau de modularité des composants système fournis dans la bibliothèque KORTEX, il a été possible de construire un système d'exécution *mono-thread*, ce qui permet d'exécuter l'application de nature séquentielle sans aucun surcoût lié à un mécanisme d'ordonnancement.

9.1.3.2 Version parallèle mono-processeur

Après avoir réalisé un système permettant l'exécution séquentielle du décodeur H.264, nous nous sommes intéressés à la parallélisation des composants qui constituent l'architecture en *pipeline* de cette application. Dans ce cadre, notre stratégie de parallélisation a été de découpler le flot d'exécution des trois principaux composants du *pipeline* en les exécutant avec des *threads* différents. La parallélisation de ces blocs nécessite une gestion asynchrone de leurs interactions. Comme présenté précédemment, le modèle d'interaction de ces trois composants consiste à tirer des images d'un étage inférieur. Ce modèle est fortement synchrone étant donné que l'appelant reste bloqué jusqu'à ce que l'appelé lui fournisse une image. Le découplage de ces flots d'exécution n'est possible qu'en insérant des mémoires tampons entre les points d'interaction de ces étages de pipeline. Par ce biais, on peut faire en sorte d'exécuter séparément l'étage inférieur qui remplirait la mémoire tampon avec des images pendant que l'étage supérieur consomme ces dernières à sa vitesse.

Grâce à l'approche à base de composants, nous avons pu implanter la parallélisation uniquement en agissant au niveau architectural, sans modifier l'implantation des composants fonctionnels. La figure 9.4 illustre la modification architecturale que nous avons implantée pour paralléliser les trois composants du décodeur H.264. Ces modifications peuvent être résumées en deux parties :

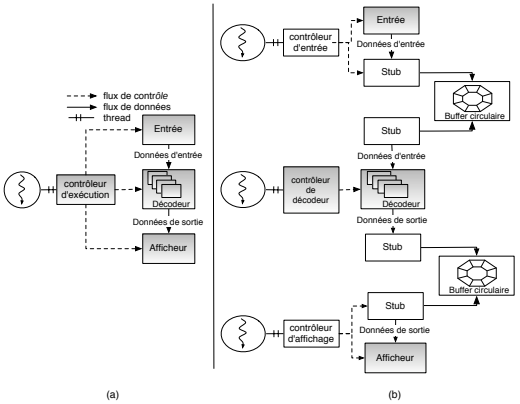


FIG. 9.4 – Modification de l'architecture du décodeur H.264 pour passer d'un mode d'exécution séquentiel à un mode d'exécution parallèle.

- Des connecteurs spéciaux ont été insérés au niveau des liaisons de ces trois composants. Ces connecteurs permettent d'insérer une mémoire tampon entre les paires de composants afin de découpler leurs interactions. Une implantation de type *buffer circulaire* a été adopté au niveau des tampons afin d'assurer les échanges de données en minimisant le nombre de copies effectuées. Des composants *stub/skeleton* ont permis d'adapter les interfaces des composants fonctionnels au composant de mémoire tampon. Par ce biais, les liaisons entre les composants ont été assurés via une mémoire tampon sans modifier leur code d'implantation.
- Des contrôleurs d'exécution ont été insérés au niveau de chaque composant parallèle. En effet, dans le modèle séquentiel, le contrôle du flot d'exécution était assuré par l'étage qui se trouvait au

architecture était écrite en FRACTALADL qui est un langage plus verbeux à cause de sa syntaxe XML.

plus haut du *pipeline*. Pour exécuter chaque étage séparément, des composants spéciaux ont été placés. Ces composants ont pour fonction d'appeler les composants fonctionnels de chaque étage pour faire en sorte que les mémoires tampons soient remplies.

Pour exécuter cette version parallèle du décodeur, un système d'exécution multi-threads est requis. Pour répondre à ce besoin, nous avons modifié le système d'exploitation initial afin d'y rajouter les mécanismes nécessaires. La figure 9.5 illustre l'architecture de l'ensemble contenant le décodeur parallèle et le système d'exploitation multi-threads. Remarquons que ce schéma est simplifié. Par exemple, le composant *scheduler* illustré en un composant est en fait constitué de plusieurs sous-composants pour la gestion des contextes du processeur, la gestion des interruptions, la gestion de politique d'ordonnancement, etc.

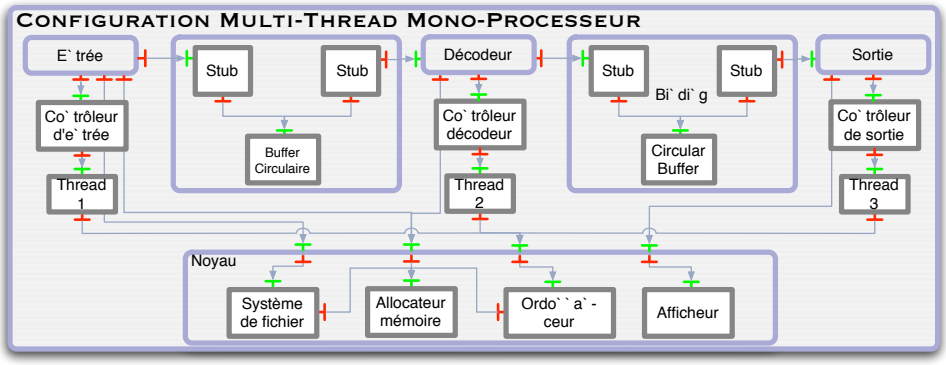


FIG. 9.5 – Architecture de l'ensemble contenant le décodeur parallèle est le système d'exploitation sous-jacent.

Comme nous disposions au préalable des composants de système d'exploitations adéquats, le passage vers cette version parallèle a nécessité uniquement l'écriture des connecteurs de mémoire et des contrôleurs d'exécution décrits ci-dessus. Le reste des modifications a été effectué au niveau des descriptions d'architecture. L'ensemble du décodeur et son système d'exploitation pour l'exécuter sur un processeur ST230 ont été décrits en 310 lignes de THINKADL. À partir de ces descriptions, le compilateur ADL nous a permis d'assembler une image binaire qui permet d'exécuter cette architecture sur la plate-forme visée.

9.1.3.3 Version parallèle multiprocesseurs

La dernière expérimentation réalisée avec le décodeur H.264 à base de composants concerne la répartition de ses trois composants principaux sur une architecture multiprocesseurs. L'architecture matérielle que nous avons considéré pour effectuer cette expérimentation est la plate-forme ST230-SMP. Cette plate-forme, destinée à la mise au point d'un système multiprocesseurs sur puce, est réifié à ce jour par un simulateur qui permet d'intégrer un nombre configurable de processeurs de type ST230 au sein d'une architecture à mémoire partagée.

La configuration parallèle de l'application ayant déjà été obtenue dans la version précédente, cette expérimentation a touché uniquement à l'adaptation de cette dernière à une architecture matérielle différente. Puisque les composants fonctionnels de l'application sont indifférents par rapport à la plate-forme d'exécution, seuls les connecteurs mémoire ont dû être adaptés. En pratique, ces composants ont été remplacés par d'autres qui implantent le même type de *tampon circulaire* mais avec des mécanismes de synchronisation de bas niveau, fournis par la plate-forme ST230-SMP.

Dans le cadre de la répartition du décodeur sur plusieurs processeurs, une autre amélioration a été effectuée au niveau du système d'exploitation. En effet, nous avons conçu des instances de système d'ex-

ploitation différentes pour chaque nœud de la plate-forme. Ces instances sont spécialisés en fonction du composant logiciel qu'elles accueillent. La figure 9.6 illustre l'architecture de cette nouvelle configuration. Nous voyons que le système d'exploitation qui accueille le composant d'entrée fournit un système de fichiers et un gestionnaire de mémoire. Le système conçu pour le composant de traitement est réduit à un simple support d'exécution (*runtime*) qui fournit des opération d'allocation dynamique de mémoire. Enfin, le système qui accueil le composant de sortie ne fournit qu'un pilote pour gérer l'affichage graphique.

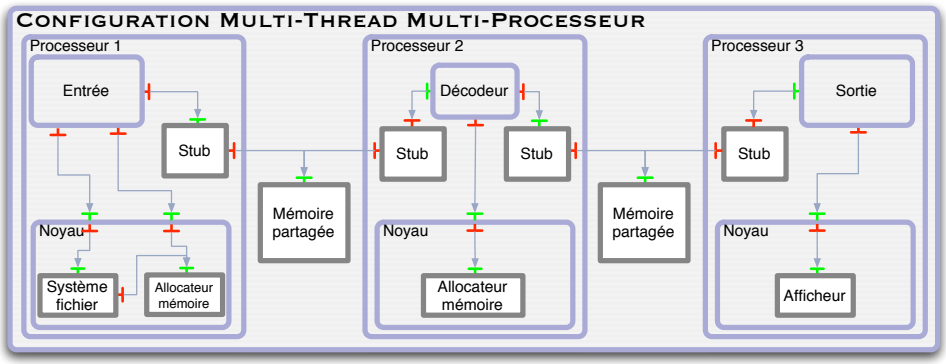


FIG. 9.6 – Architecture de la version multiprocesseurs du décodeur H.264.

9.1.4 Évaluation

Nous avons présenté les trois questions qui ont motivé notre expérimentation au début de cette section. Nous répondons maintenant à ces questions en deux volets pour distinguer les aspects qualitatifs et les aspects quantitatifs.

9.1.4.1 Évaluation qualitative

La première question concernait la viabilité de la transformation d'une application monolithique écrite en C en une application à base de composants. Nos résultats ont montré qu'il faut répondre à cette question sous deux angles. D'un point de vue faisabilité, nous avons vu qu'il était tout a fait possible d'effectuer une telle transformation. Nous avons montré qu'une méthodologie de *componentisation* a été mis en place, et a été appliqué aisément pour transformer une application de taille considérable. De plus, cette méthodologie étant incrémentale, il nous a été possible de tester l'application au fur et à mesure du processus de transformation. Nous avons ainsi obtenu une version à base de composants d'un décodeur multimédia avec un effort tout à fait acceptable. Cependant, d'un point de vue architectural, le processus de transformation n'a pas amélioré l'architecture inhérente à l'application monolithique. En effet, nous avons vu qu'en suivant une stratégie de *componentisation* qui consiste à encapsuler dans des composants les fichiers (qui sont les unités de modularité du langage C) nous avons obtenu une structure très complexe. De plus,il nous a semblé très difficile d'en dériver une architecture propre. Nous avons conclu qu'une procédure de développement à base de composants devrait être appliquée dés la conception d'un projet de développement pour obtenir une architecture propre et modulaire.

La deuxième question qualitative qui nous préoccupait concerne les avantages d'un processus de développement basé sur l'architecture logicielle pour les applications multimédia, et en particulier à destination des plates-formes multiprocesseurs sur puce. Nous avons vu au travers nos cas d'études qu'une fois que l'architecture d'un logiciel est rendue modulaire, il était possible d'en dériver plusieurs versions différentes en effectuant des manipulations au niveau de l'ADL. Nous avons en particulier montré

comment le *pipeline* du décodeur pouvait être adapté à des modèles d'exécution variés, sans effectuer la moindre modification au niveau des composants fonctionnels. En effet, la présence d'un compilateur ADL qui assemble une configuration à base de composants à partir des description d'architecture nous a permis d'assembler des couples de décodeurs et de systèmes d'exploitation spécialisés avec un effort minimal. Néanmoins, remarquons que notre expérimentation concernait uniquement la parallélisation de composants à gros grain avec des interactions faiblement couplées. La parallélisation sans modification de code fonctionnel des composants à grain plus fin nous a semblé impossible car cela nécessiterait l'emploi de protocoles d'interaction complexes qui seront introduits avec le parallélisme.

9.1.4.2 Evaluation quantitative

La dernière interrogation ayant motivé cette expérimentation était quantitative. Il s'agit d'évaluer l'impact de la *componentisation* en termes de performances. Nous préférons encore répondre à cette question sous deux angles.

Le premier aspect est le coût de la technologie à composants utilisée. Dans le cadre de nos expérimentations, nous avons utilisé le canevas THINK. Afin d'évaluer le coût de cette technologie, nous avons comparé le temps d'exécution et la taille en mémoire de la version originale de l'application à la version à base de composants s'exécutant en mode séquentiel. Nous avons choisi d'effectuer cette comparaison sur plate-forme matérielle standard. Nous avons alors considéré l'architecture Intel x86 associée à un noyau Linux 2.6. Nous avons exécuté le décodeur assez longtemps (pour 500 images qcif) afin de relativiser les impactes non déterministes de type cache. Nous avons observé un surcoût de 1.5% en ce qui concerne le temps d'exécution du décodeur à base de composants par rapport à la version monolithique. Ce coût est principalement dû aux indirections mémoire qui sont effectuées au niveau des appels aux interfaces de composants. Ce surcoût nous semble tout à fait acceptable étant donné qu'il s'agit d'une version très modulaire, constituée de 35 composants. De plus, il est tout à fait possible d'améliorer ces chiffres en intégrant certaines stratégies d'optimisation au niveau du compilateur ADL utilisé pour la génération de code. Quant au surcoût en taille mémoire, il est de 7%. Cela est principalement dû au caractère réflexif des composant THINK. Il est tout à fait envisageable d'améliorer l'implantation actuelle du compilateur ADL pour éliminer les informations tissées au niveau de chaque composant lorsque la réflexivité n'est pas utilisée par l'application.

Le deuxième aspect est le coût impliqué par la modification de l'architecture du logiciel. Dans le cadre de nos expérimentations, nous n'avons pas effectué de telles modifications au niveau du décodeur. Notre sentiment est qu'il est difficile de déterminer a priori un éventuel gain ou surcoût lié à l'architecture. Bien qu'en général une architecture modulaire est moins performante qu'une architecture monolithique, la modularité peut servir pour la mise au point des versions spécialisées d'une application et peut s'avérer bénéfique au niveau des performances. Nous avons par exemple vu que la modularité fournie par la bibliothèque KORTEX nous a permis de construire des instances de systèmes d'exploitation adaptées aux fonctions de l'application accueillie.

9.2 Comment aller plus loin ?

L'expérimentation sur la mise en œuvre du décodeur H.264 nous a permis d'identifier un certain nombre de perspectives pour améliorer la programmation des applications de *streaming* à destination de systèmes multiprocesseurs sur puce. Nous essayons, dans cette section, de décrire comment le processus de développement de telles application pourrait être amélioré. Nous commençons par préciser nos objectifs. Ensuite, nous dressons un état de l'art pour mieux connaître les différentes approches proposées dans le passé pour la programmation des applications de *streaming*. Nous présentons enfin notre proposition qui vise à améliorer notre compilateur ADL afin de mettre en place un modèle de programmation et un support d'exécution pour développer des composants aisément parallélisables.

9.2.1 Problématique

Les expérimentations présentées plus haut nous ont permis d'identifier les problèmes suivants en ce qui concerne la programmation des applications multimédia à destination de systèmes multiprocesseurs sur puce.

- La mise en œuvre des applications de streaming est un processus non trivial. D'une part, ce type d'applications contient de l'algorithmique complexe, de l'autre leur déploiement efficace sur des systèmes embarqués nécessite une maîtrise à la fois du logiciel et de la plate-forme matérielle. À cela se rajoutent les problèmes de développement dans des équipes géographiquement réparties, la maintenance et l'évolution du logiciel. Pour aborder ces problèmes, les considérations liées à l'architecture logicielle constituent la clef pour la définition d'une méthodologie de développement robuste.
- Le nombre de processeurs intégrés aux systèmes sur puce augmentant, la répartition des différentes applications ne suffit plus pour exploiter les ressources matérielles de manière efficace. Des taux de parallélisme bien supérieurs à ceux obtenus actuellement sont requis. Dans ce contexte, il y a un réel besoin pour la définition d'un processus de développement qui permet une répartition à grain fin des composants d'une application et la reconfiguration dynamique de ces derniers pour assurer la régulation de charge.
- Enfin, la notion de parallélisme est fortement liée aux modèles d'exécution et d'échange de données. Or, il n'existe pas de modèle universellement optimal. Dans ce contexte, nous avons l'intuition que le modèle de programmation adoptée devra permettre l'exploration de différentes possibilités. Comme mentionné dans [Lee06], il faudra rechercher un modèle de programmation qui rend l'implantation des composants aussi indépendants que possible des contraintes environnementales tout en acceptant la programmation dans des langages généralistes tels que C.

9.2.2 Approches existantes pour la programmation des applications de streaming

Nous nous proposons dans cette section de passer en revue, de manière très synthétique, un ensemble de langages proposés pour la programmation d'applications de type *streaming* ou d'applications parallèles en général. Nous débutons la section en fixant des axes de comparaison pour analyser l'état de l'art sous les angles que nous nous sommes fixés plus haut dans le cadre de nos objectifs. Ensuite, nous analyserons un certain nombre de propositions académiques ou industrielles en essayant de les classifier en quatre volets différents. Enfin, nous dressons une synthèse de l'ensemble des approches analysées.

9.2.2.1 Axes de comparaison

Nous étudierons par la suite un certain nombre de propositions de langages pour la programmation des applications de *streaming* parallèle. Nous axes de comparaison pour analyser ces travaux seront les suivants :

- Le premier axe de comparaison, et certainement le plus important, est le **modèle de concurrence**. Il s'agit alors de la vision du parallélisme qui est fournie au programmeurs par le langage de programmation et de la manière dont les tâches parallèles sont exécutées.
- Les tâches parallèles nécessitent une coordination. Le deuxième axe de comparaison concerne le **modèle de synchronisation**. Remarquons que tous les modèles de programmation ne nécessitent pas forcément l'expression explicite de la synchronisation mais intègrent néanmoins un modèle de synchronisation.
- Un autre axe de comparaison concerne les **structures de données** qui transitent entre les composants d'une application. Ceci est un élément crucial car les flux multimédia tels que de la vidéo ou de l'audio sont constitués de données structurées.
- Les applications visées devenant de plus en plus complexes, les considérations d' **architecture**

logicielle deviennent cruciales dans un contexte de développement industriel. Le méthodologie de développement approprié, la modularité, et la possibilité de réutilisation des blocs logiciels présentent un impact certain sur la productivité. Par conséquent, cela fait partie des éléments que nous considérerons au cours de notre analyse.

Enfin, les performance du logiciel produit constituent un des éléments crucial à prendre en considération pour évaluer un langage de programmation. Cependant, nous n'incluons pas cette dimensions dans nos axes de comparaison. Ceci est premièrement dû au fait qu'il nous était difficile de concevoir une base d'application équitable qui permettrait de comparer les différents langages étudiés. De plus, notre analyse incluant des travaux académiques et industriels, il nous a paru impossible de comparer des produits commerciaux à des prototypes de recherche parfois non disponibles. Nous avons néanmoins veillé à nous assurer de la viabilité (en termes de performances) des solutions présentées ci-après.

9.2.2.2 Programmation à base de threads

La programmation à base de *threads* constitue, sans conteste et depuis longtemps, la méthode la plus fréquente pour le développement des applications parallèles. Dans la programmation à base de *threads*, les programmeurs ont une visibilité directe des ressources de calcul qui sont virtualisées par des *threads*. L'association des segments de code à exécuter sur les ressources de calcul séparées, c'est-à-dire le modèle de concurrence de l'application, est directement exposé au programmeurs. Par conséquent, cela fait partie des choix de conception et d'implantation à prendre en considération. L'exécution d'un programme à base de *threads* nécessite un support à l'exécution, généralement assuré par le système d'exploitation, pour l'ordonnancement dynamique de ces ressources de calcul virtuelles afin de les coupler aux ressources physiques disponibles.

En ce qui concerne la synchronisation des tâches parallèles, seuls des mécanismes de synchronisation de bas niveau (de type sémaphores, exclusions mutuelles, moniteurs ou barrières) sont supportés. En effet, comme la programmation à base de *threads* fournit uniquement une vision bas niveau du parallélisme au niveau de ressources de calcul et ne fournit pas d'abstraction pour les échanges de données, elle ne prend pas en considération les dépendances fonctionnelles des modules parallèles. Ceci fait que les programmeurs doivent assurer explicitement la synchronisation de ces derniers en utilisant les moyens mentionnés précédemment.

La programmation à base de *threads* peut être supportée sous deux formes : (1) les bibliothèques de type POSIX ou UNIX pour les langages C/C++ ou bien les *Boost Threads* pour C++ fournissent un support pour les *threads* qui s'ajoute orthogonalement au langage de programmation ; (2) les langages de programmation comme Java fournissent des primitives langage natives pour la manipulation des *threads*. Ce type de programmation correspond bien, de manière générale, à la programmation des applications parallèles destinées à être exécutées sur des plates-formes multiprocesseurs à mémoire partagée (SMP pour *Symmetric Multi-Processing*). En effet, dans la plupart des infrastructures de programmation, les *threads* partagent un même espace mémoire. Par conséquent, l'échange de données entre les tâches parallèles est assuré par le partage, et leur accès est ordonné par le biais des mécanismes de synchronisation. Dans le cas des architectures matérielles à mémoire distribuée, il convient d'utiliser des interfaces d'échange de données de type MPI [Mes97].

Quiconque a déjà développé un programme parallèle à base de *threads* peut attester que la maîtrise du logiciel est extrêmement difficile. En effet, comme la gestion de la concurrence et de la synchronisation des ressources de calculs parallèles est à la charge du programmeur, la complexité du programme grossit de manière exponentielle en fonction du niveau de parallélisme. De plus, un des problèmes majeurs de ce type de programmation tient au non-déterminisme à l'exécution [Lee06], ce qui rend encore plus difficile la maîtrise du comportement des éléments parallèles. Le fait que l'ordonnancement soit effectué par un mécanisme externe de manière dynamique rend fastidieuse voire impossible la reproduction des

comportements anormaux, ce qui rend le débogage de tels programmes particulièrement difficile.

Malgré ces éléments négatifs, la programmation à base de *threads* est couramment employée, en particulier dans le domaine industriel. Nous voyons deux raisons à cet état de fait. Premièrement, le fait que les *threads* soient proposés comme des extensions de langages connus, évolués et largement acceptés simplifie son adoption au sein des projets de développement qui ont des contraintes temporelles très serrées. Ensuite, la programmation à base de *threads* est rentre bien dans la culture de la programmation traditionnelle qui consiste en la mise au point d'une version séquentielle du programme avant de se poser la question de son parallélisme. Paradoxalement, la prise en compte tardive de cet aspect fait que le parallélisme obtenu est souvent très limité.

