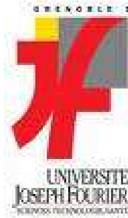


Master Recherche Systèmes et Logiciels
Université Joseph Fourier, Grenoble – ENSIMAG

Projet présenté par

Frédéric MAYOT

le 7 septembre 2006

Programmation parallèle à base de composants pour les applications de streaming

Contribution à l'infrastructure Think

ENCADREMENT

A. E.	ÖZCAN	STMicroelectronics, AST/COSA
Ph.	GUILLAUME	STMicroelectronics, AST/COSA
J.-B.	STEFANI	INRIA, Projet SARDES

JURY

J.-Cl.	FERNANDEZ	J.-L.	ROCH	(Examinateur)
J.-M.	VINCENT	G.	GOESSLER	(Examinateur)
Ph.	JORRAND			

Résumé

Les plateformes embarquées telles que les téléphones portables ou les set-top-box supportent des applications de plus en plus coûteuses en termes de calculs, tout en affichant une volonté de réduire leur consommation énergétique. Afin d'atteindre cet objectif sans pour autant en diminuer la puissance de calcul, les concepteurs de systèmes sur puce se tournent depuis quelques années vers des plateformes multi-processeurs. Nous proposons une étude visant à développer des outils adéquats pour la réalisation d'applications de streaming parallèles sur de telles architectures matérielles. Notre approche repose sur une augmentation du modèle de composant Fractal auquel nous ajoutons un langage de description de comportement collaboratif inspiré du Join calcul. Nous implémentons un compilateur intégré à la chaîne d'outils Think ADL proposant différents environnements d'exécution. Pour valider notre modèle, nous réalisons un décodeur MPEG-2 formé de sept composants parallèles.

Remerciements

Je témoigne toute ma gratitude à Philippe et son équipe chez STMicroelectronics qui m'ont chaleureusement accueilli et soutenu durant ces six mois de travail. Je remercie tout particulièrement Erdem pour son suivi et sa rigueure scientifique qui m'ont beaucoup apporté, sans oublier Matthieu qui a dû répondre à mes innombrables questions quotidiennes. Je remercie également Thierry de m'avoir accordé de son temps pour le support sur les outils autour des processeurs ST. Enfin, je remercie Jean-Bernard et l'équipe SARDES de l'INRIA qui ont apporté le financement nécessaire à ce projet.

Table des matières

1 Introduction **5**
1.1 Problématique 5
1.2 Contributions et résultats 6
1.3 Contexte de travail 7
1.4 Organisation du document 7

2 État de l'art **9**
2.1 Présentation des axes de comparaison 9
2.2 La programmation multi-threadée fortement explicite 11
2.3 La programmation parallèle explicite « assistée » 11
2.4 Les langages synchrones 12
2.5 Les *Domain Specific Language* pour le streaming 13
2.6 Les langages basés sur le Join calcul 14
2.7 Quelques exemples 15
2.8 Synthèse 19

3 Approche **21**
3.1 La programmation à base de composants 21
3.1.1 Présentation 21
3.1.2 Le modèle Fractal 22
3.1.3 La chaîne d'outils ThinkADL 23
3.2 Contribution à l'infrastructure Think ADL 23

3.2.1 Extension apportée : modèle événementiel inspiré du Join calcul . . . 23
3.2.2 Intégration dans l'ADL Fractal sur un exemple . . . 26

4 Mise en œuvre **29**
4.1 Définition du langage ThinkJoin . . . 29
4.1.1 Syntaxe . . . 29
4.1.2 Règles sémantiques . . . 29
4.1.3 Sémantique . . . 31
4.2 Compilation . . . 35
4.2.1 Extension du compilateur pour le *front end* . . . 35
4.2.2 Machine à états et génération de code . . . 37
4.3 Modèle d'exécution . . . 38
4.4 Optimisations et options de compilation . . . 43
4.5 Évolutions envisagées . . . 43