9.2.2.3 Programmation avec des langages synchrone

A l'opposé de la programmation à base de *threads*, qui adopte un modèle d'exécution totalement asynchrone se situe l'approche de programmation synchrone. Les langages de programmation synchrones prennent la concurrence comme étant une considération majeure et offrent une exécution déterministe des programmes parallèles. En effet, dans la plupart des langages de programmation synchrones, chaque instruction est considérée comme parallèle, c'est-à-dire sans relation de causalité. Un programme est constitué d'une suite d'opérations arithmétiques de type $f : (x_0...x_m) \rightarrow (y_0...y_n)$ contenant des accès mémoires. Les opérations parallèles sont alors exécutées au sein d'un même intervalle de temps logique. Seules les opérations d'accès mémoire qui exposent une relation de causalité nécessitent des opérations de synchronisation pouvant avoir comme effet d'avancer le temps logique de l'application.

L'ensemble des instructions est globalement compilé vers une machine à états déterministe qui fournit un modèle exécutable du programme. Cette machine-à-états abstraite peut être ensuite compilée vers des formes d'exécution différentes commençant par des circuits électroniques (de type FPGA ou ASIC) où l'on peut exploiter le parallélisme à un niveau choisi, allant jusqu'à des programmes séquentiels exécutables sur processeur. Au niveau de la compilation vers des programmes parallèles, il existe certains travaux comme SL [BdS96], qui permettent de répartir les sous-graphes de la machine à états sur différents nœuds de calcul physiques.

Les trois propositions principales dans le domaine de la programmation synchrone sont Lustre, Esterel et Signal dont [BCE+03] donne une introduction. Esterel, grâce à sa syntaxe impérative, constitue probablement le langage le plus approprié pour la programmation des applications de *streaming* parallèles. Les constructions de ce langage permettent des programmer des tâches séquentielles s'exécutant en parallèle et donnent des moyens de synchronisation et de communication sous formes de signaux pour établir des interactions entre ces dernières. Or, les structures de données supportées par ces langages restent assez simplistes, ce qui restreint la description des données fortement structurées que l'on trouve dans les flux multimédia.

À ce jour, les langages de programmation synchrones sont principalement employés pour le développement des applications temps-réel critiques. En effet, la propriété d'exécution déterministe des programmes écrits dans ces langages permettent d'effectuer des vérifications très poussées grâces aux preuves formelles que l'ont peut appliquer au modèle. Quant à la programmation des applications multimédia, nous ne connaissons pas de projet industriel de développement de logiciel qui emploie ce type d'approches. Les restrictions sur les structures de données supportées constituent certainement un élément qui explique en partie cette situation. De plus, l'obligation de compilation globale semble être contraignanet pour le développement de logiciels de taille importante. A ces aspects techniques se rajoutent sûrement des considérations culturelles de la communauté des développeurs de logiciel multimédia qui sont plus proche de la programmation séquentielle et qui préfèrent par conséquent les technologies de programmation à base de *threads*.

9.2.2.4 Langages pour la programmation parallèle

Certains langages sont spécifiquement conçus pour permettre aux programmeurs d'exprimer le parallélisme explicitement. Nous étudions dans cette sous section deux d'entre eux, à savoir Cilk et Athapascan.

Cilk [Ran98] est une proposition académique qui présente les principaux traits d'un langage conçu pour la programmation des applications parallèles. Il étend le langage C avec deux constructions permettant de lancer des opérations qui s'exécutent de manière parallèle et de synchroniser ces dernières. Les primitives de synchronisation sont alors d'un niveau d'abstraction plus élevé que celles fournies par les bibliothèques conçues pour la programmation à base de *threads*.

Au titre d'exemple, considérons le segment de code suivant :

```
spawn f1() ; spawn f2() ; sync ;
```

Ce morceau de code lance deux tâches parallèles pour exécuter les fonctions `f1` et `f2`, et puis se met en attente de leur terminaison pour avancer. De cette manière, la complexité de la création et de la configuration des *threads* chargés d'exécuter ces deux fonctions ainsi que de la mise en place d'une barrière pour capturer leur terminaison est cachée au programmeur. Ceci permet d'une part de simplifier l'écriture d'un programme parallèle, et, de l'autre, d'effectuer un certain nombre d'optimisations. La gestion d'un lot (*pool*) de *threads* qui permet de réutiliser des *threads* inactifs au lieu d'en initialiser un pour chaque opération parallèle constitue un exemple typique d'optimisation que le compilateur Cilk peut mettre en œuvre. Cilk fournit en plus, une bibliothèque de primitives pour créer des exclusions mutuelles sur les variables partagées entre plusieurs fonctions. Cette librairie est renforcée par un outil qui permet de détecter des conflits d'accès aux variables partagées pour éliminer les problèmes liés au non-déterminisme du modèle d'exécution sous-jacent.

Athapascan [GRCD98] propose un syntaxe proche de Cilk permettant de lancer des opération parallèles. Cependant, il renforce la synchronisation de ces derniers par le biais du typage des accès à la mémoire. Ceci permet d'écrire des programmes où la concurrence d'accès aux variables partagées est implicitement gérée par l'environnement d'exécution. En effet, le système d'exécution fourni avec Athapascan construit à la volée un graphe de dépendances entre les tâches parallèles et utilise ce dernier pour établir un ordonnancement correct. À l'instar de Cilk, Athapascan est compatible avec les structures de données du langage C, ce qui permet de mettre en œuvre des flux de données structurés.

En résumé, les langages pour la programmation parallèle exposent la gestion des ressources de calcul parallèles au niveau du langage de programmation, mais simplifient grandement la manipulation de ces derniers par comparaison aux langages de programmation classiques. Ils offrent ainsi une approche de programmation asynchrone, au même titre que les *threads*, renforcés par des macro-opérations pour la gestion du parallélisme. La plupart des langages qui rentrent dans ce cadre sont issus de travaux académiques, et ne sont utilisés dans le milieu industriel.

9.2.2.5 Langages spécifiques à la programmation des applications de streaming

Les langages de programmation généralistes, présentés plus haut, ne fournissent pas d'abstractions appropriées pour la représentation des flots de données continus. Ainsi, les compilateurs ne sont pas en mesure de réaliser des optimisations spécifiques sur ce type de flots. Afin d'aborder ce problème, un certain nombre de travaux optent pour la définition d'un langage spécifique (DSL pour *Domain Specific Language*) à la programmation des applications de *streaming*. Dans la suite, nous en étudions deux, StreamIt [TKA02] et Spidle [CHR+03], qui nous semblent constituer les propositions les plus significatives dans ce domaine.

StreamIt est une extension du langage Java. Il enrichit ce langage avec un certain nombre de constructions (i.e. filtres, bifurcation, jointure, etc.) qui permettent de décrire des modules d'une application de *streaming* en spécifiant leurs entrées et leurs sorties. Une application peut donc être composée d'un ensemble de modules organisés de manière hiérarchique. Les modules de StreamIt s'exécutent et échangent des données en suivant un modèle synchrone. C'est-à-dire qu'à un moment donné, un ensemble de modules qui sont activés en raison des événements présents dans leurs canaux d'entrée s'exécutent en parallèle au sein d'un même intervalle de temps logique. Les événements qui sont produits en résultat de leur exécution activent un autre ensemble de modules, et l'exécution se poursuit ainsi de suite. Les débits de communication des modules étant spécifiés par le programmeur, le compilateur est à même de générer un ordonnancement statique, sous forme d'une machine à états finis. Au cours de la compilation, les caractéristiques du flux de données et celles des modules sont utilisées pour effectuer des optimisations spécifiques. L'ordonnancement en fonction des échanges de données pour limiter les défauts de caches, la fusion des modules adjacents et l'optimisation des mémoires tampons insérés entre les modules font partie de ce type d'optimisations. StreamIt permet aussi de distribuer les calculs sur un système réparti. En effet, comme expliqué dans [TKA02], une fois la machine à états générée, elle peut être découpée en sous-graphes afin de répartir l'exécution de l'application sur plusieurs nœuds de calcul physiques.

Une autre propriété intéressante de StreamIt concerne la prise en compte des flux de contrôle. Le mécanisme appelé *teleport messaging*, décrit en détails dans [TKS+05], permet de transmettre des messages de contrôle entre les modules, adjacents ou non, dans le sens du flux de données aussi bien que dans le sens inverse. Ce mécanisme se base entièrement sur le modèle d'exécution synchrone adopté par StreamIt. En effet, la gestion d'un tel mécanisme avec les mêmes propriétés semble très complexe et très coûteux à mettre en œuvre dans un modèle d'exécution asynchrone. En ce qui concerne les performances de StreamIt, très peux de résultats sont publiés. Le fait que le langage soit basé sur Java laisse penser que les performances ne peuvent pas être meilleures que celle d'un programme Java classique, qui sont elles-mêmes moindres que celle d'un programme écrit en C. De plus, le typage faible des données passées en paramètre aux modules constitue une des faiblesses de l'approche.

Spidle, dont la conception a débuté presque simultanément à StreamIt, est très proche dans ces objectifs mais tend plutôt vers un langage déclaratif pour simplifier la programmation des applications de *streaming*. Les modules, tels que spécifiés dans Spidle, autorisent un découpage d'une application en filtres réutilisables avec une interface fortement typée. Comme StreamIt, Spidle autorise la composition hiérarchique des modules et permet de compiler une application vers un ordonnancement statique. De plus, grâce à son expressivité plus forte, Spidle permet d'effectuer des vérifications architecturales sur des compositions de modules. En ce qui concerne les performances, [CHR+03] démontre qu'un programme en Spidle résulte dans le pire des cas à un baisse négligeable par rapport à du code optimisé à la main. Ceci montre que des gains en conception peuvent être obtenus sans dégradation significative des performances à l'aide du langage Spidle.

9.2.2.6 Langages de programmation orientées interactions

Certains langages de programmation se concentrent sur la maîtrise des interactions entre les modules logiciels constituant une application. Un certain nombre d'entre eux sont spécifiquement conçus pour la mise en œuvre des applications de *streaming*. Ces langages fournissent en général des constructions de type événement-condition-action (ECA) pour permettre la modélisation des protocoles d'interaction de chaque module. Plus précisément, les constructions de type ECA permettent d'identifier une collection d'événements qui ont pour effet de déclencher l'exécution de certaines actions. De cette manière, le modèle de synchronisation des modules peut être identifié implicitement sous forme d'événements. Des compilateurs ou des supports d'exécution peuvent alors être utilisés pour exploiter le parallélisme qui découle des dépendances de ces modules en fonction des caractéristiques de l'environnement d'exécu-

tion.

Le Join-Calcul [FG96] constitue une base mathématique largement répandue pour concevoir ce genre de constructions. Dans ce cadre, les langages JoinJava [IJ03] et Polyphonic C# [BCF04] fournissent respectivement des extensions des langages Java et C# pour supporter ce type d'abstractions.

Polyphonic C# adopte l'expression des conditions de jointure au niveau de chaque fonction. En effet, l'entête de déclaration d'une méthode est exprimé sous la forme d'un accord (*chord*) qui correspond à un patron (*pattern*) de synchronisation. Un *chord* peut être composé de plusieurs opérations de type *async*. Le mot clé *async* permet d'identifier une opération dont l'exécution peut être assurée de manière asynchrone. Les opérations de type *async* ne sont pas compatibles avec une valeur de retour et sont par conséquent obligatoirement de type *void*. Seules les opérations synchrones peuvent avoir des valeurs de retour. Ainsi, un *chord* peut contenir une seule opération synchrone.

Le segment de code ci-dessous illustre la programmation d'un tampon (*buffer*) en Polyphonic C#.

```
public string Get() & public async Put(string s) {
  return s ;
}
```

La sémantique associé au *chord* décrit dans l'entête de l'opération est la suivante : le corps de la fonction ne sera exécuté que lorsque toutes les opérations constituant le *chord*, en l'occurrence `Get` et `Put`, auront été appelées. La valeur de la chaîne de caractèrse `s` obtenu par l'opération `Put` sera alors retournée.

Le compilateur de Polyphonic C# est implanté sous forme d'un pré-processeur qui transforme les programmes écrits dans ce langage en des programmes standards en C#. Les *chords* sont alors remplacés par des machines à états qui gèrent la jointure des événements et exécutent l'opération associée lorsque tous ces événements sont présents. Les mécanismes de synchronisation et de création de *threads* sont alors complètement cachés aux programmeurs. C'est au support d'exécution qu'il revient d'associer dynamiquement les opérations à exécuter sur un lot de *threads* afin d'optimiser les performances. De nombreux exemples sont donnés dans [BCF04] montrant ainsi que l'on peut coder des problèmes standards de la programmation parallèle à l'aide de ce modèle de façon concise et élégante. De plus, les performances de Polyphonic C# sont globalement comparables à du code optimisé à la main en C# pour une exécution sur une machine comportant de multiples *threads* physiques.

JoinJava propose globalement les mêmes propriétés que Polyphonic C# pour la programmation en Java. Une contribution de JoinJava est la possibilité de rajouter des éléments de priorité au niveau des *chords* pour avoir plus de contrôle sur la manière dont les opérations seront exécutées.

Les propositions présentées ci-dessus sont uniquement issues des projets de recherche et n'ont, à notre connaissance, pas été utilisées dans un cadre industriel. Néanmoins, nous pensons que ces travaux se rapprochent d'un juste compromis entre l'utilisation des langages de programmation classiques et l'exploitation des abstractions appropriées pour la minimisation des problèmes liées à la mise en œuvre des applications. De plus, nous les constructions de type ECA constituent, à notre avis, une manière très élégante pour la description de comportement des composants des applications de *streaming* étant donné qu'elles permettent de spécifier les éléments d'entrée dont la présence déclenche une réaction au sein d'un composant.

9.2.2.7 Synthèse

Le tableau 9.1 récapitule notre analyse de différents travaux dans une perspective de programmation des applications de streaming parallèles. Nous y voyons un certain contraste entre le modèle de synchrone et le modèle de programmation à base de *threads* qui fournit une exécution asynchrone. Alors

que le premier fournit un modèle approprié pour l'expression de la concurrence, il reste faible au niveau des structures de données, et de l'architecture logicielle qui sont deux éléments primordiaux dans la programmation des application de type multimédia.

	modèle de concurrence	modèle de synchronisation	structures de données	architecture logicielle
Langages *threadés*	*	*	****	****
Langages synchrones	****	****	*	*
Langages de streaming	***	***	**	***
Langages parallèles	***	***	****	**
Join-Calcul	***	****	****	****

TAB. 9.1 – Bilan de l'analyse des langages utilisés pour la programmation des applications de it streaming.

Les langages spécifiques à la programmation des applications de *streaming* semblent proposer de bons compromis entre l'expression de la concurrence et la structuration du code à l'aide des constructions appropriées. Néanmoins, ces propositions souffrent des inconvénients de la compilation statique qui limite en particulier les possibilités de reconfiguration dynamique. De plus, le modèle d'exécution synchrone impose des contraintes au niveau de la répartition des modules sur un système multiprocesseurs. Les langages parallèles fournissent des abstractions appropriées pour l'exécution parallèle mais ne proposent pas d'éléments architecturaux pour la construction des applications de *streaming*. Enfin, les langages basé sur le join-calcul proposent un modèle de synchronisation qui permet de maîtriser l'exécution parallèle des composants dans un environnement d'exécution asynchrone. Cette propriété nous semble très pertinente pour la conception d'applications réparties dynamiquement reconfigurables.

9.2.3 Objectifs

Après avoir analysé les travaux décrits ci-dessus, nous proposons de mettre au point un processus de développement pour la programmation des applications de *streaming* multimédia à destination des systèmes sur puce. Nos objectifs sont les suivants :

- opter pour la programmation à base de composants pour répondre aux besoins en matière d'architecture logicielle,
- développer un langage de description pour la définition des protocoles d'interaction entre composants afin d'extraire de l'implantation les aspects liés au modèle d'échange de données et d'exécution,
- renforcer le processus de développement basé sur l'architecture avec le langage mentionné au point précédent afin d'aborder les considérations de type parallélisation, répartition et reconfiguration dynamique,
- enfin, supporter par le biais de la chaîne d'outils ADL la possibilité de cibler différents modèles d'exécution et d'échange de données pour adapter efficacement une application à une plate-forme matérielle donnée.

9.3 THINKJoin : Un modèle de programmation à base de composants pour applications de streaming

Nous présentons dans cette section, une extension au modèle FRACTAL que nous avons mise au point afin d'améliorer notre processus de développement à base de composants en lumière des objectifs présentés dans la section précédente. Nous commençons par présenter une vue globale de notre proposi-

tion. Ensuite, nous présentons le langage JoinDSL qui permet de décrire les protocoles de collaboration des composants. Ensuite, nous décrivons comment le modèle FRACTAL et son compilateur ADL ont été étendus pour prendre en compte les descriptions JoinDSL. Enfin, nous présentons les différents types de modèles d'exécution qui sont compatibles avec la programmation des composants que nous proposons.

9.3.1 Vue d'ensemble

THINKJoin est une extension du modèle de composants FRACTAL ainsi que de son compilateur ADL qui a pour but d'améliorer le processus de développement des applications de *streaming* parallèles. La logique de cette extension est la suivante : chaque composant est conceptuellement considéré comme une unité de logiciel autonome, s'exécutant potentiellement en parallèle avec d'autres modules se trouvant dans son environnement. L'extension apportée vise à maîtriser la collaboration de tels composants autonomes au sein d'une application arbitrairement complexe.

Dans ce cadre, un nouveau langage est proposé aux architectes du logiciel pour leur permettre de décrire le comportement de chaque composant sous forme de patrons (*patterns*) spécifiant les conditions qui déclenchent des réactions du composant. Ces descriptions sont exploitées par le compilateur ADL pour générer le code de contrôle de l'exécution approprié pour chaque composant sous forme de machines à états. Ainsi, le contrôle de l'exécution de chaque composant, autrement dit la synchronisation des composants avec leur environnement est isolée (gérée à l'extérieur) du code d'implantation des composants. Les avantages d'une telle démarche sont doubles. Premièrement, la tâche des programmeurs est simplifiée étant donné qu'ils peuvent se concentrer uniquement sur les aspects fonctionnels en s'abstrayant totalement des contraintes architecturales. Deuxièmement, les composants peuvent êtres adaptés à des environnements différents car des machines à états variées peuvent être générées par le compilateur ADL en fonction du modèle d'exécution ciblé. Ainsi, comme les composants deviennent adaptables à des contextes d'utilisation différents, leur réutilisabilité est naturellement améliorée.


```
public interface Mixeur {
  void mix(image e1, image e2);
}
public interface FluxImage {
  void push(image i) ;
}

<definition name="mixeur">
 <interface name="entree" signature="Mixeur" role="server"/>
 <interface name="sortie" signature="FluxImage"
            role="client"/>
 <implementation class="mixeur" language="C"/>
</definition>
```

FIG. 9.7 – Architecture d'un composant mixeur.

Afin de mieux comprendre l'utilité de cette aspect d'adaptation, nous nous proposons d'examiner un cas d'étude. Considérons le cas d'un composant mixeur d'images qui reçoit deux éléments en entrée, e_1 et e_2, qu'il doit additionner pour produire son résultat. La figure 9.7 illustre l'architecture d'un tel composant ainsi que les descriptions ADL et IDL qui la décrivent. Écrit de cette manière, ce composant est une unité logicielle passive qui est exécutée lorsque son interface d'entrée est appelée par un autre composant.

La figure 9.8 représente une architecture de composition classique d'application de *streaming* où le composant *mixeur* est déployé. Il s'agit d'une application où le contrôle d'exécution est géré par une unité centrale : le composant *contrôleur* appelle en boucle les deux composants décodeurs pour obtenir les images e_1 et e_2 qui sont par la suite envoyées au composant mixeur pour obtenir l'image mixée s.

Ce type d'architecture correspond bien à une exécution séquentielle. Or, la parallélisation du traitement effectué par les blocs de décodage et de mixage nécessite la modification du composant de contrôle.

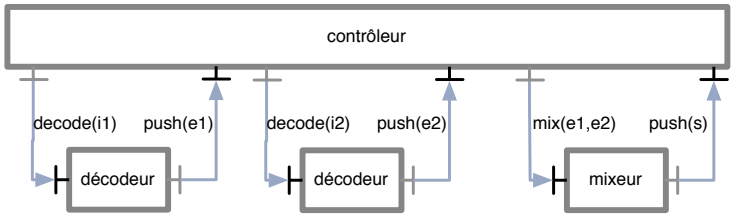


FIG. 9.8 – Architecture d'une application classique de *streaming* avec un contrôleur d'exécution central.

La figure 9.9 illustre le principe d'une nouvelle architecture de contrôle, mieux adaptée à l'exécution parallèle des composants de traitement. Les deux décodeurs s'exécutent de manière parallèle pour transformer les éléments i_1 et i_2 en e_1 et e_2 et envoient ces dernier à destination du composant de mixage en concurrence. Un composant *contrôleur* est alors requis pour constituer un tampon entre la réception des éléments e_1 et e_2 et l'opération de mixage implanté par le composant mixeur. Plus précisément, ce composant a le rôle de joindre des paires de e_1,e_2 pour appeler l'opération `mix` en les donnant en paramètre. Ainsi, le composant *mixeur* présenté dans la figure 9.7 peut être réutilisé sans modification avec un tel modèle d'exécution parallèle.

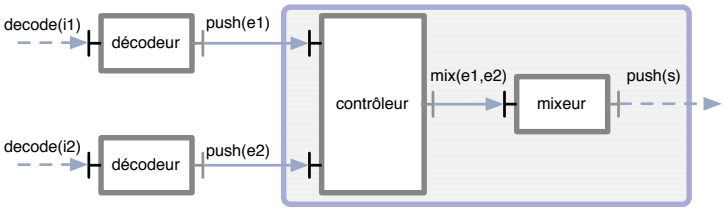


FIG. 9.9 – Architecture de contrôle répartie pour l'exécution parallèle des composants de décodage et de mixage.

Le rôle des descriptions JoindDSL est précisément de décrire le comportement de tels composants qui sont en charge de contrôler l'exécution des composants de traitement pour les générer automatiquement en fonction du modèle d'exécution adopté. De cette manière, les échanges de données et l'exécution des composants de traitement peuvent être automatiquement adaptés à des environnements d'exécution différents pour mieux exploiter les capacités de l'architecture de la plate-forme matérielle sous-jacente. Nous présentons par la suite le langage JoinDSL pour la description des comportements ainsi que les caractéristiques du compilateur ADL qui permet de générer les composants de contrôle d'exécution requis.

9.3.2 Le langage JoinDSL

Le langage JoinDSL est basé sur le Join-Calcul et contient un certain nombre d'enrichissements qui ont pour but de simplifier la programmation des composants. Une description complète de Join-Calcul est présentée dans [FG96]. Nous nous proposons par la suite de donner une introduction informelle au langage JoinDSL en identifiant ses différences par rapport au Join-Calcul standard. Ensuite, nous présentons la syntaxe et la sémantique de ce langage.

9.3.2.1 Présentation générale

Le Join-Calcul repose sur l'approche *reflexive chemical abstract machine* (CHAM). D'après le modèle CHAM, l'état d'un système est représenté par une *soupe chimique* contenant des événements activés ainsi que des processus en cours d'exécution. L'avancement de l'état du système est effectué par le déclenchement de l'exécution de certains processus lors que certains événements sont activés dans la *soupe*. L'exécution du processus a pour effet de désactiver les événements de la *soupe* qui ont déclenché son exécution et d'en activer potentiellement d'autres. Ainsi, la description du comportement du système est décrit par des règles de réduction comme ce qui suit :

$$R_1 = push(image)|pret(decodeur) \triangleright decodeur(image)$$

Nous appellerons la partie gauche du signe $\triangleright$ un *patron de synchronisation*, et la partie droite une *réaction*. La règle ci-dessus a la sémantique suivante : si les événements (ou messages) $push(i1)$ et $pret(decodeur)$ sont activés dans la *soupe*, alors la règle R_1 va s'appliquer. Ceci va avoir comme effet de déclencher la réaction $decodeur(image)$ et consommer les événements se trouvant dans la partie gauche de la règle.

Maintenant, essayons de proposer une règle décrivant le comportement du contrôleur d'exécution présenté dans la figure 9.9. Une première tentative serait la suivante :

$$R = push(e_1)|push(e_2) \triangleright mix(e_1, e_2)$$

Cette règle dit que la réaction mix sera déclenchée si deux messages $push$ sont activés dans la soupe pour fournir les paramètres e_1 et e_2. Or, cette règle est ambiguë en ce qui concerne la source des ces deux messages. Étant donnée l'exécution parallèle des composants $decodeurs$, ces deux événements peuvent être produits par un seul d'entre eux, ce qui mènera à un dysfonctionnement du système. De plus, si plus de deux messages $push$ sont activés dans la soupe au moment de l'évaluation de l'état du système, le contrôleur peut prendre n'importe quelle paire présente dans la *soupe*. La causalité des messages produits par un décodeur serait alors violée. Il y a donc besoin de pouvoir exprimer la source des messages en assurant une propriété de causalité entre les messages produits par une même source. Pour résoudre ce problème, nous introduisons dans le langage JoinDSL la notion d'interface qui permet de désigner un canal dans lequel doit être présent une événement activé. Ceci permet tout d'abord de sélectionner une source de messages. De plus, ces canaux ont la propriété d'ordonner les messages dans un ordre FIFO, ce qui assure la relation de causalité entre les messages postés via la même interface. On ne parlera alors plus d'une *soupe* globale d'événements mais plutôt d'une soupe d'événements qui sont véhiculés par des canaux (ou files de messages) ordonnés. La règle suivante décrit la règle décrivant le comportement du contrôleur d'exécution suscité conformément au langage JoinDSL.

$$R = I1.push(e_1)|I2.push(e_2) \triangleright I3.mix(e_1, e_2)$$

Afin de présenter une autre difficulté concernant la programmation dans Join-Calcul standard, rajoutons un nouvel élément dans notre architecture. Nous proposons d'enrichir l'architecture de la figure 9.9 avec un autre composant qui envoie au mixeur des sous-titres. Le comportement du mixeur est alors légèrement modifié. Il propose deux réactions différentes. La réaction précédente est gardé car des images peuvent être mixées sans la présence de sous-titres. Or, si un sous-titre est présent en entrée en plus des deux images à mixer, une réaction spécifique permet de mixer l'ensemble de ces trois entrées pour obtenir une image avec sous-titre. Les règles décrivant le comportement du mixeur deviennent alors :

$$R_1 = I1.push(e_1)|I2.push(e_2)I3.subtitle(st) \triangleright I3.mixwithsubtitle(e_1, e_2, st)$$
$$R_2 = I1.push(e_1)|I2.push(e_2) \triangleright I3.mix(e_1, e_2)$$

Le problème concernant les règles ci-dessus est l'indéterminisme en ce qui concerne la sélection d'une règle à exécuter dans le cas ou les deux *patrons de synchronisation* des deux règles sont satisfaits en même temps. En effet, dans le cas où la soupe d'événements contiendrait les trois messages $I_1.push$, $I_2.push$ et $I_3.subtitle$, les deux règles deviennent activables. Pour maîtriser le comportement du système dans ce type de cas, nous nous inspirons de la proposition des concepteurs de Join Java. Il s'agit d'augmenter le calcul avec une propriété déterministe en ce qui concerne l'ordre d'évaluation. Dorénavant, les règles seront évaluées de haut en bas, ce qui permet aux programmeurs d'ordonner les priorités des règles décrivant le comportement des composants. Dans notre cas précis, la règle R_1 serait exécutée si les trois événements qu'elle attend en entrée sont produits. Dans le cas contraire, la règle R_2 sera évaluée.

En résumé, le langage JoinDSL que nous définissons dans le cadre de nos travaux étend le Join-Calcul en deux points. Tout d'abord, nous y intégrons la notion d'interface. Cela permet d'une par d'annoter la source d'un événement en utilisant la notion de canaux de communication, et lie les messages transmis via le même canal avec une relation de causalité. La deuxième extension consiste en la définition d'un ordre d'évaluation des règles, de façon similaire à la proposition de JoinJava. Cela permet aux programmeurs d'associer des priorités aux actions à exécuter.

9.3.2.2 Syntaxe

La figure 9.10 présente la syntaxe du langage JoinDSL en utilisant un formalisme de type BNF. Les opérateurs | et ▷ de la syntaxe originelle du Join-Calcul sont substitués dans cette syntaxe par les opérateurs `&` et =>, respectivement. Les commentaires à la C++ sont permis mais ne sont pas représentés dans la grammaire pour des raisons de lisibilité. Les terminaux du langage sont des expressions régulières identiques à celle d'un identifiant en langage Java. Enfin, un mot clé `empty` est défini pour décrire des règles où la seule réaction associé à un *pattern de synchronisation* consiste en la consommation des événements présents en entrée.

$$
\begin{array}{lcl}
\mathit{Définition} & \rightarrow & \mathit{Règle} + \\
\mathit{Règle} & \rightarrow & \mathit{Pattern}\ \texttt{`=>'}\ \mathit{Réaction}\ \texttt{`;'} \\
\mathit{Pattern} & \rightarrow & \mathit{Message}\ (\ \texttt{`\&'}\ \mathit{Message}\ )* \\
\mathit{Message} & \rightarrow & \mathit{Méthode} \\
\mathit{Réaction} & \rightarrow & \mathit{Méthode} \mid \texttt{`empty'} \\
\mathit{Méthode} & \rightarrow & \mathit{idf}\ \texttt{`.'}\ \mathit{idf}\ \texttt{`('}\ \mathit{idf}\ *\ \texttt{`)'}
\end{array}
$$

FIG. 9.10 – Grammaire BNF du langage JoinDSL.

9.3.2.3 Modèle d'évaluation des règles

La règles sémantiques associée à la syntaxe décrite dans la figure 9.10 sont détaillés dans [May06]. Nous nous proposons dans ce document de présenter le modèle d'évaluation des règles de manière informelle au travers d'un exemple.

Un fichier de règle correspond à la description du comportement d'un composant. Nous allons appeler par la suite un *contrôleur d'exécution*, le mécanisme d'évaluation de règles associé à un composant. Chaque composant aura alors son propre contrôleur d'exécution. Avant de s'intéresser à l'évaluation des règles, intéressons nous d'abord à l'organisation des contrôleurs d'exécution.

La figure 9.11 présente les structures de données qui sont manipulés par un contrôleur d'exécution. Toutes les interface serveurs du composant sont représentées par des files distinctes. Les événements reçus via la même interface sont ordonnés dans un ordre FIFO au sein de ces files. Les éléments se trouvant en tête de chaque file sont activés dans la *soupe d'événements*. Cela permet d'évaluer les événements

dans l'ordre de leur arrivée, ce qui permet d'assurer une propriété de causalité entre les événements reçus via la même interface.

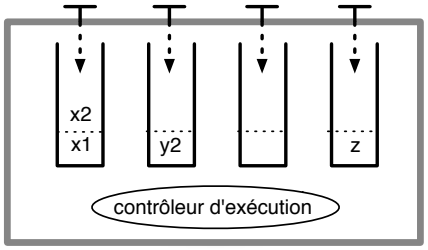


FIG. 9.11 – Architecture d'un contrôleur d'exécution qui permet d'évaluer des règles écrits en JoinDSL.

Lorsqu'une interface serveur du composant est appelée, l'invocation de méthode est transformée en une version sérialisée[2]. Ainsi, il est transformé en un événement et est enregistrée dans la queue associée à l'interface appelée. Si l'enregistrement de l'événement modifie le contenu de la *soupe d'événements*[3], il est temps de la réévaluer pour vérifier si un patron de synchronisation est satisfait. Pour ce faire, un tableau `status` modélisant la *soupe d'événements* est construit. Le contenu de ce tableau est ensuite comparé à tous les patrons reconnus, dans l'ordre de leur déclaration, jusqu'à en trouver un qui correspond à l'état actuel de la *soupe*. Dans le cas où l'on en trouve un, les événements associés sont retirés des files et un appel de méthode est construit pour déclencher la réaction associée du composant à la règle en question. La consommation de certains événements modifie l'état de la *soupe*. Cela peut résulter en la reconnaissance de nouveau patrons. Alors, tant que l'une des files ayant été modifiée contient encore des événements, on recommence le processus d'évaluation à partir de la première règle. Cette itération continue jusqu'à ce qu'aucun patron ne soit reconnu.

Le mot clé `empty` représente une réaction vide. Elle a donc pour effet de consommer uniquement les événements qui ont déclenché son exécution.

Essayons maintenant d'exécuter un scénario simple afin de mieux comprendre le fonctionnement d'un contrôleur d'exécution. Considérons un composant dont le comportement est décrit avec les règles suivantes :

```
i1.A() & i3.E() => i5.r1(); // R1
i1.B() & i2.C() => i5.r2(); // R2
i2.D() & i4.F() => i5.r3(); // R3
```

Il s'agit d'un composant avec quatre interfaces serveur qui peut recevoir au total six types d'événements différents. Trois réactions sont implantées par le composant contrôlé. Elles sont toutes accessibles via l'interface `i5`. Supposons maintenant que la séquence d'événements suivante soit produite en entrée de ce composant :

$$\langle a_1, b_1, c_1, d_1, b_2, c_2, d_2, f_1, f_2, a_2, e_1, e_2 \rangle$$

La figure 9.12 présente l'évolution de l'état du contrôleur. Aucun patron n'est reconnu jusqu'à ce que l'événement e_1 soit produit. Les files sont remplies alors par les événements qui leur sont associées. Au moment de la réception de l'événement e_1, le patron décrit par la règle `R1` est reconnu. Les événements

[2]La version sérialisée d'un appel de méthode contient l'identificateur du méthode invoquée ainsi que les paramètres à passer.
[3]Le contenu de la soupe est modifié si est seulement si l'événement a été enregistré dans une file vide.

`A` et `E` sont alors retirés des files concernées et la réaction `r1` est exécutée. La consommation des événements fait évoluer l'état de la soupe : elle contient alors des événements `B`, `C` et `F`. Cela fait que la règle de la ligne 2 est reconnue. La réaction `r2` est alors exécutée après avoir consommé les événements `B`, `C`. Cela modifie encore l'état de la *soupe*. L'évaluation des règles continue jusqu'à ce que plus aucun patron ne soit reconnu. Dans le cas de notre exemple, l'évaluation est terminée lorsque tous les événements sont consommés.

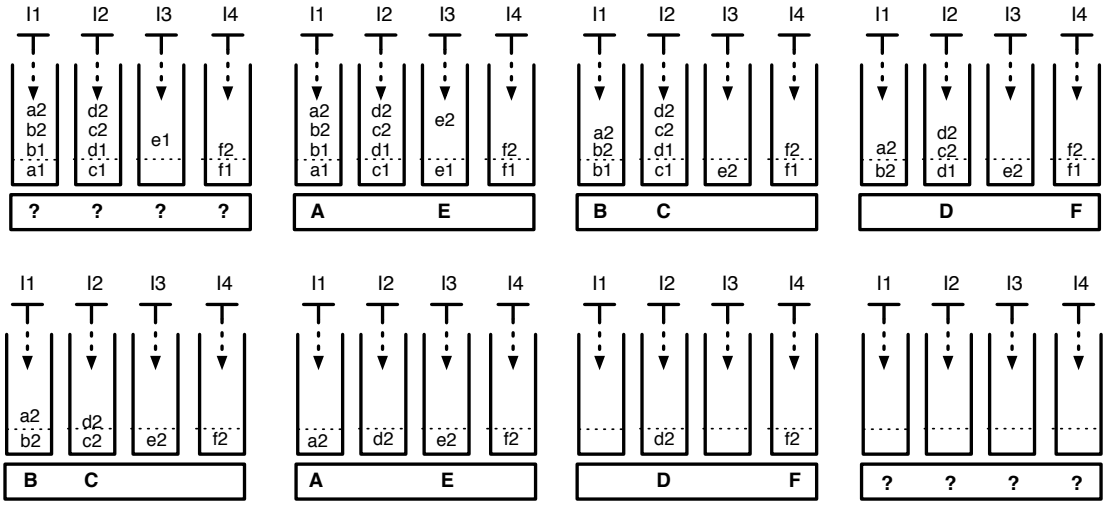


FIG. 9.12 – Illustration d'un scénario d'exécution de l'évaluation de règles.

9.3.3 Architecture d'un composant avec contrôleur d'exécution

Idéalement, l'implantation d'un contrôleur d'exécution devrait être intégré dans le modèle FRACTAL comme élément de *membrane* spécifique. Celui-ci serait alors réifié comme un composant de contrôle qui expose les interfaces sur lesquelles différents appels de méthodes peuvent être acceptés. Ce composant serait ainsi connecté au contenu pour déclencher les réactions implantées au sein de ce dernier. Or, l'implantation actuelle du langage FRACTALADL ainsi que le modèle d'implantation utilisé dans le canevas THINK imposent un certain nombre de limitations en ce qui concerne l'implantation des modules de membrane. De manière générale, il est impossible, à l'heure actuelle, de réifier les modules de membrane sous forme de composants et de définir des interfaces et des connections propres à ces derniers. L'amélioration de ces aspects est prévue dans la définition de la prochaine version du modèle de composants FRACTAL.

Compte tenu de ces limitations, nous avons utilisé les composants composites pour réifier un composant THINK contenant un contrôleur d'exécution sous forme de composant. La figure 9.13 présente la description d'architecture du composant mixeur mentionné précédemment dans la section 9.3.1. Remarquons que la définition de composant `composant-mixeur` contient deux composants. Le premier est le composant assurant le contrôle de l'exécution. Son contenu est décrit en JoinDSL comme expliqué en section 9.3.2.1. Le deuxième est le composant `contenu` dont la définition est décrite par la définition du composant `mixeur` dans la figure 9.7. Ainsi, les interfaces du contrôleur sont exportées à l'extérieur du composant de manière à cacher l'interface du contenu qui serait appelé en fonction de la reconnaissance des patrons de synchronisation.

```
<definition name="composant-mixeur">
  <interface name="i1" signature="FluxImage" role="server" />
  <interface name="i2" signature="FluxImage" role="server" />
  <interface name="i3" signature="FluxImage" role="client" />

  <component name="controleur">
    <interface name="i1" signature="FluxImage" role="server" />
    <interface name="i2" signature="FluxImage" role="server" />
    <interface name="i3" signature="Mixeur" role="client" />
    <content class="comportement-mixeur" language="joinDSL" />
  </component>

  <component name="contenu" definition="mixeur" />

  <binding client="this.i1" server="controleur.i1" />
  <binding client="this.i2" server="controleur.i2" />
  <binding client="controleur.i3" server="contenu.entree"/>
  <binding client="contenu.sortie" server="this.i3"/>
</definition>
```

FIG. 9.13 – La description ADL du composant mixeur qui est contrôlé par un contrôleur d'exécution conformément à la figure 9.9.

9.3.4 Génération de code pour JoinDSL

Le support nécessaire à la prise en compte du JoinDSL comme un langage d'entrée a été intégré dans le compilateur ADL présenté tout au long de la deuxième partie de ce document. Cela permet de générer les composants de contrôle d'exécution en fonction des règles écrites dans des fichiers JoinDSL. Les modules appropriés à ce support ont étés intégrés dans la personnalité de génération de code de notre compilateur ADL. Cette exercice d'extension de la chaîne d'outils a eu des impacts sur le module de chargement et de traitement du compilateur ADL. Nous décrivons par la suite l'extension de ces deux modules.

9.3.4.1 Extension du module de chargement

Rappelons que le module de chargement est en charge de construire un AST qui unifie l'ensemble des informations lues dans des fichiers d'entrée. Ce module a été étendu afin de prendre en compte le langage JoinDSL. La figure 9.14 montre l'intégration des composants qui sont consacrés au chargement des entrées JoinDSL dans la chaîne de chargement originelle conçue pour la personnalité de génération de code de notre compilateur ADL. Cette figure montre que quatre composants sont crées pour charger des descriptions JoinDSL :

- Le composant *JoinDSL Loader* est en charge d'enrichir l'AST avec des sous AST correspondant aux règles définies dans les fichiers JoinDSL. Pour ce faire, il analyse les définitions des composants pour détecter ceux dont le contenu est décrit en JoinDSL. Lorsqu'un tel composant est détecté, le chargement du fichier JoinDSL mentionné par ce composant est demandé au *JoinDSL Decorator*. Un AST modélisant les règles du fichier en question est obtenu en retour. Cet AST est fusionné avec l'AST de l'architecture en rajoutant le premier comme une décoration du nœud modélisant le contenu du composant traité.
- Le composant *JoinDSL Decorator* demande le chargement d'un AST au composant *JoinDSL Semantic Checker* et effectue des opérations de décoration au sein de cet AST avant de le retourner au composant *JoinDSL Loader*. Typiquement, des pointeurs sont rajoutés au niveau des règles pour associer ces dernières aux interfaces qui sont concernées. Cela permet plus tard au module de traitement d'accéder plus facilement à certains informations dans l'AST qui sont requises pour

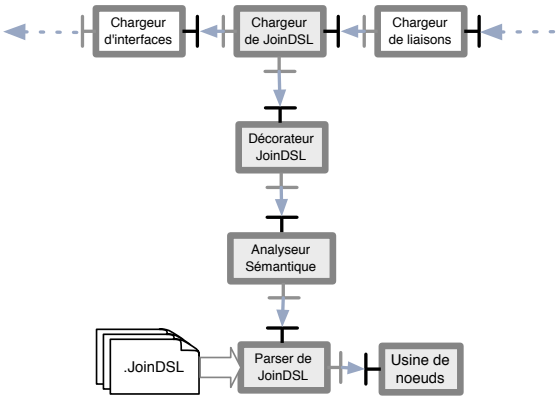


FIG. 9.14 – Extension du module de chargement pour prendre en compte les fichier JoinDSL.

le traiter les règles JoinDSL.

- Le composant *JoinDSL Semantic Checker* demande le chargement d'un AST au composant *JoinDSL Parser* et effectue certaines opérations d'analyse sémantiques avant de retourner l'AST au composant supérieur. L'ensemble des règles sémantiques vérifiées par ce composant sont décrites dans [May06] (section 4.1.2, page 29).
- Enfin, le composant *JoinDSL Parser* qui se trouve tout au bout de la chaîne est en charge de lire un fichier JoinDSL pour construire l'AST lui correspondant. Ce composant est implanté en utilisant l'outil JavaCC. En effet, grâce à cet outil, un parseur de fichier JoinDSL a été généré tout simplement en décrivant la grammaire et les mots clés présentés dans la figure 9.10. Remarquons que le composant *JoinDSL Parser* utilise une version spécialisée de l'usine de nœuds pour la construction de l'AST.

La spécialisation de l'usine de nœuds est effectuée à l'aide d'une description de grammaire. La figure 9.15 présente la description qui a été mise en œuvre. Selon cette grammaire, la racine d'une AST de JoinDSL est définie par un nœud de type `regles`. Ce nœud peut contenir plusieurs fils de type `regle`. Chaque nœud `regle` a deux fils pour le `pattern` et la `reaction`. Un `pattern` a un ensemble de fils référent aux événements qui le définissent. Un événement est désigné par une paire constituée d'une interface et d'un nom de méthode. Les nœuds `evenement` peuvent contenir des arguments en tant que fils si la méthode correspondante requière des paramètres formels. Enfin une `reaction` est définie par une référence de méthode. Dans le cas où la réaction serait vide, ceci est indiqué par un booléen au niveau du nœud `reaction`.

9.3.4.2 Extension du module de traitement

Un greffon (*plugin*) a été mis au point pour étendre le compilateur ADL de manière à générer automatiquement les composants de contrôle d'exécution. La génération de tels composants peut être traitée en deux parties : la génération de l'implantation et la génération de l'encapsulation du composant. Le premier requière l'écriture de nouveaux composants de traitement pour générer du code à partir des règles JoinDSL. Or, puisque les composants de contrôle d'exécution sont encapsulés de la même manière qu'un composant normal, la deuxième partie de la génération est déjà fournie dans le générateur de code standard de mise au point pour THINK. Par conséquent, le plugin de traitement spécifique à JoinDSL est implanté en dérivant le plugin de génération de code présenté dans la section 7.2.3. Comme présenté dans la figure 9.16, cette extension surcharge le composant qui est en charge de l'intégration du fichier d'implantation (voir la figure 7.5). En effet, le couple *visiteur, backend* en question est remplacé

```
<!ELEMENT regles ( regle* ) >
<!ELEMENT regle ( pattern , reaction ) >
<!ELEMENT pattern ( evenement* ) >
<!ELEMENT reaction ( methodReference ) >
<!ATTLIST reaction
        emptyReaction CDATA #REQUIRED

<!ELEMENT methodReference ( argument* ) >
<!ATTLIST methodReference
        interface CDATA #REQUIRED
        name CDATA #REQUIRED

<!ELEMENT evenement ( argument* ) >
<!ATTLIST evenement
        interface CDATA #REQUIRED
        name CDATA #REQUIRED

<!ELEMENT argument EMPTY >
<!ATTLIST argument
        name CDATA #REQUIRED
```

FIG. 9.15 – Description de la grammaire de l'AST modélisant les règles décrits en JoinDSL.

par d'autres qui assurent la génération d'un contenu qui assure le contrôle de l'exécution conformément aux règles JoinDSL associées.