5 Évaluation **45**
5.1 Application sur un décodeur MPEG-2 . . . 45
5.1.1 Choix de l'application . . . 45
5.1.2 Fonctionnement du décodeur MPEG-2 . . . 46
5.2 Plateformes . . . 48
5.3 Résultats . . . 49
5.3.1 Aspects quantitatifs . . . 49
5.3.2 Analyse qualitative . . . 50
5.4 Synthèse . . . 50

6 Conclusion **53**
6.1 Bilan sur le projet . . . 53
6.2 Perspectives . . . 54

Bibliographie **55**

Table des figures **57**

Liste des tableaux **59**

Table des listings **61**

Partie

1

Introduction

1.1 Problématique

Les plateformes embarquées telles que les téléphones portables ou les set-top-box supportent des applications de plus en plus complexes. Cette complexité tient en deux points :

- d'une part, les algorithmes implantés requièrent une puissance de calcul grandissante alors que l'espace mémoire reste limité et la consommation énergétique une préoccupation première ;
- d'autre part, le nombre croissant de standards à supporter (comme par exemple des codecs de normes diverses) engendre de considérables problèmes quant à la maîtrise, l'intégration et le déboggage du code, tout en requérant une réutilisation importante de manière à assurer le *time to market*[1] des produits.

Afin de réduire la consommation en énergie sans pour autant diminuer la puissance de calcul, les concepteurs de systèmes sur puce (*System on Chip*, SoC) se tournent depuis quelques années vers des plateformes multi-processeurs (*Multi-processors on Chip*, MPSoC). Dédiés généralement à des fonctions de traitement du signal, ces processeurs sont regroupés dans des architectures à mémoires sophistiquées qui assurent une bande passante importante. Cependant, ces systèmes répartis deviennent encore plus difficiles à programmer.

Si l'on résume notre analyse, nous constatons les besoins suivants :

[1] Le *time to market* est le temps compris entre la conception d'un produit et sa mise sur le marché.

1. Désormais, il ne suffit plus de répartir des applications, mais il faudra les paralléliser. Cela impactera directement sur la manière dont on programme de façon à obtenir des taux de parallélisme supérieurs à ceux obtenus actuellement.
2. Une nécessité apparaît concernant les langages et plus probablement les modèles de programmation appropriés pour attaquer ce type de plateformes. La maintenance et la réutilisation étant les maîtres mots, il sera indispensable d'avoir à disposition des outils pour assembler et non écrire ex nihilo ces applications. Pour cette raison, nous nous engageons sur la programmation à base de composants qui, premièrement, identifie clairement les composants logiciels réutilisables, et secondement, favorise l'utilisation de langages haut niveau de type *Architecture Description Language* (*ADL*) pour structurer et assembler les applications.
3. Enfin, la notion de parallélisme va de pair avec la notion de collaboration entre les différents acteurs et l'échange de données. Alors que différents langages ont été proposés pour adresser ces besoins, notamment les langages synchrones, les programmeurs d'applications multimédia ont dans leur large majorité préféré employer des langages bas niveau ou classiques comme le C et le C++, manipulant directement les ressources de calcul avec, en particulier, les threads. Une impasse se profile et nous partons de l'intuition qu'il faudra explorer de nouvelles voies, tout en acceptant la programmation en C, en lui rajoutant cependant une extension qui permette de gérer aisément cet aspect collaboratif entre les acteurs parallèles.

1.2 Contributions et résultats

Dans ce travail, nous présentons des solutions qui visent à répondre à la problématique préalablement exposée. Pour ce faire, nous faisons le choix d'utiliser le modèle de composant Fractal en raison de sa légèreté et de sa flexibilité.

1. Nous définissons un langage de description de comportement collaboratif pour les composants inspiré du Join calcul. Ce langage s'implante comme extension de l'ADL Fractal et permet pour chaque composant de décrire sa manière de collaborer avec son environnement.
2. Nous implémentons un compilateur intégré au compilateur Think ADL qui génère le gestionnaire de collaboration sous forme d'une machine à états (*Finite State Machine*, FSM) optimisée pour contrôler l'exécution de chaque composant. Différents types de FSM peuvent être générés par notre compilateur pour être conformes à différents environnements d'exécution.
3. Nous implémentons une série d'environnements d'exécution avec des caractéristiques spécifiques, allant d'une exécution mono-threadée, jusqu'à une

exécution sur une architecture multi-processeurs en passant par une exécution synchrone.