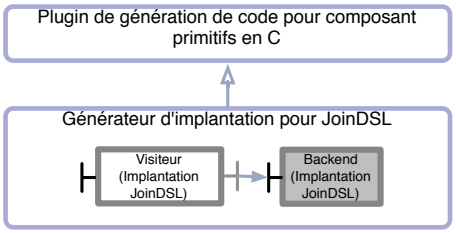


FIG. 9.16 – Plugin de génération de code pour JoinDSL dérivant du plug-in présenté dans la figure 7.5.

Quant à la compilation optimisée des machines à états réifiant l'implantation des contrôleurs d'exécution, nous nous sommes inspirés des techniques présentées dans [FM98] qui permettent d'optimiser considérablement les performances en termes de temps d'exécution et d'espace mémoire en utilisant des masques de bits pour modéliser les patterns de synchronisations et les types d'événements acceptés en entrée. Nous avons dû légèrement modifier ces techniques pour y intégrer les notions d'interface et de priorité d'évaluation. Les détails des techniques de compilation et des structures de données déployés sont présentés dans [May06].

9.3.5 Modèle d'exécution des composants

Comme présenté dans les sections précédentes, les descriptions de comportement écrites en JoinDSL sont utilisés pour générer des contrôleurs d'exécution. Cela permet entre autres de découpler l'implantation des composants de leur modèle d'exécution et de synchronisation. En effet, la gestion de tous les aspects liés à l'exécution d'un composant est assurée par son contrôleur d'exécution. Il est alors possible de générer différentes formes de contrôleurs d'exécution pour exécuter les composants selon des modèles différents.

Nous distinguons, dans une première partie, deux modèles d'exécution : le modèle synchrone et le modèle asynchrone. Le premier est conforme au modèle classique à base de *threads*. Selon ce modèle, la

fonction appelée est exécutée dans le flot d'exécution de l'appelant. L'appelant reste alors bloqué jusqu'à ce que l'appelé termine son exécution. Dans ce cas, le comportement des contrôleurs d'exécution peut être décrit comme suivant : la machine à états évalue les patrons de synchronisation à chaque appel, une réaction est exécutée immédiatement si un patron est reconnu, ou bien l'évaluation se termine sans réaction dans le cas contraire.

Du fait que l'appelant attend l'exécution de l'appelé, le modèle synchrone est peu adapté au parallélisme. Afin de rendre notre modèle de programmation compatible avec une exécution parallèle transparente, nous imposons deux restrictions. D'abord, nous imposons que les réactions ne puissent avoir des valeurs de retour. Ensuite, tant qu'une des réactions d'un composant est en exécution, aucune autre réaction du même composant ne peut être exécutée. Cette dernière restriction permet de ne pas avoir de conflit d'accès concurrents aux variables d'instance d'un même composant. En effet, dans le cas où l'appelant n'attend pas de valeur de retour et que les deux extrémités ne partagent aucune donnée, l'appelé peut être exécuté en parallèle à ce dernier.

La figure 9.17 illustre l'architecture d'un contrôleur d'exécution adapté au modèle d'exécution asynchrone. Remarquons que ce dernier est connecté dans ce cas à un ordonnanceur qui est en charge de coupler les réactions à un ensemble de ressources d'exécution. Le comportement du contrôleur peut être résumé comme suit : la machine à états évalue les patrons de synchronisation à chaque appel et détermine si oui ou non une réaction devient actionnable. Si oui, cette réaction est enregistrée au sein de l'ordonnanceur et sera exécutée plus tard lorsqu'une ressource d'exécution sera disponible. Sinon, l'évaluation se termine sans réaction. De cette manière, l'activation des réactions est décidée au plus tôt, dans le flot d'exécution de l'appelant. Par contre, l'exécution réelle de la réaction est réalisée dans un autre flot d'exécution qui lui sera associé par l'ordonnanceur. Ainsi, l'ordonnanceur peut exécuter les réactions sur un ensemble de *threads* virtuels ou physiques en fonction de la plate-forme matérielle sous-jacente.

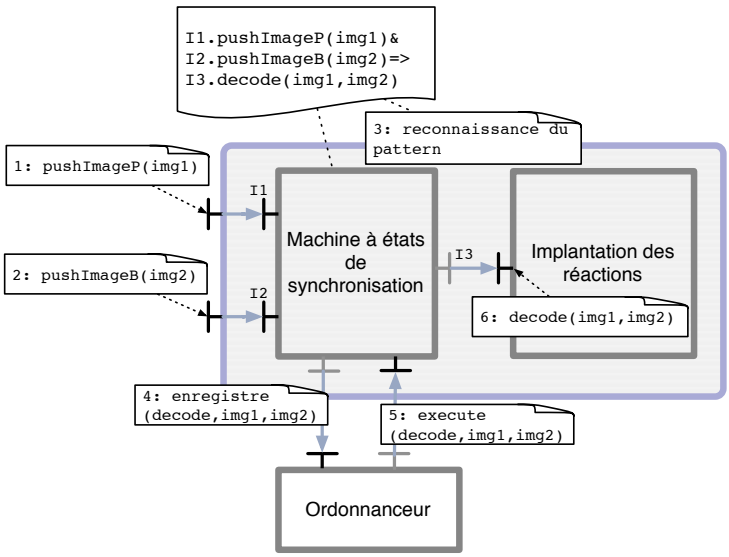


FIG. 9.17 – Architecture d'un composant adapté au modèle d'exécution asynchrone

9.4 Mise-en place d'un décodeur MPEG-2

L'environnement de programmation proposé a été évalué en utilisant une application de décodage de flux vidéo. Nous présentons dans cette section les caractéristiques de cette application de *streaming* et l'architecture à base de composants à grain fin qui a été mise en œuvre. Ensuite, nous dressons une

évaluation quantitative et qualitative et discutons des avantages et inconvénients de cette approche.

9.4.1 Présentation des travaux concernant le décodage de flux MPEG-2

Plusieurs choix possibles ont été explorés afin de trouver une application qui permette d'évaluer au mieux le modèle de programmation que nous proposons dans le cadre de nos travaux. Un des critères majeurs était naturellement la disponibilité d'un code exemplaire pour la réduction de l'effort de développement et de la disposition d'une base de comparaison. Alors que notre expérimentation initiale concernait le décodage de flux H.264, nous avons trouvé que l'implantation de référence que nous avions auparavant utilisée était trop complexe pour être réarchitecturée dans un délais acceptable. En contrepartie, une application de *streaming* trop simple comme AC3 ou MP3 nous semblait insuffisante pour se confronter à des problèmes significatifs.

Nous avons finalement opté pour la mise en œuvre d'un décodeur vidéo respectant la norme MPEG-2. Ce standard est utilisé depuis quelques années dans des produits grands publics tel que le DVD et la télévision sur internet. Une introduction et la spécification de la norme peuvent être trouvés respectivement dans [Ros06] et [MPEb]. Les raisons qui nous ont conduit à ce choix sont doubles. Premièrement, l'implantation d'un décodeur de MPEG-2 est considérablement moins complexe que celle d'une norme plus récente telle que H.264. De plus, la parallélisation et l'organisation des décodeurs MPEG-2 a constitué le sujet de nombreux travaux de recherche, ce qui fournit une base de comparaison intéressante.

Une équipe de recherche de la société STMicroelectronics s'est intéressée à la parallélisation d'un décodeur MPEG-2 sur un modèle d'exécution à base de *threads*. [SMM04] compare les estimations théorique des gains en performance aux gains obtenus en pratique sur une plate-forme SMP contenant quatre processeurs. Alors que les gains théoriques sont assez prometteurs, l'accélération mesurée reste à la hauteur d'un facteur $1,5$. Ceci est expliqué par le non déterminisme du comportement à l'exécution, qui cause des lectures de fichiers ordonnancées de façon aléatoire et rend le coût des attentes pour la synchronisation considérables. [CLW02] et [BFS97] proposent des stratégies de parallélisme au niveau des données. Le premier propose un parallélisme à grain fin au niveau des images et des macroblocs alors que le deuxième propose un parallélisme de plus haut niveau liés aux groupes d'images indépendants.

La seule proposition qui utilise des éléments d'architecture pour organiser ce type de décodeur est StreamIt. Les concepteurs de ce langage présentent dans [DHRA06] un découpage à grain fin des modules composant un décodeur MPEG-2. Les résultats présentés exposent une architecture très compréhensible. Si les auteurs annoncent qu'ils s'intéressent à la parallélisation et aux mesures de performances, aucun résultat de ce type n'est publié à ce jour.

9.4.2 Architecture à base de composants du décodeur

Nous nous sommes inspirés de trois sources pour mettre au point le décodeur MPEG-2 à base de composants : la spécification de la norme qui nous a donné une idée globale de l'architecture qu'il faudrait mettre en œuvre [MPEb], l'architecture d'implantation qui a été proposée par les concepteurs de StreamIt qui a illustré un découpage possible [DHRA06] et l'implantation de référence de la norme qui nous a permis de réutiliser le code fonctionnel du décodeur [MPEa].

La figure 9.18 présente l'architecture mise en œuvre. Tous les composants présents dans le schéma, sauf celui intitulé `Framestore`[4] sont écrits selon la logique de développement que nous proposons. Chaque composant réifie des fonctions bien définies prenant un certain nombre de paramètres en entrée. Ces paramètres sont produits par divers composants faisant partie du décodeur. Ainsi, les contrôleurs d'exécution de chaque composant, dont le comportement est décrit en JoinDSL, s'occupent de collecter

[4]Le composant `Framestore` est en charge de donner accès à des images décodées précédemment. Compte tenu la quantité de données échangées avec ce composant, nous avons opté pour un modèle d'exécution synchrone. Par conséquent, ce composant est écrit dans le modèle standard et n'utilise pas de contrôleur d'exécution.

les paramètres nécessaires à l'exécution d'une fonction implantée par le composant et de déclencher l'exécution de cette dernière lorsque l'ensemble des paramètres requis sont reçus.

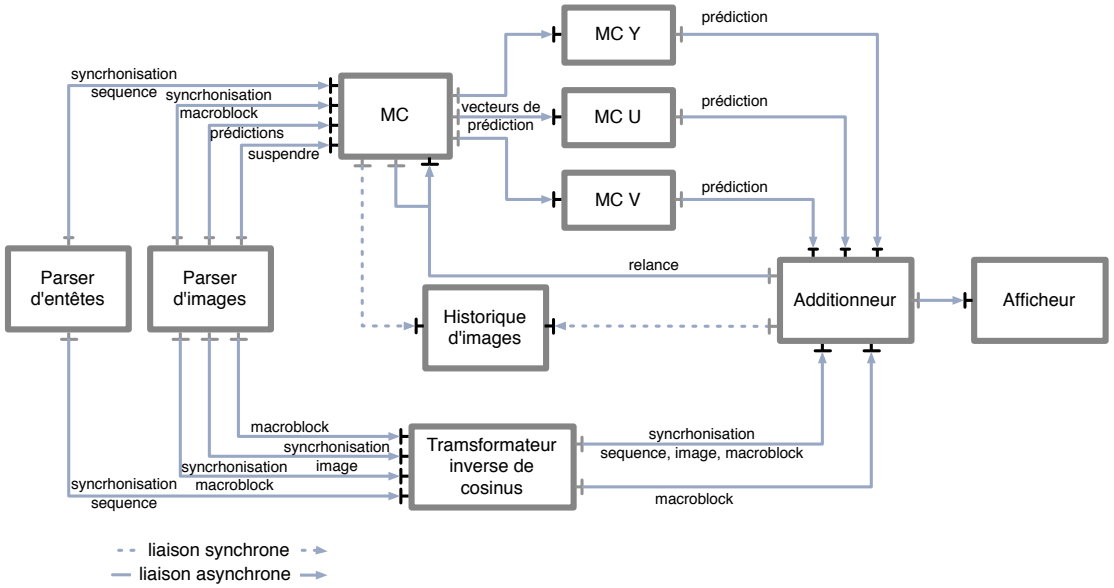


FIG. 9.18 – Architecture à base de composants du décodeur MPEG-2 mis en œuvre

Le décodeur reçoit deux flux séparés en entrée. Les composants `Header Dec` et `Picture Dec` sont en charge de parser le flux de contrôle indiquant respectivement les changements de séquence et le flux de données. Le flux est analysé au niveau des macroblocs. Ces derniers peuvent être de deux types. Les macroblocs de type I contiennent des morceaux d'image compressés. En revanche, les macroblocs P contiennent des vecteurs de mouvement par rapport à des macroblocs I ainsi que des indices permettant d'effectuer des corrections d'erreur. Les macroblocs sont envoyés vers le composant de décodage `IDCT` qui est en charge d'appliquer une transformation de Fourier inverse pour obtenir les blocs d'image décodés. Le décodage des macroblocs B nécessite une opération plus complexe. Il sont d'abord traités par le composant `Motion Compensation` qui déduit à partir des vecteurs de mouvement et des images précédemment décodées des changements à appliquer pour obtenir un nouveau bloc d'image décodé. Une partie de ce traitement est effectuée en parallèle par différents blocs qui traitent séparément les différentes dimensions en couleur Y, U, V de l'image. Une fois les changements à effectuer déterminés, le composant `Adder` applique ces changements et renvoie l'image décodée vers le composant `Output` qui est en charge d'exporter le résultat sous le format attendu.

L'exécution d'une grande partie des composants est dirigée par le flux de données. Par conséquent, les fichiers JoinDSL décrivant le comportement de tels composants associent des réactions à exécuter à des jointures de paramètres d'entrée. Parce que les accès au composant `Framestore` sont asynchrones, il nécessite des mécanismes de synchronisation additionnels. En effet, le contenu du composant `Framestore` doit être modifiée à la fin de chaque séquence d'image B pour mettre à jour l'image précédente qui a été décodée. Cette modification nécessite la synchronisation des composants `Motion Compensation` et `Adder` pour éviter des accès concurrents. Deux signaux de contrôle sont alors utilisés en plus des données. Lors qu'une fin de séquence est détecté par les parsers, un signal `suspend` est envoyé au composant `Motion Compensation` pour suspendre son activité. Le composant `Adder` enverra le signal `resume` pour relancer ce dernier lorsque la modification du `Framestore` sera terminée.

9.4.3 Évaluation

Une fois le décodeur MPEG-2 implanté avec une architecture à base de composants utilisant des descriptions de comportement en JoinDSL, nous avons effectué un certain nombre d'expérimentations. Nos objectifs principaux étaient d'évaluer la parallélisation de cette architecture sur des plates-formes matérielles différentes et d'effectuer des mesures de performances. Nous commençons par présenter les caractéristiques de plates-formes que nous avons utilisées. Ensuite, nous discutons des résultats en deux volets, quantitatif et qualitatif.

9.4.4 Plates-formes matérielles

Trois plates-formes matérielles ont été utilisé au cours de nos expérimentations :

- Une station Intel/Linux a constitué la plate-forme de développement. Un ordonnanceur qui permet d'exécuter plusieurs composants en parallèle a été mis au point en se basant sur la bibliothèque *pthread* fournie par Linux.
- Un processeur *multi-threadé* ST31-MT développé au sein de STMicroelectronics a été utilisé pour évaluer les performances du décodeur sur des ressources physiques partiellement séparées. En effet, le processeur MT en question permet de multiplexer plusieurs *threads* physiques sur une partie opératoire unique pour exploiter le temps perdu en attente de défauts de cache.
- Enfin, un simulateur de plate-forme SMP intégrant un nombre configurable de processeurs ST230 a constitué une autre plate-forme d'expérimentation. La particularité de cette plate-forme est la présence de plusieurs ressources physiques permettant une exécution réellement parallèle.

Dans le cas des deux dernières plates-formes, nous avons utilisé un support d'exécution ad-hoc développé par l'équipe de développement des processeurs ST230. Ce support implante, de manière efficace, un sous ensemble de l'API *pthread*, ce qui nous a permis de réutiliser, sur ces plates-formes, l'ordonnanceur que nous avions mis au point pour la station Linux.

9.4.5 Aspects quantitatifs

Afin d'évaluer les performances du décodeur mis au point sur des architectures matérielles différentes, nous décodé le même film sur plusieurs plates-formes. Le film décodé contient 32 images QCIF, ce qui correspond à un temps d'affichage d'un peu plus d'une seconde et demi. Chaque image est divisée en 330 macroblocs. Ceci choix de longueur de film nous parait assez long pour que le comportement du décodeur atteigne un point de fonctionnement stable et assez court pour que l'on puisse obtenir des résultats de simulations dans des délais acceptables[5].

Le tableau 9.2 présente le temps total traduit en nombre de cycles qui a été nécessaire pour effectuer le décodage sur les plates-formes considérées. La version synchrone représente l'exécution de l'ensemble de l'application dans un flot d'exécution unique. Elle et exécuté sur un processeur ST230 ayant un seul *thread* physique. Comme présentée dans la section 9.3.5, les contrôleurs d'exécution sont compilés dans ce cas de manière à ne nécessiter aucun ordonnacement, ni synchronisation. Comparée à l'implantation de référence, celle-ci est à peu près trois fois plus lente. Cette perte de performance est en effet lié à deux aspects : l'efficacité des contrôleurs d'exécution et l'architecture de découpage. Premièrement, l'implantation des contrôleurs d'exécution de notre protoype n'est pas encore mature et nous pensons que leur optimisation pourrait améliorer les performances globales. Deuxièmement, et sans doute le plus important, notre découpage architectural qui est influencé par une perspective de répartition sur plusieurs processeurs semble moins efficace que la version monolithique de l'implantation de référence lorsqu'il s'agit d'exécuter le application sur un seul processeur. Le résultat de cette comparaison doit être relativisé en tenant du fait que notre version a l'avantage d'être mieux architecturée et automatiquement

[5]En effet, la simulation d'un décodage d'une telle longueur avec une précision au niveau cycle de processeur prend en moyenne quelques jours.

répartissable sur plusieurs processeurs. Rappelons l'importance du découpage architecturale que nous avons pu observer dans les deux expérimentations précédentes : alors que la réécriture du décodeur H.264 (présenté dans la section 9.1) sans modification architecturale (voir la figure 9.2) résultait en des performances tout à fait comparables à sa version initiale, la restructuration propre tel que proposée par l'architecture THINK4L résultaient en des pertes de performances importantes.

Mode d'exécution	Plate-forme	Temps d'exécution (en million de cycles)	
Synchrone	ST230	1 thread	900
Asycnhrone	ST230	1 thread	1500
	ST230 MT	4 threads	1250
		7 thread	1580
	ST230 SMP	4 thread	880
		7 thread	-

TAB. 9.2 – Temps d'exécution du décodeur MPEG-2 sur différentes plates-formes matérielles.

Toutes les autres lignes du tableau présentent des résultats obtenus avec un modèle d'exécution asynchrone. Les contrôleurs d'exécution sont alors générés de manière à découpler l'exécution des réactions sur des flots séparés. La version *mono-thread* permet de voir le surcoût de ce mécanisme de découplage. On aperçoit que le fait d'enregistrer des réactions qui seront exécutées plus tard rend l'exécution de l'application 1,7 fois plus lente[6]. Dés gains sont obtenus lorsque la version asynchrone est exécute sur des architectures avec des ressources physiques multiples. Alors que la plate-forme MT montre peu d'accélération avec quatre *threads*, la plate-forme SMP présente une accélération importante. Nous observons que sur la plate-forme MT, l'augmentation du nombre de *threads* à 7 diminue les performances. Ceci est sûrement dû au coût de la gestion sous-jacente de *threads* qui devient trop important. Nous n'avons pu mesurer les performances sur une plate-forme SMP a 7 processeurs car la simulation n'a pu être terminée au bout de quelque jours.

Enfin, bien que nous ayons évalué notre approche sur des architectures matérielles à mémoire partagée (en raison de la disponibilité de ces plates-formes), nous remarquons que notre approche est plus adaptée à des architectures à mémoire répartie. En effet, puisque la programmation à base de composants fournit une encapsulation forte des données, il est nécessaire de faire circuler, entre les composants, un flux de données plus important que dans le cas d'une programmation à base de *threads* classique. Néanmoins, la tendance au niveau des architectures matérielles étant plutôt favorable à l'utilisation de la mémoire répartie, notre approche nous semble justifiée pour la programmation des applications de *streaming* à destination des prochaines générations de plates-formes.

9.4.6 Aspects qualitatifs

L'approche proposée nous satisfait en grande partie en ce qui concerne la programmation des application de *streaming*. En effet, grâce à cette méthodologie, deux métiers peuvent être réellement distingués dans le processus de développement. Les programmeurs de composants sont ceux qui ont l'expertise des fonctions et de l'algorithmique à implanter. Leur tâche est simplifiée car ils sont dégagés des problèmes

[6]Ce résultat est obtenu par la comparaison des version asynchrone et synchrone exécutées par un seul *thread*.

liés au modèle d'exécution ou à la synchronisation. Les architectes sont chargés d'assembler une configuration optimale du logiciel en tenant compte des spécificités de la plate-forme cible. Ils disposent des outils de haut niveau comme l'ADL et JoinDSL pour adapter les composants dans un contexte d'exécution spécifique. En particulier, au travers de notre expérimentation, nous sommes plutôt satisfait du langage JoinDSL qui permet une bonne maîtrise de la synchronisation des composants dans un modèle d'exécution à base d'événements. Remarquons que ce langage peut être encore étendu pour donner des constructions de synchronisation plus sophistiquées (réception d'un nombre fixé d'éléments du même type, synchronisation en fonction des valeurs de messages, etc.) et pour spécifier les événements qui seront produits par la réaction déclenchée. En effet, la spécification des événements produits peut ouvrir des perspectives de compilation d'un ordonnancement statique de l'ensemble de composants pour optimiser les performances.

Enfin, nous avons démontré que la génération automatique des contrôleurs d'exécution des composants à partir de leur description de comportement permet d'adapter ces derniers à des modèles d'exécutions différents. Cela permet entre autre de paralléliser ou de répartir les composants d'une application. Grâce aux outils de génération de code de haut niveau qui supportent les langages ADL et JoinDSL, les architectes peuvent explorer différents choix architecturaux afin de trouver une solution optimale.

9.5 Conclusion

Ce chapitre était dédié à la mise en œuvre d'applications de streaming reparties à destination des MPSoC. Deux applications ont été présentées dans l'ordre chronologique de leur développement.

La première application était un décodeur H.264 monolithique qui a été *componentisé* sans modification de son architecture d'implantation. Cette expérimentation nous a permis de tirer les conclusions suivantes. Premièrement, une architecture monolithique, qu'il soit implantée à l'aide des composants ou non, reste très difficilement maîtrisable, modifiable et répartissable. Deuxièmement, l'utilisation des composants Think, même à une granularité très fine, ont peu d'impacte sur les performances globales comparées à une implantation entièrement écrite en C. Cela confirme que les pertes de performances telles que nous avons constaté dans le chapitre précédent sont plutôt dues à une conception privilégiant la modularité et la clarté architecturale. Troisièmement, nous avons constaté que les propriétés de répartition des composants d'une telle application étaient non seulement liées à son architecture modulaire, mais aussi à la transparence de l'implantation de ses composants vis-à-vis de leurs modèles d'exécution et de synchronisation.

Ce dernier point nous a motivé à investir dans une extension du modèle de programmation des composants FRACTAL. Dans ce cadre, nous avons opté pour un modèle d'exécution événementiel, pour ses bonnes propriétés en ce qui concerne la programmation répartie, et avons proposé le langage JoinDSL pour simplifier la gestion de la synchronisation des composants répartis. À l'aide de ce langage, les concepteurs peuvent décrire les contraintes de synchronisation qui sont nécessaires à l'exécution d'un composant sous forme de jointure d'événements. Le support nécessaire à cette extension (i.e. la génération des machines à états pour la synchronisation) a été implanté sans difficulté dans la chaîne de compilation sous forme d'un *plugin*. Le développement de ce dernier a démontré les capacités de la chaîne proposée à accepter un nouveau langage d'entrée et à effectuer les opérations de vérifications et de génération de code associées.

Pour évaluer l'approche suscitée, un décodeur MPEG-2 a été développé. Concernant les aspects quantitatifs, nous avons mesuré des pertes de performances importantes par rapport au décodeur de référence, ayant une implantation monolithique en C. Ces pertes sont à relativiser car (i) notre version à base de composants constitue une première implantation, sans doute un peu naïve pour obtenir de bonnes performances et (ii) elle propose une architecture répartissable, uniquement à l'aide des options de compilation et de modification de description d'architecture. Les évaluations de performance effectuées ont

montré l'importance de pouvoir explorer facilement des stratégies de répartition différentes. En effet, nous avons observé des taux d'accélérations importantes sur des architectures de type SMP. Enfin, bien que nos évaluations soient effectuées uniquement sur des architectures matérielles à mémoire partagée pour des raisons de disponibilité de plate-forme, notre approche nous semble tout à fait compatible avec des architectures à mémoire répartie.

Chapitre 10

Conclusion

Dans ce dernier chapitre, nous dressons un bilan des apports de nos travaux et énumérons les perspectives qui pourront êtres poursuivies.

10.1 Principaux apports

Les systèmes sur puce sont devenus de véritables systèmes répartis intégrant plusieurs processeurs couplés à des architectures de mémoire variées. Cette transformation se traduit par une croissance importante du nombre de logiciels déployés sur ce type de plates-formes. Or, la programmation de telles plates-formes n'est pas triviale. En plus de la difficulté de programmation d'un système réparti hétérogène, elle nécessite une maîtrise totale de la programmation de bas niveau afin d'exploiter au mieux les spécificités du matériel. Pour ces raisons, les concepteurs de systèmes sur puce cherchent à élaborer des modèles de programmation pouvant simplifier le processus de développement de logiciel.

Le travail de thèse présenté dans ce rapport s'inscrit dans ce contexte. Notre objectif était de proposer une méthodologie de développement de logiciels à base de composants, adaptée aux contraintes des systèmes sur puce. Pour ce faire, nous avons analysé l'état de l'art des modèles de composants existants et choisi le modèle FRACTAL qui convenait aux besoins identifiés. Puis, nous avons mis au point une infrastructure de développement permettant d'assembler des logiciels à base de composants FRACTAL en utilisant des descriptions d'architectures. Enfin, nous avons évalué cette infrastructure en programmant deux applications significatives dans le cadre des systèmes sur puce : la première illustre le développement de systèmes d'exploitation pour ceux-ci, la deuxième illustre la définition d'un nouveau langage facilitant l'expression de la synchronisation au sein d'application de traitement de flux multimédia répartis.

Les sections suivantes rappellent les principales contributions de nos travaux.

Proposition d'un canevas logiciel pour la construction de compilateurs ADL

Notre thèse se base sur l'adoption de la programmation à base de composants à tous les niveaux logiciels des systèmes sur puce, de la construction de systèmes d'exploitation, au développement des applications multimédias. Nous défendons l'idée que cette technologie est dotée des bases convenant à la programmation de systèmes répartis complexes dont les systèmes sur puce font partie. Compte tenu de la diversité des contraintes que l'on peut rencontrer dans différents domaines de calcul de ces plates-formes hétérogènes, la flexibilité des modèles de composants, c'est-à-dire leur capacité d'extension pour satisfaire des besoins spécifiques, constitue une propriété cruciale.

Ce raisonnement nous a amené à considérer le modèle FRACTAL comme base de nos travaux. Nous avons repris une de ses implantations, THINK, et nous avons contribué à son amélioration dans la perspective de la rendre plus légère et plus flexible. Nous avons montré au travers du développement de plusieurs applications et systèmes (e.g. THINK4L, décodeurs H.264 et MPEG-2) que THINK satisfaisait

les contraintes de mise en œuvre de logiciels répartis sur puce.

La programmation à base de composants permet un raisonnement au niveau de l'architecture des logiciels. Ce raisonnement doit être supporté par des outils de compilation permettant d'automatiser en partie le processus de développement des logiciels. Or, d'une part, le caractère ouvert du modèle de composants considéré et, d'autre part, la diversité des utilisations possibles dans le cadre de la programmation de systèmes sur puce, rend très difficile la construction d'une chaîne d'outils universelle.

En partant de cette problématique, nous avons proposé un canevas logiciel facilitant la mise au point d'outils spécifiques [LÖQS07]. Le canevas proposé se concentre sur la définition d'un certain nombre de patrons de programmation et de composants utilitaires et n'impose aucune interface de programmation. En résumé, ce canevas définit une architecture minimale pour constituer une chaîne d'outils ADL qui peut être étendue par les tierces parties par le biais du développement de *plugins*. Une des particularités intéressantes de cette architecture est qu'elle repose sur un AST extensible permettant d'intégrer les informations décrites dans des langages de description différents. De plus, un canevas de tâches extensible est fourni pour décrire les dépendances des opérations de traitements, afin d'exécuter ces dernières en respectant des contraintes d'ordonnancement.

Nous avons utilisé ce canevas logiciel pour mettre au point un outil de traitement d'ADL dédié à la génération de code. Cet outil a pour fonction d'automatiser l'assemblage d'un système écrit en FRACTAL. Cette chaîne d'outils peut actuellement être utilisée pour assembler des composants écrits dans des langages de programmation différents (C, C++ et Java). De plus, cette chaîne intègre un canevas de génération d'adaptateurs de communication permettant d'automatiser la création des composants de liaison implantant des protocoles arbitrairement complexes.

Enfin, notons que la chaîne de génération de code mise au point a été intégrée au sein du projet THINK pour remplacer l'outillage existant et est actuellement utilisée dans divers projets de développement.

Mise au point d'un micro-noyau L4 à base de composants

Dans le cadre de nos travaux, nous avons implanté un micro-noyau L4 en utilisant la technologie de programmation proposée. Le but de cette expérimentation était d'identifier les avantages et les inconvénients d'un choix de conception privilégiant les aspects architecturaux par rapport aux aspects liés aux performances. Bien qu'elle soit préliminaire, notre implantation constitue une alternative modulaire aux implantation existantes qui ont la caractéristique d'être monolithiques. Nous avons constaté que l'organisation à base de composants et la disposition d'un outil de génération de code qui assemble le système à partir de la description de son architecture jouaient un rôle essentiel pour la maîtrise d'un tel système. Cette maîtrise est selon nous cruciale pour assurer une bonne qualité d'implantation, surtout concernant des noyaux comme L4 qui sont employés pour garantir des aspects de type sécurité. Enfin, nos analyses ont montré que les pertes de performances observées par rapport aux architectures monolithiques étaient en partie liées au format d'implantation des composants THINK. Cela montre que les performances pourraient être améliorées par le biais de l'intégration de certaines fonctions d'optimisation dans la chaîne de génération de code proposée.

Proposition d'un modèle de programmation pour le développement d'application de streaming réparties

Nos expérimentations dans le cadre de la programmation d'applications multimédia ont montré que la programmation à base de *threads* n'était pas satisfaisante pour la répartition de telles applications sur plusieurs processeurs. En partant de ce constat, nous avons défini un modèle de programmation basé sur le Join Calcul, afin de découpler les aspects fonctionnels des composants des aspects liés à leurs modèles d'exécution et de synchronisation. Ce modèle de programmation est réifié par la définition d'un langage de description de comportements qui est utilisé par la chaîne de génération de code afin de tisser des

contrôleurs d'exécution au niveau de chaque composant. Ainsi, différents types de contrôleurs peuvent êtres générés pour adapter les composants à des environnements d'exécutions variés. L'évaluation de ce modèle de programmation au travers de la mise en œuvre d'un décodeur vidéo a montré que si les performances d'exécution devraient être améliorées pour arriver à une efficacité acceptable, il permettait cependant de tester aisément plusieurs architectures de répartition.

10.2 Perspectives

Les travaux que nous avons effectués au cours de cette thèse ouvrent différentes perspectives. Nous les présentons sous trois volets principaux :

Modèle de composants FRACTAL

Dans le cadre de nos travaux, nous avons utilisé le modèle de composants FRACTAL pour le développement de logiciel à destination des systèmes sur puce. Ceci a apporté, à la communauté FRACTAL, un cadre applicatif non expérimenté jusqu'à présent. Nous estimons que les axes de recherche suivants peuvent êtres poursuivis afin d'améliorer ce modèle.

Implantation de la membrane sous forme de composants Les deux implantations du modèle FRACTAL que nous avons expérimentées au cours de nos travaux, à savoir JULIAet THINK, fournissent la possibilité de sélectionner parmi plusieurs implantations de *membranes*, mais ne permettent pas de créer des membranes via des descriptions d'architectures. Or, nous avons constaté à plusieurs reprises qu'il était très utile de pouvoir spécifier l'architecture d'implantation des interfaces de contrôles. Des travaux récents comme AOKell [SPDC06, MB05], se sont intéressés à ce sujet et ont fourni un tel support. Nous pensons que des directives pour avoir ce type de support devrait être intégrées dans la spécification du modèle FRACTAL.

Révision du modèle de partage FRACTAL propose un modèle de partage de composants. Or, ce modèle manque de précisions sur la manière dont le partage devrait être supporté au niveau des implantations. Nous pensons que ce sujet qui est très intéressant pour la modélisation des ressources partagées d'un système mériterait plus de réflexion, en particulier pour avoir une meilleure maîtrise concernant l'aspect administration des composants partagés (e.g. contrôle de cycle de vie, destruction, etc.).

Définition d'un langage de description d'interfaces À ce jour, le modèle FRACTAL ne spécifie pas de langage de description d'interfaces. Or, la normalisation d'un tel langage nous semble cruciale pour mettre en place une infrastructure de programmation multi-langages. A notre sens, un tel langage de description devrait disposer de trois propriétés principales. Premièrement, comme tout langage de description d'interfaces, ce langage devrait disposer d'un ensemble de constructions de base trouvant une translation directe dans des langages de programmation considérés. Deuxièmement, ce langage devrait être extensible de manière à permettre aux tierces parties de rajouter des constructions pour satisfaire certains besoins spécifiques (e.g. spécification de structures sérialisables, structures de données spécifiques à une architecture matérielle, etc.). Enfin, ce langage devrait disposer des constructions modernes, comme des types génériques de structure de données, ou bien des annotations.

Implantations multi-langages Nous nous sommes intéressés dans le cadre de nos travaux à la mise en place d'une infrastructure de programmation multi-langages en se basant sur le modèle de composants FRACTAL. Nous pensons que cette initiative devrait être poursuivie. En particulier, nous trouvons qu'il serait intéressant de disposer d'une implantation du modèle en .Net/CLI [MR03], car celui-ci constitue un environnement d'intégration naturel pour les programmes écrits dans différents langages. Une telle infrastructure d'implantation devrait supporter des manipulations au niveau bytecode pour permettre des aspects avancés comme la génération dynamique de composants (de façon similaire à ce qui est

actuellement supporté par JULIA), et la compilation tardive vers des formes binaires natives. Des travaux comme ASM [ASM02], Spoon [PNP06] et CTR [FCL06] peuvent constituer des pistes de démarrage intéressantes.

Personnalisation vers d'autres modèles de composants Comme notre analyse le montre, le modèle FRACTAL peut être considéré comme un méta-modèle de composants qui peut être décliné en des personnalités différentes de manière à remplir les spécifications d'autres modèles de composants. Des travaux récents comme [MP06, LSS06], ont analysé comment FRACTAL pouvait être personnalisé pour effectuer des modélisations UML et EMF. Nous pensons que cette base de réflexion devrait être enrichie. Deux pistes de recherche intéressantes consisteraient à créer des personnalités SystemC et OpenMax du modèle FRACTAL. La première peut contribuer à la définition d'un processus de conception conjoint matériel/logiciel en identifiant comment des modèles de composants matériels pourraient être intégrés dans un processus de conception à la FRACTAL. La deuxième permettrait d'identifier comment FRACTAL pourrait être intégré dans un modèle de composants dédié à la modélisation d'une application de type flux de données.

Développement d'outils de traitement d'ADL

Voici un certain nombre d'axes de recherche et de développement qui pourraient bénéficier du canevas logiciel présenté dans l'objectif d'améliorer le processus de développement.

Définition d'un langage d'implantation de composants Actuellement, l'implantation des composants est effectuée dans des langages de programmation classiques. Les langages orientés objets supportant les notions d'interface et d'encapsulation de comportement et de données sont relativement compatibles avec la programmation à base de composants. Ceci n'est pas le cas du langage C dont l'utilisation semble obligatoire dans la programmation des systèmes sur puce. Dans le cadre de nos travaux, nous avons défini un jeu de *macros* pour améliorer l'ergonomie de la programmation en C. Cette approche peut être poussée plus loin, toute en restant dans les limites du langage C. En effet, il nous semble tout à fait possible d'augmenter ce langage avec des constructions de type définition de composante, d'interfaces et d'attributs, et d'effectuer une étape de pré-traitement supportée par le compilateur ADL de manière à traduire le code d'implantation en C standard. Cette approche peut aussi améliorer le niveau des optimisations qui peuvent êtres effectuées en se basant sur les informations architecturales.

Optimisations Actuellement, très peu d'opportunités d'optimisations ont été explorées dans le cadre de l'implantation THINK du modèle FRACTAL. Nous pensons que des niveaux d'optimisations importants peuvent être atteints en utilisant l'outil de génération de code proposé. Parmi les stratégies possibles, citons par exemple la mise à plat d'une architecture à base de composants de manière à éliminer les composants composites à l'exécution et l'élimination ou la réduction des données de contrôle qui servent pour le support d'introspection. Dans le cas des composants destinés à la partie média d'un système sur puce, une stratégie d'optimisation peut être de répartir la partie fonctionnelle du composant sur le processeur média et la partie contrôle sur le processeur hôte. De plus, des techniques d'évaluation partielle peuvent résulter en des gains importants en performance en permettant d'adapter le code d'un composant à son contexte de déploiement. Finalement, des techniques d'optimisation réversible peuvent être employées pour améliorer les performances des composants à l'exécution tout en gardant la possibilité de retrouver les informations architecturales éliminées pour des raisons d'optimisation.

Mise au point d'outils de vérification Le compilateur d'ADL que nous avons développé dans le cadre de nos travaux supporte peu de fonctions de vérification. Un axe de recherche intéressant serait d'y intégrer de nouveaux modules de vérification, aussi bien dynamique que statique. Par exemple, une vérification statique basée sur les types de messages échangés entre les composants, similaire au système de type présenté dans [BLQ+05], permettrait de détecter des assemblages définissant des liaisons connectant des

paires de composants incompatibles au niveau des contenus des messages produits. D'autres modules de vérifications dynamiques peuvent êtres utilisés pour l'analyse comportementale d'un système. Typiquement, les descriptions d'architectures prises en entrée peuvent êtres enrichies avec des informations indiquant les messages produits et admis par les composants, ou bien définissant les protocoles d'interaction entre composants. Ainsi, des modules de traitement à la Wright ou Darwin pourrait être greffés dans le compilateur ADL de manière à effectuer des vérifications de comportement dynamique des composants. Dans ce cadre, nous pensons que l'intégration de l'environnement de modélisation Ptolemy [BHLM94] dans la chaîne de traitement d'ADL constituerait une piste intéressante. En effet, Ptolemy fournit un environnement de modélisation supportant divers modèles de calcul qui peuvent être utilisés pour évaluer le comportement dynamique des composants.

Mise au point d'un support de *debug* et de traçage Dans sa version actuelle, notre infrastructure ne supporte aucune fonction de *debug* dédiée aux aspects composants. Cependant, la programmation à base de composants peut être exploitée pour approfondir les capacités des outils de *debug* et de traçage classiques. Un axe de recherche intéressant est de concevoir un support d'instrumentation à la DTrace [CSL04]. Ce dernier est spécifiquement conçu pour instrumenter des systèmes d'exploitation. Entre autre, il permet de décrire des prédicats et des actions d'instrumentation à l'aide d'un langage dédié, appelé D. Ce langage contient des constructions permettant de concevoir des modules et des sondes pour les systèmes d'exploitation. Nous pensons que le langage D pourrait être étendu ou généralisé pour désigner des composants d'un système en exploitant la présence de ces derniers à l'exécution. Ainsi, des outils de traçage pourraient être conçus pour introspecter les entrée/sorties et l'état interne des composants au cours de l'exécution. Nous pensons que la définition d'une telle extension pourrait être inspirée du langage FScript qui est conçu pour décrire des références de composants dans une architecture hiérarchique à la FRACTAL.

Amélioration de l'ergonomie du compilateur ADL Bien qu'accessoire, nous constatons que l'ergonomie d'utilisation du compilateur ADL est importante pour son utilisation dans un cadre industriel. Dans ce cadre, nous pensons que la chaîne d'outils actuelle pourrait être enrichie avec un générateur de documentation et une interface graphique permettant de visualiser ou de concevoir des architectures à base de composants.

Développement d'applications

Nous identifions ci-dessous des perspectives concernant, non seulement les deux cas d'étude que nous avons présentées dans ce manuscrit, mais aussi de nouveaux cadres applicatifs qui pourront être expérimentés par la suite.

Poursuite du projet THINK4L Notre expérimentation avec le micro-noyau THINK4L nous a montré qu'une grande partie des pertes de performance étaient liées au format d'implantation des composants en THINK, et non aux architectures construites. Dans ce cadre, nous pensons qu'il serait intéressant de réévaluer les performances d'exécution après avoir implanté les optimisations suscitées au sein de l'outil de génération de code. Une autre perspective de recherche est de proposer des mécanismes de gestion des ressources virtuelles (e.g. création-destruction d'IPC et de *threads*) au sein d'un micro-noyau L4 en se basant sur les apports des architectures à base de composants.

Poursuite du projet ThinkJoin Le langage de description de synchronisation que nous avons défini dans le cadre du projet ThinkJoin a montré la faisabilité d'une telle approche et a identifié des gains nets en termes d'exploration d'architectures de parallélisation. Or, le langage défini reste assez primitif. Il pourrait être enrichi avec des constructions de type synchronisation sur une valeur attendue, ou la description des résultats de réaction. De plus, il pourrait être soumis à une formalisation pour identifier des améliorations potentielles. Par exemple, une piste de recherche intéressante serait de vérifier si les

descriptions écrites en JoinDSL peuvent êtres directement traduites dans des constructions de type Join Calculus. Si ceci est le cas, le langage JoinDSL pourrait bénéficier d'outils de vérification mis au point pour Join Calculus. Enfin, des stratégies d'optimisations devraient êtres explorées afin d'améliorer les performances d'exécution écrite à l'aide du canevas logiciel ThinkJoin. Parmi les stratégies d'optimisation qui pourraient être envisagé, citons la génération d'un ordonnancement statique ou bien l'adaptation dynamique de mode de communication inter-composants (i.e. synchrone ou asynchrone) en fonction des charges de calcul subies.

Mise en place d'un intergiciel de déploiement sur puce La programmation à base de composants est compatible avec le développement de systèmes répartis. Une infrastructure logicielle pourrait être mise en place pour simplifier l'exécution répartie des composants. Une telle infrastructure devrait permettre le chargement, l'installation et la liaison dynamique des composants en se basant sur une description d'architecture. Par ce biais, la configuration du logiciel déployé sur une puce pourrait être modifiée à la volée en fonction des applications exécutées à un moment donné. Une partie des mécanismes de base dont devrait disposer une telle infrastructure ont déjà été mis en place dans le cadre de nos travaux. En effet, la génération d'usines de composants permettant de créer et détruire des instances en cours d'exécution, ainsi que le support de génération d'adaptateurs de communication pour assurer une communication transparente entre les composants répartis, constituent l'ensemble minimal des éléments nécessaires. Deux mécanismes pourraient être greffés sur ces derniers. Premièrement un mécanisme de création de liaisons pourrait être mis en place pour automatiser l'instanciation de chaînes de communication complexes (e.g. IPC, RPC, etc.) entre les composants s'exécutant dans des domaines d'exécution différents. Le patron de programmation *export/bind* [KS05] nous semble être parfaitement adapté pour implanter un tel mécanisme. Deuxièmement, un support de reconfiguration dynamique comme celui présenté dans [POS06] pourrait être utilisé afin d'assurer des modifications architecturales sans compromettre l'intégrité du système.

Annexe A

Formats d'implantation des composants dans divers langages de programmation

L'outil de génération de code présenté dans le chapitre 7 supporte actuellement trois langages de programmation. Ce sont le C, le C++, et le Java. Cette annexe est dédiée à la présentation des codes générés pour chacun de ces langages. Dans les sections suivantes, qui sont chacune dédiés à un langage, est d'abord présentée la structure de donnée des implantations de composants, et ce en soulignant les parties qui sont automatiquement générées par la chaine de génération de code présentée dans le chapitre 7. Puis, est décrit le guide de programmation proposé aux programmeurs pour chacun des langages de programmation.

A.1 Implantation en C pour le canevas logiciel THINK

Nos travaux se sont intéressés au canevas THINK à partir de sa deuxième version et ont contribué à la définition de sa troisième version. Cette nouvelle version apporte quelques modifications au modèle binaire des composants afin d'améliorer en particulier la séparation des structures de contrôle et des structures fonctionnelles. De plus, un guide de programmation basé sur des *macros* du langage C a été proposé afin de simplifier l'implantation des composants comparativement au guide préexistant. Bien que notre outillage de génération de code assume aussi l'implantation binaire des composants conformément à la spécification de THINK v2, nous présenterons par la suite uniquement le support de la version 3 en nous focalisant sur les améliorations que nous y avons apporté.