4. À l'aide du modèle de programmation défini, nous avons réalisé un décodeur MPEG-2 incrémentalement, partant d'un parallélisme gros-grains pour finir avec une architecture à grains fins. Nous disposons au final d'un parallélisme de tâches réparties sur 7 composants pouvant s'exécuter de manière répartie.
5. Nous proposons des évaluations quantitatives pour la performance de notre décodeur par rapport aux implémentations existantes. En outre, nous apportons une discussion qualitative quant à l'apport de notre approche pour une application aussi complexe.
6. Enfin, nous exposons les éventuelles sources de problèmes identifiés durant l'élaboration de ce modèle et nous donnons des perspectives pour la continuation de ce projet.

1.3 Contexte de travail

Le projet se situe dans le cadre d'une collaboration STMicroelectronics avec l'équipe SARDES de l'INRIA. Les outils de Fractal/Think ont été initiés chez France Télécom puis repris par STMicroelectronics dans une perspective MP-SoC. STMicroelectronics conçoit et produit des circuits avancés développés par des équipes de recherche qui visent à mettre en place des puces intégrant des dizaines de processeurs. L'équipe ST dans laquelle a été réalisé ce projet a pour mission de définir et supporter des modèles de programmation ainsi que des systèmes d'exploitation pour ce type de systèmes.

Ce travail fait suite à une initiative d'Ali-Erdem Özcan et d'Oussama Layaida concernant la componentisation d'applications multimédia [1]. Les problèmes soulevés par cette étude méritaient d'être développé plus en avant par une approche de fond. Si la conception du modèle de programmation et de son intégration dans Think ADL ont été réalisée conjointement par A.-E. Özcan, M. Leclercq et moi-même, j'ai implémenté la totalité du code à la fois sur le framework et sur l'application test, un décodeur MPEG-2.

1.4 Organisation du document

Nous dresserons un état de l'art des langages de programmation parallèle et des langages dédiés aux applications de streaming avant de décrire dans une deuxième partie les bases de notre approche. Nous retrouverons une description du modèle de composants Fractal ainsi qu'une présentation informelle de notre modèle de programmation. La partie « mise en œuvre » donnera quant à elle une définition plus précise du langage de collaboration ainsi que de la manière dont il a été

implémenté, intégré au compilateur Think ADL et optimisé. Finalement, nous présenterons notre décodeur MPEG-2 grâce auquel nous tenterons d'évaluer notre approche. La conclusion donnera quelques éléments récapitulatifs et un ensemble de perspectives sur le projet.

Partie

2

État de l'art

Cet état de l'art passe en revue un ensemble de langages destinés à la programmation parallèle d'applications de streaming ou présentant des propriétés intéressantes pour leur mise en oeuvre. Nous débuterons notre étude par l'approche traditionnelle multi-threadée puis nous présenterons des frameworks ou langages proposant des facilités pour le parallélisme. Nous donnerons ensuite un aperçu des langages « généralistes » de type synchrone avant d'étudier trois *Domain Specific Language* (DSL) pour le streaming. Nous terminerons enfin avec des langages généralistes syntaxiquement et sémantiquement proches dans lesquels ont été introduits un modèle de concurrence spécifique basé sur le Join calcul.

Une application de streaming reçoit un flot de données qu'elle traite à la volée. Elle peut être représentée comme indiquée dans la figure 2.1 page suivante, et se compose d'un ensemble de *filtres*, F_1 à F_5 sur le schéma, reliés par des *canaux* sur lesquels transitent les données dans le sens indiqué par la flèche. Formellement, cela correspond à un graphe orienté, potentiellement cyclique comme dans notre exemple. Deux filtres peuvent être reliés par plusieurs canaux, ce qui correspondrait à un multi-graphe. Tous les langages présentés ne proposent cependant pas cette possibilité. Le filtre F_1 démultiplexe (*split*) le flux, tandis que F_5 le multiplexe (*join*).

2.1 Présentation des axes de comparaison

Nous étudierons ces différents langages sous l'angle de notre besoin : développer des applications de streaming parallèle, et ce dans un contexte industriel.

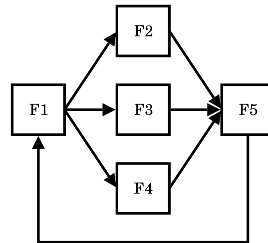


Fig. 2.1 – Exemple d'une application de streaming.

1. Le premier axe, et probablement le plus important, concerne le **modèle de concurrence**, c.-à-d. de quelle manière il est possible de programmer des tâches parallèles et comment celles-ci sont exécutées.

2. Le **modèle de synchronisation** sera notre second élément de comparaison, c.-à-d. comment coordonner les tâches parallélisées. Tous les modèles de concurrence ne nécessitent pas de synchronisation comme nous le verrons dans quelques exemples.

3. Les flots multimédia tels que de la vidéo ou de l'audio sont structurés. Par conséquent, il nous faudra analyser les possibilités offertes par ces langages pour faire transiter des **structures de données** complexes entre filtres.

4. Les applications se faisant de plus en plus complexes, l'**architecture logicielle** est devenue industriellement parlant un point crucial quant à l'évolutivité et la conception de logiciels multimédia. Il en découle souvent une méthodologie de développement appropriée qui présente une incidence certaine sur la productivité des développeurs d'applications.

Mentionnons enfin que les langages, aussi élaborés soient-ils dans leur facilité de programmer des applications parallèles ou de streaming, ne doivent pas négliger les performances. Ce point ne sera pas abordé dans cette étude du fait que certains langages constituent des produits commerciaux et d'autres de simples prototypes de recherche parfois non encore disponibles. En outre, tous ne sont pas directement utilisables pour réaliser simplement une application de streaming. Implémenter une application test dans chaque environnement serait un travail conséquent mais probablement fructueux.

2.2 La programmation multi-threadée fortement explicite

La manière la plus fréquente et la plus ancrée dans les pratiques de développement pour la programmation d'applications parallèles reste sans conteste, et depuis plus de 10 ans, la programmation à base de threads. Les ressources de calcul, représentées par les threads, sont gérées directement par le programmeur qui doit donc faire l'effort de distribuer manuellement son code. Il s'en suit que la synchronisation s'effectue aussi à la main grâce par exemple à des barrières.

Les threads font l'objet d'abstractions plus ou moins élaborées proposées dans des librairies (C++ par exemple avec *Boost Threads*) ou dans les primitives mêmes du langage comme en Java. C'est au système d'exploitation qu'il revient de mapper ces threads logiques sur des threads physiques suivant l'architecture matérielle ainsi que de réaliser leur ordonnancement.

Les modèles threadés répondent de manière générale à la programmation parallèle sur multi-processeurs à mémoire partagée (SMP). Dans la plupart des implémentations des modèles de thread, ces derniers partagent leur espace mémoire et il convient de les synchroniser à l'aide de primitives de synchronisation comme des mutex ou encore des sémaphores. Dans le cas de mémoires distribuées, il convient d'utiliser des librairies comme MPI.

Qui a déjà réalisé un programme multi-threadé un tant soit peu conséquent sait à quel point il est difficile d'en maîtriser le code. Un des problèmes majeurs soulevé dans [2] tient au non-déterminisme de ce modèle du point de vue du programme. L'ordonnancement est réalisé de manière externe par le système d'exploitation, ce qui rend le déboggage particulièrement fastidieux, voire impossible.

Malgré cela, le multi-threading est retenu pour deux raisons. Premièrement, il est proposé comme extensions de langages connus et évolués (orientés objets) largement éprouvés pour la conception d'applications complexes. Secondement, il cadre dans une méthodologie de développement où l'on débute par une implémentation séquentielle pour ensuite se poser le problème de la répartition. Comme les ressources sont gérées à la main, et que l'on part de programmes séquentiels, le parallélisme obtenu reste par conséquent limité.