A.1.1 Structure de donnée des composants

L'essentiel du format binaire des composants THINK étant présenté dans la section 4.4, nous reprenons ici uniquement les modifications proposées dans la version 3. Ces modifications ont pour but de séparer totalement les structures de *contenu*, liées aux aspects purement fonctionnels des composants, des structures de *membrane*, liées aux aspects de contrôle. Une telle organisation présente deux avantages. Premièrement, la séparation de ces deux aspects rend l'existence d'une partie du contrôle optionnelle et augmente la flexibilité du format binaire des composants pour permettre l'encapsulation d'un même *contenu* par différents types de *membranes*. Deuxièmement, les deux structures étant entièrement séparées, on peut envisager de répartir le contrôle et les fonctions métier d'un même composant sur deux machines différentes. Ceci est en particulier très utile dans le cadre des systèmes multiprocesseurs sur puce qui constitue notre cadre applicatif privilégié, où l'on peut déployer le contenu d'un composant sur un processeur dédié tout en gardant ses structures de contrôle sur un processeur hôte, dans le but d'optimiser l'empreinte mémoire sur des processeurs dédiés aux ressources souvent très limitées, et améliorer les performances à l'exécution.

La figure A.1 présente la nouvelle structure de donnée des composants THINK dans sa version 3, et met en évidence la distinction entre la partie *contenu* et la partie *membrane* :

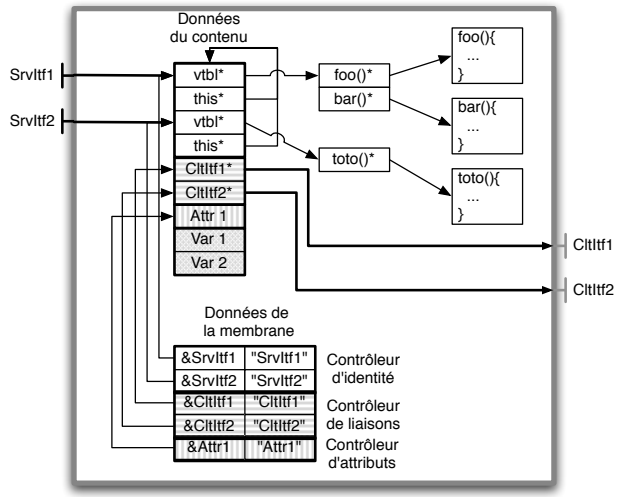


FIG. A.1 – Structure binaire d'implantation des composants FRACTAL en THINK.

Le contenu du composant est constitué de ses interfaces, de ses attributs, de ses données privées et de l'implantation de ses interfaces serveurs. Une structure de donnée est spécifiquement générée pour regrouper ces éléments[1]. La partie en blanc de la figure A.1 représente les structures dédiées aux interfaces serveurs. Ceci comprend les descripteurs d'interfaces (i.e. le couple de pointeurs vers les tables de fonctions virtuelles et la structure de donnée du composant) qui pointent vers les tables de fonctions implantées par le composant. Notons que si un composant implante des interfaces de contrôle, les descripteurs de ces interfaces se trouveront à ce niveau car ces descripteurs sont des éléments fonctionnels. Ensuite se situent les champs contenant les interfaces clients du composants, suivis par les champs contenant des valeurs des attributs. Enfin, la structure qui permet de stocker les champs privés du composant est agrégée à la structure de contenu. L'ensemble de ces structures forment le minimum nécessaire pour l'implantation d'un composant qui ne fournirait aucun mécanisme de contrôle.

La membrane est implantée par des structures de données qui sont propres aux contrôleurs. Par conséquent, la structure générée peut prendre des formes très différentes. Nous présentons dans la partie grise foncée de la figure A.1 un exemple d'implantation de structure de *membrane*. Dans cet exemple, le contrôleur d'attributs qui fournit les opérations de mise-à-jour et de consultation de ceux-ci utilise des couples $nom, pointeur$ pour associer des valeurs textuelles aux champs d'attributs présents au niveau de la structure de *contenu* vue précédemment. Un contrôleur de structure similaire est utilisé pour désigner les interfaces clients du composant. Remarquons que le pointeur `this` d'une structure d'interface de contrôle ne pointe pas obligatoirement sur la structure de donnée privée d'un composant, mais peut aussi pointer vers des structures de contrôle.

L'organisation des structures de données pour les *composants composites* est similaire à celle qui est présentée pour les *composants primitifs*. Étant donné que le contenu d'un *composite* est fourni par ses sous-composants, ce dernier se voit associer uniquement du code et des structures de données formant sa membrane. Ces éléments pouvant être très variés en fonction des interfaces de contrôle fournies par chaque *composite*, nous ne rentrerons pas ici dans plus de détails. Notons simplement que l'organisation de la membrane décrite ci-dessus s'applique bien à l'implantation de la membrane des *composites*.

[1]Conformément à la version 3 du canevas THINK, seul les fonctions d'implantation des interfaces serveurs sont fournies dans le fichier d'implantation. Le reste de la structure présentée est généré par l'outillage ADL.

A.1.2 Guide de programmation

Dans le chapitre traitant du modèle de composants FRACTAL, la figure 4.14 donnait un extrait de code d'implantation des composant en THINK v2. Nous avons fait le constat que ce guide de programmation dédié à la programmation des composants primitif était difficile à mettre en œuvre en raison de nombreuses conventions de nommage à respecter, et de la nécessité d'initialiser explicitement les structures de données. Il est apparu que ces éléments étaient assez systématiques pour être cachés derrières des *macros* spécialisées en fonction de l'architecture des composants. L'intérêt du guide de programmation que nous présentons dans cette sous-section est donc de proposer un tel guide de programmation, simplifiant considérablement la mise en œuvre des composants primitifs. La suite de cette sous-section est dédiée à la description de ce guide de programmation, et pourra s'avérer utile au programmeur usager de l'infrastructure, ou au lecteur curieux de mieux saisir les détails de ce guide. Elle commence par donner la structure générale d'un fichier d'implantation, puis énumère l'ensemble très réduit des *macros* qui peuvent être utilisées pour implanter des composants primitifs.

Fichier d'implantation Un fichier d'implantation correspond à l'implantation d'un composant (i.e. classes en Java). Le contenu du fichier doit respecter la structure donnée ci-dessous :

IMPLANTATION : := *DECLAREDONNEE* "#include <think.h>" (*METHODE*)*

La partie *DECLAREDONNEE* est dédiée à la déclaration des données privées du composant. Elle est suivie de l'inclusion du fichier entête qui définit certaines *macros* utilisées dans l'implantation des méthodes. Le reste du fichier est constitué des définitions de méthodes pour l'implantation des interfaces serveur du composant.

Déclaration des donnée d'instance privée La déclaration des données d'instance privées est effectuée sous la forme d'une liste de variables en utilisant la *macro* `DECLARE_DATA` et en suivant la structure ci-dessous :

DECLAREDONNEE : := "`DECLARE_DATA {`" (déclaration de champs)* "`};`"

La valeur du *macro* `DECLARE_DATA` est définie par le support ADL en fonction du nom de composant. En pratique, à cette *macro* sera substituée une définition de structure dont le nom est celui du composant suffixé du mot `_private_data`.

Implantation de méthodes Un fichier d'implantation doit contenir l'implantation de l'ensemble des méthodes faisant partie des interfaces fournies par le composant. Deux règles sont à respecter : (1) les méthodes doivent être déclarées en utilisant la *macro* `METHOD` qui prend en paramètre le nom de la méthode et celui de l'interface à laquelle elle appartient, et (2) le premier paramètre formel de la méthode doit être un pointeur de type `void` appelé `_this` pour récupérer le contexte du composant. Cette dernière règle correspond à la manière dont les objets C++ peuvent être implanté en C. La structure d'une implantation de méthode doit donc être ainsi construite :

METHODE : := typeDeRetour "`METHOD(`" nomInterface "`,`" nomMéthode "`)`"
"`(void *_this`" ("`,`" paramètreFormel)* "`)`"
"`{`"
code d'implantation.
"`}`"

La valeur de la *macro* `METHOD` est calculée par le générateur de code en fonction du *type de composant*, du *nom de l'interface* et du *nom de la méthode*. La valeur qui se substitue à cette *macro* est une agrégation de ces trois termes.

Invocation d'interface client La *macro* `CALL` permet d'invoquer les interfaces clients du composant. Elle prend en paramètre la référence de l'interface client et le nom de la méthode à invoquer ainsi que

l'ensemble des paramètres formels à passer.

INVOCATION : := "CALL (" nomInterface "," nomMéthode ("," paramètreFormel)* ") ;"

La valeur de la *macro CALL* est fixe et est définie comme ci-dessous dans le fichier entête `think.h`.

```
#define CALL(itf, meth, args...) \
        itf->vtbl->meth(itf->this, args...)
```

Accès aux données d'instance privées La *macro* `DATA` donne accès aux données d'instance privées du composant. Elle est utilisée en respectant la forme ci-dessous :

ACCESDONNEE : := "DATA." nomDeChamps

La valeur de la *macro* `DATA` permet de *caster* la structure pointée par l'argument `_this` de la méthode et ainsi d'accéder à son champs `private_data` afin de donner un accès direct aux données d'instance du composant. Le nom de la structure qui est utilisée pour l'opération de *cast* est naturellement fonction du nom du composant, et est calculée par la chaîne de compilaiton ADL.

Accès aux interfaces clients La *macro* `REQUIRED` permet d'accéder à la référence d'une interface client. Son utilisation est similaire à la *macro* `DATA`.

ACCESITF : := "REQUIRED." nomInterface

De façon similaire à la *macro* `DATA`, la *macro* `REQUIRED` permet d'accéder aux champs contenant les références aux interface clients du composant en *castant* le paramètre `_this` de la méthode. Le nom de la structure qui est utilisée pour l'opération de *cast* est dépend du nom du composant, et est calculée de la même façon que la macro `DATA`.

Accès aux attributs La *macro* `ATTRIBUTES` donne accès aux attributs du composant. Son utilisation et sa valeur sont équivalentes aux deux *macros* précedemment décrites.

ACCESATTR : := "ATTRIBUTES." nomDeChamps

La figure A.2 illustre la mise en œuvre d'un composant de fonction arithmétique identique à celui présenté précédemment dans la figure 4.14. Remarquons que l'interface client `console` du composant n'est pas déclarée par le programmeur. Elle est en effet déclarée par la chaîne de compilation d'ADL et est accessible via la *macro* `REQUIRED`. Les méthodes implantées sont déclarées à l'aide de macro `METHOD`. Ainsi, comparativement à la version Think-V2, le programmeur n'a plus à initialiser les table de fonctions virtuelles, puisqu'elles sont générées par la chaîne de compilation.

```
// Données d'instance
DECLARE_DATA {
  int nbrInvocation ; // Attribut
};

// Implantation des fonctions
int METHOD(arithmetic,add) (void *_this, int a, int b) {
  CALL(REQUIRED.console, afficheEntier, DATA.nbrInvocation ++) ;
  return a+b ;
}

float METHOD(arithmetic, div) (void *_this, float a, float b) {
  CALL(REQUIRED.console, afficheEntier, DATA.nbrInvocation ++) ;
  return a/b ;
}
```

FIG. A.2 – Implantation en THINK v3 du même composant que celui présenté dans la figure 4.14.

La figure A.3 reprend un extrait du fichier d'implantation présenté dans la figure A.2 après la phase de *pré-traitement*. Sont supprimées dans cet extrait toutes les structures liées à la membrane du composant par souci de simplification. Nous constatons que la structure de définition de données d'instance du composant est renommé en fonction du nom de composant. De même, la structure de donnée du contenu de composant est générée conformément au modèle d'implantation présenté dans la figure A.1. Le pointeur `_this`, donné en tant que premier argument des méthodes d'implantation, pointe vers l'instance d'une telle structure qui est associée au composant récepteur. Enfin les *macros* `CALL`, `REQUIRED` et `DATA` utilisées dans l'implantation originale sont transformées en des opérations d'accès conformes à la structure du contenu du composant.

```
// Données d'instance
struct {
  int nbrInvocation ; // Attribut
} ArithmeticComp_private_data;

// Déclaration de la structure de contenu
struct {
  struct ArithmeticComp_server_itfs  server_itfs  ;
  struct ArithmeticComp_client_itfs  client_itfs  ;
  struct ArithmeticComp_private_data private_data ;
} ArithmeticComp_data ;

// Implantation des fonctions
int ArithmeticComp_arithmetic_add_method (void *_this, int a, int b) {
  ((struct ArithmeticComp_data*)_this)->client_itfs.console->vtbl->afficheEntier(
      ((struct ArithmeticComp_data*)_this)->client_itfs.console->this,
      (((struct ArithmeticComp_data*)_this)->private_data.nbrInvocation) ++
    ) ;
  return a+b ;
}

float ArithmeticComp_arithmetic_div_method (void *_this, float a, float b) {
  ((struct ArithmeticComp_data*)_this)->client_itfs.console->vtbl->afficheEntier(
      ((struct ArithmeticComp_data*)_this)->client_itfs.console->this,
      (((struct ArithmeticComp_data*)_this)->private_data.nbrInvocation) ++
    ) ;
  return a/b ;
}
```

FIG. A.3 – Extrait du code d'implantation après la phase de *pré-traitement*.

A.2 Implantation en C++

Nous avons implanté dans le cadre de nos travaux une spécialisation de la chaîne de génération de code pour la programmation des composants en C++. Les différences principales de notre approche par rapport à celle qui était déjà proposée dans le cadre des travaux présenté dans [Lay05] sont au nombre de deux. Premièrement, nous visons la proposition d'un guide de programmation qui bénéficie pleinement de la disponibilité d'un outil de génération de code basé sur le traitement de l'ADL afin de minimiser la quantité de code écrit par les programmeurs. Deuxièmement, nous voulons exploiter la proximité des langages C et C++ en permettant l'invocation transparente ou quasi transparente de méthodes entre des composants écrits en C et en C++, le but étant de permettre l'utilisation simultanée simple de ces deux langages dans la mise en œuvre des applications à base de composants. Dans la suite de cette sous-section, nous présentons la structure de donnée des composants ainsi que le guide de programmation proposé. Ensuite, nous décrivons le code généré par le support ADL.

A.2.1 Structure de donnée des composants

Nous nous proposons d'analyser les structures de données utilisée pour implanter des composant en deux temps. D'abord, nous expliquons notre choix d'implantation des interface après avoir discuté des différentes alternatives qui s'offrent à nous ; puis nous décrivons comment sont implantées les parties fonctionnelle et contrôle des composants.

Format binaire des interfaces
Le langage C++ ne propose pas de construction adéquate pour la mise en œuvre des interfaces, contrairement au langage Java. Cependant, l'utilisation des fonctions virtuelles, i.e. la définition de fonctions sans implantation, apparaît propice à une telle mise en oeuvre, les *interfaces* Java pouvant être vues comme des *classes* sans implantation. C'est donc le choix vers lequel nous nous sommes tournés pour implanter les interfaces de composants en utilisant des classes purement virtuelles, i.e. des classes contenant uniquement des définitions de fonctions virtuelles. La compilation de telles classes virtuelles résulte en effet en des représentation binaires des tables de fonctions similaires à celles représentées dans la figure 4.11.

Une fois le choix d'implantation pour les interfaces fixé, se pose la question du format binaire de descripteurs d'interface. Plus particulièrement, il s'agit de trouver une solution au changement de contexte des composants (i.e. passage de pointeur `this` en premier paramètre de l'appel de fonction) au moment de l'invocation d'une méthode. Dans une solution d'implantation restreinte au langage C++, ce problème eut été pris en compte automatiquement par le compilateur C++ de par la notion d'objet. En l'occurence, la volonté de mélanger des composants écrits en C et en C++ requiert la définition d'un format binaire pour les descripteurs d'interfaces qui permette de réaliser le changement de contexte des composants manuellement afin de permettre aux composants C d'invoquer des méthodes d'interfaces implantées dans des composants C++.

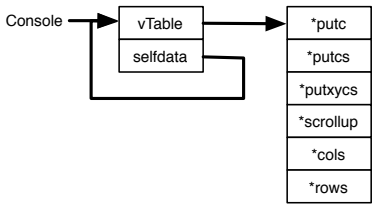
Afin de résoudre ce problème, nous avons adopté comme format de descripteur d'interfacs le format binaire des composants THINK tel que présenté dans la figure 4.11. La figure A.4 illustre la mise en œuvre de cette structure en C++. Une interface de composant est implantée à travers une classe purement virtuelle dont le premier champ est un pointeur qui réserve un mot pour le stockage de l'adresse du contexte du composant. Ce champ est initialisé à la construction de la classe implantant les méthodes de cette classe virtuelle. A droite de la figure A.4 est illustré le résultat de la compilation d'une telle classe virtuelle. Remarquons que le résultat est quasiment identique au format présenté dans la figure 4.11 à l'exception de l'endroit pointé par le pointeur `selfdata`. En effet, ce pointeur est sensé pointer vers les données d'instances des composants, mais dans le cas de C++, il pointe vers l'interface elle même et non pas vers sa classe d'implantation. Ceci est correct, et résulte en une exécution normale, car le compilateur C++ insère par défaut des *thunks* au niveau des pointeurs de fonctions virtuelles qui permettent de réajuster la case pointé en premier paramètre de l'appel de fonction (i.e la valeur de `selfdata`) avec une valeur fixe qui correspond à la distance du pointeur à la classe d'implantation de l'interface.

Une dernière convention reste à fixer pour finaliser la définition des interfaces. Il s'agit de la convention d'appel. Deux alternatives : l'invocation classique d'un objet en C++ ou l'usage de la *macro* `CALL` employée dans THINK. Le tableau A.1 compare ces deux possibilités d'invocation vis-à-vis de leur compatibilité avec les deux langages d'implantations. La convention d'appel du langage C++ pose en particulier des problèmes lorsque la partie serveur (i.e. la méthode réceptrice) est écrite en C. En effet, les composants implantés en C nécessitent le passage en premier paramètre du pointeur `selfdata` enregistré dans le descripteur d'interface. Or, la permutation de contexte est effectuée d'une manière différente par les compilateurs C++.

C'est essentiellement pour cette raison que nous suggérons l'utilisation systématique de la *macro* CALL y compris dans le cas de la programmation des composants en C++ afin d'obtenir une transparence

```
namespace video{
  namespace api{
    class Console{
      public:
      void *selfdata ;
      virtual void putc(char c) = 0 ;
      virtual void putcs(char* str) = 0
      virtual void putxycs(int x, int y,
                           char* str) =
      virtual void scrollup() = 0 ;
      virtual jint cols() = 0 ;
      virtual jint rows() = 0 ;
      Console(){
        this->selfdata = this ;
      }
    };
  };
};
```

(a)



(b)

FIG. A.4 – a) Définition d'une interface de composant en utilisant des classes purement virtuelles, et b) représentation binaire résultante.

complète vis-à-vis des 2 langages d'implantation. Neanmoins, les programmeurs sont libres d'utiliser l'invocation classique de méthodes en C++, au prix de s'assurer que le composant serveur ne soit pas implanté en C.

Client	**Serveur**	itf → meth(...)	CALL(itf, meth, ...)
C	**C**	-	√
C	**C++**	-	√
C++	**C**	×	√
C++	**C++**	√	√

TAB. A.1 – Comparaison des deux méthodes d'invocations en terme d'interopérabilité entre les composants écrits en C et en C++.

Structure des composants

Un composant FRACTAL est implanté en C++ par un objet qui hérite de plusieurs autres afin d'implanter les aspects fonctionnels et les aspects de contrôles séparément. Le modèle d'héritage qui donne lieu à l'implantation d'un composant est présenté dans la figure A.5. Dans la partie en haut à droite de la figure, nous distinguons le type fonctionnel d'un composant. Il s'agit d'une classe qui hérite de l'ensemble des classes virtuelles représentant les interfaces serveurs fonctionnelles du composant. Cette classe définit aussi les champs nécessaires à l'enregistrement des interfaces clients et des valeurs des attributs. Dans la partie gauche haute de la figure sont représentées les classes d'implantation des interfaces de contrôles fournies par le composant, autrement appelée *membrane* du composant. Chacune de ces classes peut définir des structures de données propres aux opérations qu'elles fournissent.

Au centre de la figure A.5, la classe *component type* définit le type complet du composant. Il s'agit d'une classe qui hérite du type fonctionnel et de l'ensemble des classes d'implantation de la *membrane*. Cette classe implante ainsi un constructeur sans paramètres qui a pour tâche d'invoquer les constructeurs de chacune des classes de la *membrane* afin d'initialiser leurs structures de données. Typiquement, les adresses des enregistrements des interfaces clients et des attributs seront communiquées à ces classes

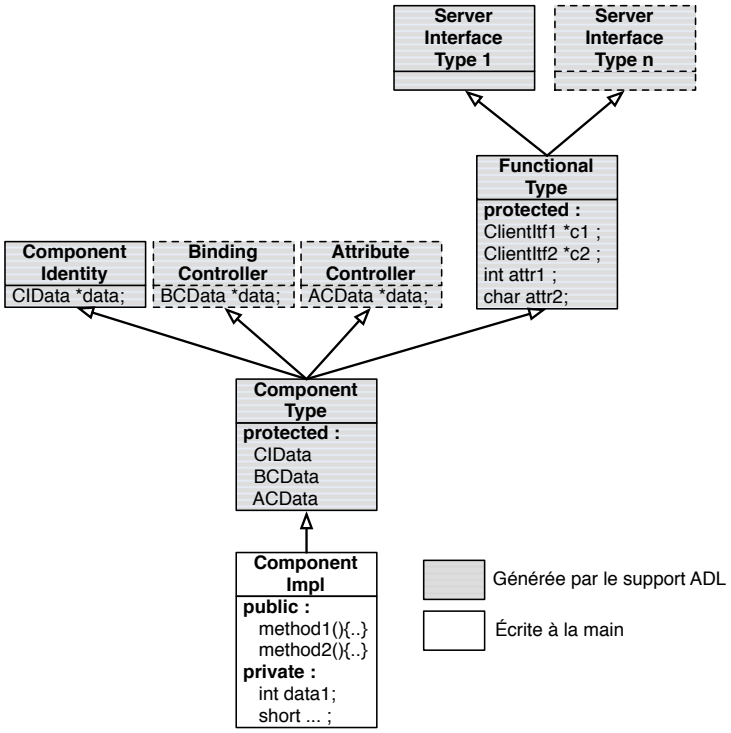


FIG. A.5 – Modèle d'héritage utilisé pour l'implantation des composants FRACTAL en C++.

pour qu'elles puissent créer des associations similaires à celles que nous avons décrites dans le cadre de THINK. La raison pour laquelle les champs d'interface clients et d'attributs sont déclarés en mode protégé et non pas en mode privé est précisément pour permettre aux classes de la *membrane* de manipuler leurs contenues.

Enfin, en bas de la figure, une dernière classe *Component impl* hérite du type du composant et implante son contenu. Il s'agit de la classe d'implantation des interfaces fonctionnelles. Cette classe peut déclarer de plus des champs privés afin d'utiliser des variables d'instances dans l'implantation du composant. Un avantage intéressant de ce modèle d'héritage est qu'il permet aux programmeurs de surcharger le comportement implanté par la membrane pour implanter des fonctions de contrôles spécifiques. Pour cela, il suffit de définir les fonctions que l'on veut surcharger dans la classe d'implantation. Il est ainsi possible d'intercepter la *membrane* depuis la classe d'implantation afin d'effectuer des pré- et des post-traitements.

A.2.2 Guide de programmation

Grâce aux notions objets présentes dans le langage C++, le guide de programmation proposé pour la mise en œuvre des composants est assez simple, et se réduit à deux éléments. Premièrement, la classe d'implantation implantée par le programmeur est supposée hériter de la classe définissant le type du composant. Ceci permet de bénéficier de la génération par la chaîne de traitement de l'ADL des classes d'implantation de la *membrane* et des autres structures de données qui sont spécifiées ci-dessus. Cependant, les programmeurs sont libres de ne pas suivre cette suggestion si la structure générée ne leur convient pas. Dans ce cas, ils peuvent mettre en œuvre des implantations spécifiques à leur besoin, à condition qu'ils respectent le format binaire des interfaces tels qu'il est défini dans la sous-section précédente. Deuxièmement, l'utilisation de la *macro* `CALL` est préconisée afin d'obtenir une transparence

complète vis-à-vis du langage d'implantation du composant serveur. Néanmoins, comme expliqué précédemment, les programmeurs sont libres d'utiliser d'autres procédés s'ils maîtrisent l'implantation du composant serveur.

La figure A.6 illustre l'implantation en C++ du composant d'opération arithmétique, déjà considéré dans le cadre de THINK. Cette classe porte le nom qui a été déclaré dans la description ADL comme étant le nom de la classe d'implantation. Elle hérite d'une classe portant le nom de cette classe suffixé du mot `Type`. Remarquons que la classe d'implantation peut fournir un constructeur et un destructeur mais ces derniers doivent être sans paramètres pour que la chaîne de génération puisse instancier des composants de manière générique. Cela sera par exemple utile pour la gestion sophistiquée du cycle de vie du composant, qui est sensée être assurée par une interface spécifique de type *life cycle controller* selon la spécification du modèle FRACTAL. L'implantation des méthodes des interfaces serveur se fait de la même manière qu'un objet C++. L'interface client `console` est utilisé sans être déclarée sachant qu'elle est déclarée dans la classe générée qui définit le type du composant. Enfin, le champs `nbrInvocation` est déclaré et utilisé comme un variable d'instance normale d'un objet C++.

```
class ComposantArithmetic : public ComposantArithmeticType {
  public :
    ComposantArithmetic () {...};
    ComposantArithmetic () {...};

    // Implantation des fonctions
    int add (int a, int b) {
      CALL(this->console, afficheEntier, this->nbrInvocation ++) ;
      return a+b ;
    }

    float div (float a, float b) {
      CALL(this->console, afficheEntier, this->nbrInvocation ++) ;
      return a/b ;
    }

  private :
    int nbrInvocation ;
} ;
```

FIG. A.6 – Implantation en C++ du même composant que celui présenté dans la figure 4.14.

A.3 Implantation en Java

Dans le cadre de nos travaux liés à la personalisation de la chaîne de traitement d'ADL pour la génération de code, nous avons mis au point une nouvelle implantation du modèle de composants FRACTAL en Java qui suit une logique différente de celles qui sont proposées précédemment par la communauté. La motivation qui a guidé cet effort de preuve de concept a été d'utiliser au maximum les facilitées apportées par la chaîne de traitement d'ADL afin de simplifier la programmation des composants primitifs, et ce en fournissant une génération automatique des types de composants et de l'implantation des membranes. En particulier, grâce à la génération de membrane spécialisées pour chaque composants, ces derniers n'ont plus de dépendances envers des libraries *runtime* ni vers des transformateurs dynamiques de bytecode comme c'est le cas avec les implantations Julia et AOkell du modèle FRACTAL. Ainsi, l'implantation des composants devient complètement statique, ce qui rend possible de compiler les composants en binaire natif afin d'améliorer les performances à l'exécution.

A.3.1 Structure de donnée des composants

L'organisation de la structure de donnée des composants en Java est fortement inspirée de l'implantation en C++, présentée dans la section précédente. L'interdiction de l'héritage multiple en Java nous a néanmoins amené à repenser le modèle d'héritage. En effet, l'implantation C++ bénéficie de cette propriété au niveau de la classe du type de composant qui hérite de l'ensemble des classes d'implantation de la *membrane* et du type fonctionnel du composant. Grâce à cette organisation en C++, la classe d'implantation peut supporter l'ensemble de ces propriétés architecturales simplement en héritant de son type. Étant donné que ce modèle n'est pas applicable en Java, nous avons inversé le modèle d'héritage afin d'obtenir une implantation quasiment équivalente au niveau des propriétés tout en restant compatible avec la contrainte d'héritage unique de Java.

La figure A.7 présente le modèle d'héritage appliqué dans notre implantation du modèle FRACTAL en Java. Sur le haut de la figure, la classe *Functional Type* définit le type fonctionnel du composant. Cette classe, générée par le support ADL, implante l'ensemble des interfaces serveurs fonctionnelles du composant et définit des champs protégés pour enregistrer les références des interfaces clients et les valeurs des attributs. Notons que la construction `interface` du langage Java est utilisée pour définir les interfaces[2]. La classe d'implantation peut hériter de cette classe pour bénéficier de ces définitions. Soulignons que l'héritage de cette classe générée par la classe d'implantation est pratique mais optionnel car cela peut amener certain restrictions. Par exemple, si la classe d'implantation hérite de la classe de type, elle n'a plus la possibilité d'hériter d'une autre classe générique d'implantation puisque Java ne permet pas l'héritage multiple. De plus, la classe de type étant générée par la chaîne de traitement, elle ne sera pas forcément présente au moment de la mise en œuvre de la classe d'implantation. Ceci peut avoir des conséquences au niveau de l'aide à la programmation fournie par des atelier logiciels de type Eclipse qui ont besoin d'interpréter la classe héritée. Dans le cas ou ces limitations gêneraient la mise en œuvre de la classe d'implantation, les programmeurs sont invités à définir eux même les aspects liés au type fonctionnel du composant au sein de la classe d'implantation en respectant les conventions de nommage figés par rapport à la description ADL du composant. Il peuvent aussi effectuer une passe préliminaire de génération de code basée sur des implantations vides.

Dans l'implantation Java, contrairement à l'implantation C++, c'est la classe de définition de composant qui hérite de la classe d'implantation. Ce modèle d'héritage permet de faire bénéficier librement la classe d'implantation de l'héritage. De plus, comme la classe d'implantation n'a pas de dépendance vers la classe de définition de composant qui est générée, le problème potentiel lié aux ateliers logiciels, mentionnés ci-dessus dans le cas de la classe générée *Functional Type* n'est pas reproduit. La classe de définition de composant hérite alors des propriétés de la classe d'implantation et implante elle-même les interfaces de contrôle ou *membrane* générée par les tisseurs de *membrane* lors de la génération de code. Remarquons que le fait de générer des méthodes de contrôle en connaissant les propriétés architecturales du composant permet de générer des implantations spécialisées de *membrane*. Ainsi, la chaîne de génération de code fournit l'implantation de certains contrôleurs comme *Binding Controller* ou *AttributeController*, ceci de façon comparable à AOKell mais purement statiquement, contrôleur qui sont par contre obligatoirement écrits à la main par les programmeurs dans le canevas JULIA.

Comparativement à la spécialisation de la génération de code vers le langage C++, la surcharge des méthodes de contrôle dans la classe d'implantation nécessite ici un mécanisme supplémentaire en raison de l'inversion du graphe d'héritage entre la classe d'implantation de la *membrane* et la classe d'implantation du composant. En l'occurence, les tisseurs de code de contrôle analysent la classe d'implantation préalablement chargée à l'aide de la réflexivité native du langage java afin de savoir si certains méthodes de contrôles sont implantées dans la classe d'implantation. Des implantations de méthodes de contrôle

[2]La définition des interfaces peut être soit écrite à la main en Java, soit générée à partir des fichiers IDL par le support ADL

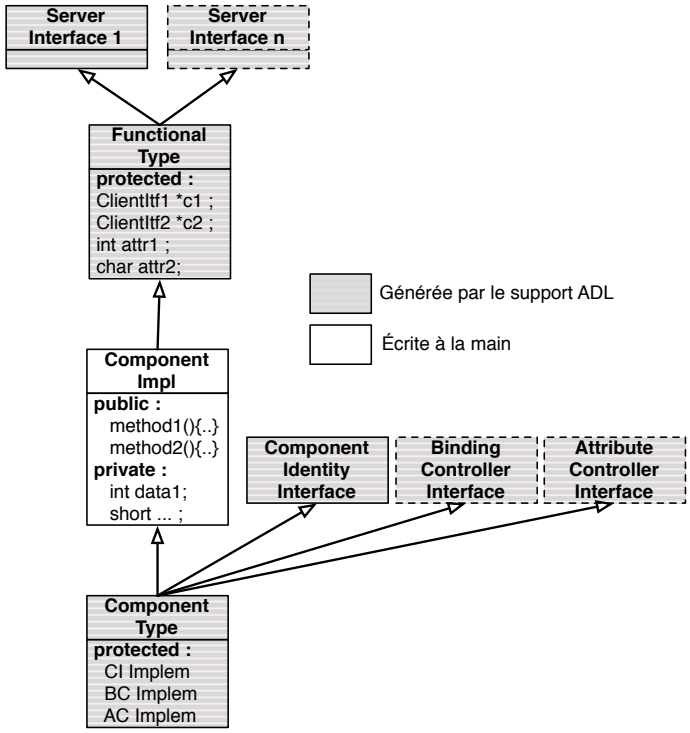


FIG. A.7 – Modèle d'héritage utilisé pour l'implantation des composants FRACTAL en Java.

spécifiques sont alors générées au niveau de la membrane, et ont pour rôle de faire suivre les invocations vers les implantations fournies par le programmeur. Ainsi, la surcharge des méthodes de contrôles est supportée mais par délégation explicite.

Enfin, notons que le modèle d'implantation présenté reste plus limité en termes de fonctionnalités et d'efficacité par rapport aux autres modèles comme AOKell ou JULIA. En revanche, ce travail a constitué une preuve de concept pour illustrer comment les composants FRACTAL peuvent être implantés en utilisant purement le langage Java, c'est à dire sans bénéficier des outils supplémentaires comme des tisseurs d'aspects ou des manipulateurs de bytecode. Cela permet entre autres de compiler les composants en binaire natif vers une plate-forme matérielle donnée en utilisant un compilateur de type Excelsior [Exc] ou GCJ [GCJ].

A.3.2 Guide de programmation

La figure A.8 illustre la classe d'implantation écrite par un programmeur pour mettre en œuvre en Java le composant primitif fournissant des opérations arithmétiques. Cette classe porte le nom de la classe d'implantation mentionnée dans la description ADL du composant. Dans ce cas d'exemple, la classe d'implantation n'hérite pas de la classe de type fonctionnel du composant, qui est définie directement dans la classe d'implantation. Cette classe explicite donc le fait qu'elle implante l'interface `Arithmetic` et déclare un champ pour enregistrer l'interface client `console`. La gestion des liaisons, i.e. l'interface `BindingController`, n'est pas implantée à ce niveau, et sera fournie dans la classe de définition qui héritera de celle-ci. Pour cette raison, les champs des interfaces clients déclarées à ce niveau ainsi que ceux des attributs doivent être des variables protégés (i.e. accesible par les fils qui héritent de cette classe) et doivent obligatoirement porter les noms qui ont étés spécifiés dans la description ADL.

```
public class ArithmeticImpl implements Arithmetic  {
  // Définition de l'interface serveur "console"
  protected video.api.Console console;
  // Définition d'une variable d'instance
  private nbrInvocation ;

  // Implantation des méthodes de l'interface Arithmetic
  int add (int a, int b) {
    console.afficheEntier(this.nbrInvocation++) ;
    return a+b ;
  }

  float div (float a, float b) {
    console.afficheEntier(this.nbrInvocation++) ;
    return a/b ;
  }
}
```

FIG. A.8 – Implantation en Java du même composant que celui présenté dans la figure 4.14.

Bibliographie

[ACN02a] Jonathan Aldrich, Craig Chambers, and David Notkin, *Architectural reasoning in archjava.*, ECOOP (Boris Magnusson, ed.), Lecture Notes in Computer Science, vol. 2374, Springer, 2002, pp. 334–367.

[ACN02b] ———, *Archjava : connecting software architecture to implementation*, ICSE '02 : Proceedings of the 24th International Conference on Software Engineering (New York, NY, USA), ACM Press, 2002, pp. 187–197.

[AG98] Robert Allen and David Garlan, *Errata : A formal basis for architectural connection.*, ACM Trans. Softw. Eng. Methodol. **7** (1998), no. 3, 333–334.

[AGD97] R. Allen, D. Garlan, and R. Douence, *Specifying Dynamism in Software Architectures*, Proceedings of the Workshop on Foundations of Component-Based Software Engineering (Zurich, Switzerland), September 1997.

[AL03] Andrei Alexandrescu and Konrad Lorincz, *ArchJava : An Evaluation*, Tech. report, University of Washington, 2003.

[ALZ00] Davide Ancona, Giovanni Lagorio, and Elena Zucca, *Jam — A smooth extension of Java with mixins*, Lecture Notes in Computer Science **1850** (2000), 154+.

[Ant] *Ant Apache web site*, http ://ant.apache.org.

[AOK] *AOKEll web site*, http ://fractal.objectweb.org/tutorials/aokell/index.html.

[Arc] *ArcStyler*, http ://www.io-software.com.

[Arg] *ArgoUML*, http ://www.argouml.tigris.org.

[ASCN03] J. Aldrich, V. Sazawal, C. Chambers, and David Notkin, *Language Support for Connector Abstractions*, Proceedings 17th European Conference on Object-Oriented Programming (ECOOP), 2003.

[ASM02] *ASM : a Java Byte-Code Manipulation Framework*, 2002.

[BBB+98] R. Balter, L. Bellissard, F. Boyer, M. Riveill, and J.Y. Vion-Dury, *Architecturing and Configuring Distributed Applications with Olan*, Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98) (The Lake District, UK), September 1998.

[BBH+05] Sara Bouchenak, Fabienne Boyer, Daniel Hagimont, Sacha Krakowiak, Adrian Mos, Noel de Palma, Vivien Quema, and Jean-Bernard Stefani, *Architecture-based autonomous repair management : An application to j2ee clusters*, The 24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005) (Orlando, FL, USA), Oct 2005.

[BCE+03] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, *The synchronous languages twelve years later*, Proc. of the IEEE, Special issue on embedded systems **91** (2003), no. 1, 64–83.

[BCF04] Nick Benton, Luca Cardelli, and Cédric Fournet, *Modern concurrency abstractions for C#*, ACM Trans. Program. Lang. Syst. **26** (2004), no. 5, 769–804.

[BCL+04] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani, *An Open Component Model and its Support in Java*, Proceedings of the International Symposium on Component-based Software Engineering (CBSE'2004) (Edinburgh, Scotland), 2004.

[BCL+06] E. Bruneton, T. Coupaye, M. Leclercq, V. Qu éma, and J.-B. Stefani, *The Fractal Component Model and its Support in Java*, Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems **36** (2006), no. 11-12, 1257–1284.

[BCM03] F. Baude, D. Caromel, and M. Morel, *From distributed objects to hierarchical grid components*, 2003.

[BCP+05] Aimen Bouchhima, Xi Chen, Frédéric Pétrot, Wander O. Cesário, and Ahmed Amine Jerraya, *A unified HW/SW interface model to remove discontinuities between HW and SW design*, EMSOFT, 2005, pp. 159–163.

[BCS03] E. Bruneton, T. Coupaye, and J.B. Stefani, *The Fractal Component Model, v2*, 2003.

[BdS96] Frederic Boussinot and Robert de Simone, *The SL Synchronous Language*, IEEE Transactions on Software Engineering **22** (1996), no. 4, 256–266.

[Bel] Luc Bellissard, *From distributed objects to distributed components : the olan approach.*

[Bel97] ———, *Construction, configuration et administration d'applications réparties*, Ph.D. thesis, Institut National Poytechnique de Grenoble, 1997.

[BFS97] Angelos Bilas, Jason Fritts, and Jaswinder Pal Singh, *Real-time parallel MPEG-2 decoding in software*, IPPS '97 : Proceedings of the 11th International Symposium on Parallel Processing (Washington, DC, USA), IEEE Computer Society, 1997, pp. 197–203.

[BG01] Jean Bézivin and Olivier Gerbé, *Towards a Precise Definition of the OMG/MDA Framework*, ASE '01 : Proceedings of the 16th IEEE international conference on Automated software engineering (Washington, DC, USA), IEEE Computer Society, 2001, p. 273.

[BGS04] Xavier Blanc, Marie-Pierre Gervais, and Prawee Sriplakich, *Model bus : Towards the interoperability of modelling tools.*, MDAFA, 2004, pp. 17–32.

[BHA+05] A. Baumann, G. Heiser, J. Appavoo, D. Silva, O. Krieger, R. Wisniewski, and J. Kerr, *Providing dynamic update in an operating system*, 2005.

[BHLM94] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt, *Ptolemy : A framework for simulating and prototyping heterogenous systems.*, Int. Journal in Computer Simulation **4** (1994), no. 2, 0–.

[BLQ+05] Philippe Bidinger, Matthieu Leclercq, Vivien Quéma, Alan Schmitt, and Jean-Bernard Stefani, *Dream Types - A Domain Specific Type System for Component-Based Message-Oriented Middleware*, 4th Workshop on Specification and Verification of Component-Based Systems (SAVCBS'05), in association with ESEC/FSE'05 (Lisbon, Portugal), September 2005.

[BM06] Tobias Bjerregaard and Shankar Mahadevan, *A survey of research and practices of network-on-chip.*, ACM Comput. Surv. **38** (2006), no. 1.

[Box98] Dob Box, *Essential COM*, Addison Wesley - Object Technology Series, 1998.

[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*, SOSP '95 : Proceedings of the fifteenth ACM symposium on Operating systems principles (New York, NY, USA), ACM Press, 1995, pp. 267–283.

[BW98] Alan W. Brown and Kurt C. Wallnau, *The Current State of CBSE*, IEEE Softw. **15** (1998), no. 5, 37–46.

[CBCP01] M. Clarke, G. Blair, G. Coulson, and N. Parlavantzas, *An Efficient Component Model for the Construction of Adaptive Middleware*, Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware'01) (Heidelberg, Germany), November 2001, pp. 160–178.

[CBG+04] G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama, *A component model for building systems software*, 2004.

[CDI+97] S. Crawley, S. Davis, J. Indulska, S. McBride, and K. Raymond, *Metameta is better-better*, 1997.

[Cel] *The cell project at ibm research*, http ://www.research.ibm.com/cell.

[CGH04] Yuhong Cai, John Grundy, and John Hosking, *Experiences integrating and scaling a performance test bed generator with an open source case tool*, ASE '04 : Proceedings of the 19th IEEE international conference on Automated software engineering (Washington, DC, USA), IEEE Computer Society, 2004, pp. 36–45.

[CHR+03] Charles Consel, Hedi Hamdi, Laurent Réveillère, Lenin Singaravelu, Haiyan Yu, and Calton Pu, *Spidle : a DSL approach to specifying streaming applications*, GPCE '03 : Proceedings of the second international conference on Generative programming and component engineering (New York, NY, USA), Springer-Verlag New York, Inc., 2003, pp. 1–17.

[CIRM93] Roy H. Campbell, Nayeem Islam, David Raila, and Peter Madany, *Designing and implementing choices : an object-oriented system in c++*, Commun. ACM **36** (1993), no. 9, 117–126.

[Cla] *Classages web site*, http ://www.cs.jhu.edu/ yliu/Classages/.

[CLW02] Han Chen, Kai Li, and Bin Wei, *A parallel ultra-high resolution MPEG-2 video decoder for PC cluster based tiled display systems*, IPDPS '02 : Proceedings of the 16th International Parallel and Distributed Processing Symposium (Washington, DC, USA), IEEE Computer Society, 2002, p. 30.

[COR] *CORBA language mapping specifications*, http ://www.omg.org/technology/documents/formal/corba_langu

[Cox86] Brad J Cox, *Object oriented programming : an evolutionary approach*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[CSL04] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal, *Dynamic instrumentation of production systems*, USENIX 2004 Annual Technical Conference, 2004, pp. 15–28.

[CWJ05] Wander O. Cesário, Flávio Rech Wagner, and Ahmed Amine Jerraya, *Hardware/software interfaces design for soc.*, The Industrial Information Technology Handbook, 2005.

[CWMa] *Common Warehouse Metamodel (CWM) Specification, Volume 1*, Object Management Group, Octobre 2001.

[CWMb] *Common Warehouse Metamodel (CWM) Specification, volume 1*, Object Management Group, Octobre 2001.

[DdHT01] Eric M. Dashofy, André ; Van der Hoek, and Richard N. Taylor, *A highly-extensible, xml-based architecture description language*, WICSA '01 : Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01) (Washington, DC, USA), IEEE Computer Society, 2001, p. 103.

[DHRA06] Matthew Drake, Hank Hoffmann, Rodric Rabbah, and Saman Amarasinghe, *MPEG-2 decoding in StreamIt*, IPDPS, 2006.

[Dij68] Edsger W. Dijkstra, *The structure of the the-multiprogramming system*, Commun. ACM **11** (1968), no. 5, 341–346.

[DK75] Frank DeRemer and Hans Kron, *Programming-in-the large versus programming-in-the-small*, Proceedings of the international conference on Reliable software (New York, NY, USA), ACM Press, 1975, pp. 114–121.

[DK06] Linda DeMichiel and Michael Keith, *Jsr 220 : Enterprise javabeans, version 3.0*, Tech. report, Sun Microsystems, 2006.

[DL03] Pierre-Charles David and Thomas Ledoux, *Towards a Framework for Self-adaptive Component-Based Applications*, DAIS, LNCS, vol. 2893, Springer, 2003.

[DL06] _____, *Safe dynamic reconfigurations of fractal architectures with fscript*, 2006.

[dmddl] Liste des marques déposées de l'OMG, *http ://www.omg.org/legal/tm_list.htm*.

[DOM] *W3C Document Object Model web site*, http ://www.w3.org/DOM.

[DvdH01] Eric M. Dashofy and André van der Hoek, *Representing product family architectures in an extensible architecture description language*, PFE, 2001, pp. 330–341.

[DvdHT05] Eric M. Dashofy, Andr&#233 ; van der Hoek, and Richard N. Taylor, *A comprehensive approach for the development of modular software architecture description languages*, ACM Trans. Softw. Eng. Methodol. **14** (2005), no. 2, 199–245.

[Ecl] *Eclipse UML*, http ://www.omondo.com.

[eco] *eCos web site*, http ://ecos.sourceware.org.

[Ei90] Institute O. Electrical and Electronics E. (ieee), *Ieee 90 : Ieee standard glossary of software engineering terminology*, 1990.

[EK95] D. R. Engler and M. F. Kaashoek, *Exterminate all operating system abstractions*, HOTOS '95 : Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V) (Washington, DC, USA), IEEE Computer Society, 1995, p. 78.

[EMF] *Eclipse Modeling Framework*, http ://www.eclipse.org/emf.

[Eng98] Dawson R. Engler, *The exokernel operating system architecture*, Ph.D. thesis, 1998, Supervisor-M. Frans Kaashoek.

[ERFL01] E. Eide, A. Reid, M. Flatt, and J. Lepreau, *Aspect weaving as component knitting : Separating concerns with knit*, 2001.

[Exc] *Excelsior Web Site*, http ://www.excelsior-usa.com/.

[Fas01] Jean-Philippe Fassino, THINK *: vers une architecture de systèmes flexibles*, Ph.D. thesis, Ecole Nationale Supérieure des Télécommunications, 2001.

[FBB+97] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers, *The Flux OSKit : a substrate for kernel and language research*, SOSP '97 : Proceedings of the sixteenth ACM symposium on Operating systems principles (New York, NY, USA), ACM Press, 1997, pp. 38–51.

[FCL06] Manuel Fähndrich, Michael Carbin, and James R. Larus, *Reflective program generation with patterns*, GPCE '06 : Proceedings of the 5th international conference on Generative programming and component engineering (New York, NY, USA), ACM Press, 2006, pp. 275–284.

[FECA04] Robert E. Filman, Tzilla Elrad, Siobhan Clarke, and Mehmet Aksit, *Aspect-Oriented Software Development*, Addison Wesley Professional, 2004.

[FG96] Cédric Fournet and Georges Gonthier, *The reflexive CHAM and the join-calculus*, POPL '96 : Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), ACM Press, 1996, pp. 372–385.

[FM98] Fabrice Le Fessant and Luc Maranget, *Compiling join-patterns*, Electronic Notes in Computer Science **16** (1998), no. 2.

[FPS+01] Bertil Folliot, Ian Piumarta, Lionel Seinturier, Carine Baillarguet, C. Khoury, A. Leger, and Frederic Ogel, *Beyond flexibility and reflection : The virtual virtual machine approach.*, IWCC, 2001, pp. 16–25.

[Fraa] *Fractal web site*, http ://fractal.objectweb.org.

[Frab] *FractNet web site*, http ://www-adele.imag.fr/fractnet.

[Frac] *FrameKit*, http ://www-src.lip6.fr/logiciels/mars/FRAMEKIT.

[FSLM02] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller, *THINK : A Software Framework for Component-based Operating System Kernels*, USENIX Annual Technical Conference, 2002.

[GBB+04] Ferid Gharsalli, Amer Baghdadi, Marius Bonaciu, Giedrius Majauskas, Wander O. Cesário, and Ahmed Amine Jerraya, *An efficient architecture for the implementation of message passing programming model on massive multiprocessor.*, IEEE International Workshop on Rapid System Prototyping, IEEE Computer Society, 2004, pp. 80–87.

[GCJ] *GCJ GNU Compiler for Java web site*, http ://gcc.gnu.org/java/.

[GCL05] John C. Grundy, Yuhong Cai, and Anna Liu, *Softarch/mte : Generating distributed system test-beds from high-level software architecture descriptions.*, Autom. Softw. Eng. **12** (2005), no. 1, 5–39.

[GMW97] D. Garlan, R. Monroe, and D. Wile, *ACME : An Architecture Description Interchange Language*, 1997.

[GRCD98] François Galilée, Jean-Louis Roch, Gerson G. H. Cavalheiro, and Mathias Doreille, *Athapascan-1 : On-line building data flow graph in a parallel language*, PACT '98 : Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (Washington, DC, USA), IEEE Computer Society, 1998, p. 88.

[Gro01] Flux Research Group, *Knit user's manual and tutorial version 1.0.0*, Tech. report, School of Computing, University of Utah, 2001.

[H2603] *Advanced Video Coding. ISO/IEC 14496-10 :2003*, Tech. report, Coding of Audiovisual Objects-art 10, 2003.

[Ham97] G. Hamilton, *Java beans specification v 1.01*, Tech. report, Sun Microsystems, 1997.

[Ham06] Ali Hamie, *On the Relationship between the Object Constraint Language (OCL) and the Java Modeling Language (JML)*, pdcat (2006), 411–414.

[HF98] Johannes Helander and Alessandro Forin, *Mmlite : a highly componentized system architecture*, EW 8 : Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications (New York, NY, USA), ACM Press, 1998, pp. 96–103.

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter, *The performance of μ-kernel-based systems*, SOSP '97 : Proceedings of the sixteenth ACM symposium on Operating systems principles (New York, NY, USA), ACM Press, 1997, pp. 66–77.

[Hil92] Dan Hildebrand, *An architectural overview of qnx*, Proceedings of the Workshop on Microkernels and Other Kernel Architectures (Berkeley, CA, USA), USENIX Association, 1992, pp. 113–126.

[HLA+05] Galen C. Hunt, James R. Larus, Martín Abadi, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, , and Brian Zill., *An overview of the singularity project*, Tech. report, Microsoft Research Technical Report MSR-TR-2005-135, Microsoft Corporation, Redmond, WA, USA, Oct 2005.

[Hoa78] C. A. R. Hoare, *Communicating sequential processes*, Commun. ACM **21** (1978), no. 8, 666–677.

[HTM+05] Assia Hachichi, Gaël Thomas, Cyril Martin, Bertil Folliot, and Simon Patarin, *A generic language for dynamic adaptation.*, Euro-Par (José C. Cunha and Pedro D. Medeiros, eds.), Lecture Notes in Computer Science, vol. 3648, Springer, 2005, pp. 40–49.

[IJ03] G. Stewart Itzstein and Mark Jasiunas, *On implementing high level concurrency in Java*, vol. 2823/2003, pp. 151–165, Springer Berlin / Heidelberg, 2003.

[ISO95] ISO/IEC, *ISO. Open Distributed Processing Reference Model - Part 3 : Architecture. International Standard ISO/IEC IS 10746-3*, 1995.

[Jav] *JavaCC web site*, https ://javacc.dev.java.net/.

[JDT] *Eclipse Java Development Tools (JDT) Supproject web site*, http ://www.eclipse.org/jdt.

[JM] *JVT software page. http ://bs.hhi.de/suehring/tml*.

[JM06] Ahmed Amine Jerraya and Trevor N. Mudge, *Guest editorial : Concurrent hardware and software design for multiprocessor soc.*, ACM Trans. Embedded Comput. Syst. **5** (2006), no. 2, 259–262.

[JNI] *Java Native Interface Specification*, http ://java.sun.com/j2se/1.4.2/docs/guide/jni/spec/jniTOC.html.

[JT03] Axel Jantsch and Hannu Tenhunen, *Networks on Chip*, Kluwer Academic Publishers, Fevrier 2003.

[JW05] Ahmed Amine Jerraya and Wayne Wolf, *Hardware/software interface codesign for embedded systems.*, IEEE Computer **38** (2005), no. 2, 63–69.

[KDH+05] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, *Introduction to the cell multiprocessor*, IBM J. Res. Dev. **49** (2005), no. 4/5, 589–604.

[KEG+97] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Brice&#241 ;o, Russell Hunt, David Maziés, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie, *Application performance and flexibility on exokernel systems*, SOSP '97 : Proceedings of the sixteenth ACM symposium on Operating systems principles (New York, NY, USA), ACM Press, 1997, pp. 52–65.

[Kis02] Ivan Kiselev, *Aspect-Oriented Programming with AspectJ*, Sams, Indianapolis, IN, USA, 2002.

[KM90] J. Kramer and J. Magee, *The evolving philosophers problem : Dynamic change management*, IEEE Transactions on Software Engineering **16** (1990), no. 11, 1293–1306.

[KS05] Sacha Krakowiak and Jean-Bernard Stefani, `export-bind` *: Un patron d'architecture pour la liaison adaptable*, Informatique Répartie (D. Trystram, Y. Slimani, and M. Jemni, eds.), Hermès, 2005, Hors-série de la Revue des sciences et technologies de l'information.

[L4K06] L4Ka Team, *L4 Experimental Kernel Reference Manual - Version X.2*, Tech. report, 2006.

[Lay05] Oussama Layaïda, *Un support logiciel à composants pour la construction d'applications multimédia adaptatives*, Ph.D. thesis, Institut National Polytechnique de Grenoble, Grenoble, December 2005.

[Lee06] Edward A. Lee, *The problem with threads*, Computer **39** (2006), no. 5, 33–42.

[LES+97] J. Liedtke, K. Elphinstone, S. Schiinberg, H. Hartig, G. Heiser, N. Islam, and T Jaeger, *Achieved ipc performance*, HOTOS '97 : Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI) (Washington, DC, USA), IEEE Computer Society, 1997, p. 28.

[LH05] Oussama Layaida and Daniel Hagimont, *Designing self-adaptive multimedia applications through hierarchical reconfiguration.*, DAIS, LNCS, vol. 3543, Springer, 2005.

[Lie93] Jochen Liedtke, *Improving ipc by kernel design*, 14th ACM Symposium on Operating System Principles (SOSP), December 1993.

[Lin] *Linux online*, http ://www.linux.org.

[LKA+95] David C. Luckham, John L. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann, *Specification and analysis of system architecture using rapide*, IEEE Transactions on Software Engineering **21** (1995), no. 4, 336–355.

[LQS05] Matthieu Leclercq, Vivien Quema, and Jean-Bernard Stefani, *DREAM : a Component Framework for the Construction of Resource-Aware, Configurable MOMs*, IEEE Distributed Systems Online **6** (2005), no. 9.

[LS05] Yu David Liu and Scott F. Smith, *Interaction-based programming with classages*, OOPSLA '05 : Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (New York, NY, USA), ACM Press, 2005, pp. 191–209.

[LSS06] Frédéric Loiret, David Servat, and Lionel Seinturier, *A First Experimentation on High-Level Tooling Support upon Fractal*, 5th Fractal Workshop - ECOOP'06, 2006.

[LV95] David C. Luckham and James Vera, *An event-based architecture definition language*, IEEE Transactions on Software Engineering **21** (1995), no. 9, 717–734.

[LÖQS07] Matthieu Leclercq, Ali Erdem Özcan, Vivien Quéma, and Jean-Bernard Stefani, *Supporting heterogeneous architecture descriptions in an extensible toolset*, 29th International Conference on Software Engineering, May 2007.

[Mar02] Raphaël Marvie, *Séparation des préoccupations et méta-modélisation pour environnements de manipulation d'architectures logicielles à base de composants.*, Ph.D. thesis, Laboratoire d'Informatique Fondamentale de Lille, 2002.

[May06] Frédéric Mayot, *Programmation parallèle à base de composants pour les applications de* streaming *– contribution à l'infrastructure Think*, Rapport de Master Recherche Informatique : Systèmes et Logiciel (Master's thesis), École Doctorale Mathématiques et Informatique, Grenoble, September 2006.

[Maz06] Sébastien Mazaré, *Dynamic Reconfiguration of THINK-based OS with FScript*, Master's thesis, ENST, Paris and Eurecom Institute, Sophia-Antipolis, June 2006.

[MB02] Giovanni De Micheli and Luca Benini, *Networks on chip : A new paradigm for systems on chip design.*, DATE, 2002, pp. 418–419.

[MB05] Vladimir Mencl and Tomas Bures, *Microcomponent-based component controllers : A foundation for component aspects*, APSEC '05 : Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05) (Washington, DC, USA), IEEE Computer Society, 2005, pp. 729–737.

[McI68] M. D. McIlroy, *Mass produced software components*, Proceedings of NATO Software Engineering Conference, 1968, pp. 138–155.

[MDA] *OMG Model Driven Architecture (MDA), Object Management Group, Juillet, 2001*.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, *Specifying Distributed Software Architectures*, Proc. 5th European Software Engineering Conf. (ESEC 95) (Sitges, Spain) (W. Schafer and P. Botella, eds.), vol. 989, Springer-Verlag, Berlin, 1995, pp. 137–153.

[MDK94] J. Magee, N. Dulay, and J. Kramer, *Regis : A constructive development environment for distributed programs*, 1994.

[Mes97] Message-Passing Interface Forum, *MPI-2.0 : Extensions to the Message-Passing Interface*, MPI Forum, june 1997.

[MIC] *The MICO CORBA Component Project*, http ://www.fpx.de/MicoCCM.

[MKJK99] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek, *The click modular router*, Symposium on Operating Systems Principles, 1999, pp. 217–231.

[Mod] *ModFact Web Site*, http ://modfact.lip6.fr.

[MOF00] *OMG Meta Object Facility (MOF) Specification, version 1.3, object management group*, Mars 2000.

[Moo00] Gordon E. Moore, *Cramming more components onto integrated circuits*, 56–59.

[MP96] David Mosberger and Larry L. Peterson, *Making paths explicit in the scout operating system*, Operating Systems Design and Implementation, 1996, pp. 153–167.

[MP03] Philippe Magarshack and Pierre G. Paulin, *System-on-chip beyond the nanometer wall*, DAC '03 : Proceedings of the 40th conference on Design automation (New York, NY, USA), ACM Press, 2003, pp. 419–424.

[MP06] Vladimir Mencl and Matej Polak, *UML 2.0 Components and Fractal : An Analysis*, 5th Fractal Workshop - ECOOP'06, 2006.

[MPEa] *MPEG-2 video codec reference C code*, `http://www.mpeg.org/MPEG/MSSG/`.

[MPEb] *MPEG-2 specification*, google : "ISO/IEC 13818" ou `http://neuron2.net/library/mpeg2/neuron2.net/library/mpeg2/iso13818-2.pdf`.

[MPE98] *MPEG-4 Specification*, Tech. report, International Standard ISO/IEC 14496-1, 1998.

[MR03] Jim S. Miller and Susann Ragsdale, *The common language infrastructure annotated standard*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[MT00] N. Medvidovic and R. N. Taylor, *A Classification and Comparison Framework for Software Architecture Description Languages*, IEEE Transactions on Software Engineering **26** (2000), no. 1.

[MW] Stephen J. Mellor and Andrew Watson, *Roles in the MDA process*.

[OIS+06] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani, *Mpi microtask for programming the cell broadband enginetm processor*, IBM Syst. J. **45** (2006), no. 1, 85–102.

[ÖJS05] Ali Erdem Özcan, Sébastien Jean, and Jean-Bernard Stefani, *Bringing ease and adaptability to mpsoc software design : A component-based approach.*, CASSIS, 2005, pp. 118–137.

[OMG01a] OMG, *CORBA 3.0 New Components Chapters*, Tech. Report OMG TC Document otc/2001-11-03, Object Managment Group, 2001.

[OMG01b] ———, *CORBA/IIOP 2.4.2 Specification*, Tech. Report OMG TC Document Formal/01-02-01, Object Management Group, 2001.

[OMG02] ———, *Lightweight CORBA component model*, Tech. Report Lightweight CCM 2003 FTF ptc/2004-06-10, Object Management Group, 2002.

[Ope] *Openccm*, http ://openccm.objectweb.org/.

[Ope05] *OpenMAX Integration Layer Application Programming Interface Specification Version 1.0*, december 2005.

[Ope06] *OpenMAX Development Layer API Specification, Version 1.0.1*, Tech. report, The Khronos Group Inc, Juin 2006.

[OSC05] OSCI, *Draft standard SystemC language reference manual*, Tech. report, Open SystemC Initiative, 2005.

[PAB+95] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang, *Optimistic incremental specialization : Streamlining a commercial operating system*, Proc. 15th ACM Symposium on Operating Systems Principles (Copper Mountain CO (USA)), 1995.

[Par72] D. Parnas, *On the criteria for decomposing systems into module*, Communications of the ACM **15** (1972), no. 12, 1053–1058.

[Pau04] Pierre G. Paulin, *Automatic mapping of parallel applications onto multi-processor platforms : A multimedia application*, DSD '04 : Proceedings of the Digital System Design, EUROMICRO Systems on (DSD'04) (Washington, DC, USA), IEEE Computer Society, 2004, pp. 2–4.

[PNP06] Renaud Pawlak, Carlos Noguera, and Nicolas Petitprez, *Spoon : Program analysis and transformation in java*, Tech. Report 5901, INRIA, may 2006.

[POF01] Ian Piumarta, Frédéric Ogel, and Bertil Folliot, *YNVM : dynamic compilation in support of software evolution*, Engineering Complex Object Oriented System for Evolution, OOPSLA, Tampa Bay, Floride, Octobre 2001.

[Pos] *Poseidon*, http ://www.gentleware.com.

[POS06] Juraj Polakovic, Ali Erdem Ozcan, and Jean-Bernard Stefani, *Building reconfigurable component-based os with think*, EUROMICRO '06 : Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (Washington, DC, USA), IEEE Computer Society, 2006, pp. 178–185.

[Qué05] Vivien Quéma, *Vers l'exogiciel. Une approche de la construction d'infrastructures logicielles radicallement configurables.*, Ph.D. thesis, Institut National Polytechnique de Grenoble, ENSIMAG, 2005.

[RAA+92] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser, *Overview of the Chorus distributed operating system*, Workshop on Micro-Kernels and Other Kernel Architectures (Seattle WA (USA)), 1992, pp. 39–70.

[Ran98] Keith H. Randall, *Cilk : Efficient multithreaded computing*, Ph.D. thesis, MIT Department of Electrical Engineering and Computer Science, 1998.

[Rat] *Rational Software web site*, www.rational.com/.

[Rei01] Alastair Reid, *Report on the language knit : A component definition and linking language version 1.0.0*, Tech. report, School of Computing, University of Utah, 2001.

[RFBLO01] Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen, and Nosa Omorogbe, *The architecture of a UML virtual machine*, Conference on Object-Oriented, 2001, pp. 327–341.

[RFS+00] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide, *Knit : Component composition for systems software*, Proc. of the 4th Operating Systems Design and Implementation (OSDI), Octobre 2000, pp. 347–360.

[Rha] *Rhapsody web site*, http ://www.ilogix.com/.

[RJO+89] Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alesandro Forin, David Golub, and Michael B. Jones, *Mach : a system software kernel*, Proceedings of the 1989 IEEE International Conference, COMPCON (San Francisco, CA, USA), IEEE Comput. Soc. Press, 1989, pp. 176–178.

[Ros06] Kaj Rosengren, *Modelling and implementation of an MPEG-2 video decoder using a GALS design path*, Master's thesis, University of Linköping, 2006.

[SAH+03] C. Soules, J. Appavoo, K. Hui, D. Silva, G. Ganger, O. Krieger, M. Stumm, R. Wisniewski, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis, *System support for online reconfiguration*, 2003.

[Sch06] Douglas C. Schmidt, *Guest editor's introduction : Model-driven engineering.*, IEEE Computer **39** (2006), no. 2, 25–31.

[SDZ96] Mary Shaw, Robert DeLine, and Gregory Zelesnik, *Abstractions and implementations forarchitectural connections*, ICCDS '96 : Proceedings of the 3rd International Conference on Configurable Distributed Systems (Washington, DC, USA), IEEE Computer Society, 1996, p. 2.

[Sem] *Sematech*, http ://www.sematech.org/.

[SMM04] Kaushik Saha, Mona Mathur, and Srijib Maiti, *Parallelization of MPEG-2 video decoder in shared memory systems*, Tech. report, STMicroelectronics, 2004.

[SPC06] Lionel Seinturier, Nicolas Pessemier, and Thierry Coupaye, *AOKell 2.0 Documentation*, Tech. report, ObjectWeb, 2006.

[SPDC06] Lionel Seinturier, Nicolas Pessemier, Laurence Duchien, and Thierry Coupaye, *A component model engineered with components and aspects.*, CBSE, 2006, pp. 139–153.

[SPJ+05] Galen S. Swint, Calton Pu, Gueyoung Jung, Wenchang Yan, Younggyun Koh, Qinyi Wu, Charles Consel, Akhil Sahai, and Koichi Moriyama, *Clearwater : extensible, flexible, modular code generation*, ASE '05 : Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (New York, NY, USA), ACM Press, 2005, pp. 144–153.

[SS94] C. Small and M. Seltzer, *VINO : An integrated platform for operating systems and database research*, Tech. Report TR-30-94, Cambridge, MA, 1994.

[Sym] *Symbian os*, http ://www.symbian.com.

[Szy98] Clemens Szyperski, *Component software : beyond object-oriented programming*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[Tan01] Andrew S. Tanenbaum, *Modern operating systems*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[TBO05] Jean-Charles Tournier, Jean-Philippe Babau, and Vincent Olive, *Qinna, a component-based qos architecture.*, CBSE, LNCS, vol. 3489, Springer, 2005.

[Thi] *Think web site*, http ://think.objectweb.org.

[TKA02] William Thies, Michal Karczmarek, and Saman P. Amarasinghe, *StreamIt : A language for streaming applications*, CC '02 : Proceedings of the 11th International Conference on Compiler Construction (London, UK), Springer-Verlag, 2002, pp. 179–196.

[TKS+05] William Thies, Michal Karczmarek, Janis Sermulins, Rodric Rabbah, and Saman Amarasinghe, *Teleport messaging for distributed stream programs*, PPoPP '05 : Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming (New York, NY, USA), ACM Press, 2005, pp. 224–235.

[TM81] Andrew S. Tanenbaum and Sape J. Mullender, *An overview of the amoeba distributed operating system*, SIGOPS Oper. Syst. Rev. **15** (1981), no. 3, 51–64.

[UML04] *Unified Modeling Language (UML), version 2.0, Object Management Group*, 2004.

[UMT] *UMT-QVT*, http ://umt-qvt.sourceforge.net.

[Win] *Windows ce*, msdn.microsoft.com/embedded/windowsce/.

[xAC] *xACME web site*, http ://www.cs.cmu.edu/∼acme/pub/xAcme/.

[XMI02] *OMG Metadata Interchange (XMI) Specification, version 1.2*, Janvier 2002.

[Yac] *Yacc : Yet Another Compiler-Compiler web site*, http ://dinosaur.compilertools.net/yacc/index.html.