2.3 La programmation parallèle explicite « assistée »

Nous introduisons deux langages présentant des similarités dans leur approche de la programmation parallèle, Cilk et Athapascan qui proposent une abstraction plus adéquate pour le parallélisme.

Cilk

Cilk, dont [3] en présente les principaux traits, étend le langage C de deux constructions permettant de réaliser des appels de fonction asynchrones et de synchroniser ces appels par des primitives de plus haut niveau que celles fournies dans la programmation par threads. Par exemple, le code suivant

```
spawn f1(); spawn f2(); sync;
```

lance deux tâches parallèles `f1()` et `f2()`, puis se met en attente de leur terminaison (jointure). Une bibliothèque fournit des primitives pour l'exclusion mutuelle sur les variables partagées. Le scheduling des threads d'exécution se fait dynamiquement et se base sur le principe du vol de tâche. Est fourni un outil qui permet de détecter statiquement les *data races* ce qui supprime les problèmes présentés plus haut dus au non déterminisme du modèle threadé.

Athapascan

Athapascan, décrit dans [4], propose une syntaxe proche de Cilk permettant d'exécuter des appels de méthodes asynchrones. Cependant, une contribution importante tient au fait que la synchronisation est implicite grâce à un typage des accès à la mémoire (écriture, lecture). Le système construit à la volée un graphe de dépendances des données pour ordonnancer les tâches de manière non préemptive. L'implémentation est réalisée sous forme d'une bibliothèque C++ ne nécessitant donc pas de compilateur ad hoc. De même que pour Cilk, il n'y a pas de limitations quant aux structures de données du fait que les langages associés sont le C et le C++.

2.4 Les langages synchrones

A l'opposé de la programmation à base de threads se situent les langages synchrones, reposant sur une base mathématique, qui combinent synchronisme (discrétisation du temps) et concurrence déterministe. Ces langages sont totalement abstraits des ressources de calcul et peuvent être compilés vers des circuits électroniques de type FPGA (*Field-Programmable Gate Array*[1]) ou vers du code séquentiel en C (ou même multi-threadé). La compilation vers une machine à états déterministe facilite donc grandement le déboggage des applications. Au niveau de la répartition, il existe des travaux qui permettent de répartir les sous-graphes de la machine à états finale. Cependant, il n'est pas possible de gérer l'architecture

[1] Les FPGA sont des puces comportant des blocs logiques interconnectables permettant de programmer n'importe quelle fonction logique.

de la répartition. Les structures de données dans les approches synchrones restent très faibles.

Les trois principaux acteurs en sont Lustre, Esterel et Signal dont [5] donne une introduction. Esterel, de par sa syntaxe impérative (contrairement à la syntaxe déclarative de Lustre) constitue probablement le langage le plus approprié pour notre domaine d'application. Les constructions du langage Esterel permettent de programmer simplement des tâches séquentielles, parallèles et donnent des primitives de manipulation de signaux qui sont la seule manière pour deux threads de communiquer. On trouve notamment l'émission de signal, l'attente d'un signal ou encore l'exécution d'un bloc d'instruction tant qu'un signal donné n'a pas été reçu.

Ces langages restent principalement employés dans des environnements temps réels critiques nécessitant des certifications facilitées par les preuves sur les programmes que les fondations mathématiques autorisent.

2.5 Les *Domain Specific Language* pour le streaming

StreamIt

Dans la continuité des langages synchrones vus section 2.4 page précédente, nous présentons maintenant le langage StreamIt [6] dédié aux applications de streaming. Les concepteurs prétendent, et ce à juste titre, que les langages généralistes ne fournissent pas une abstraction naturelle pour représenter les flots. Ainsi, les compilateurs ne sont pas en mesure de réaliser des optimisations spécifiques sur ces flots. StreamIt se place dans le cadre du modèle de flot de données synchrones (*Synchronous Dataflow Model*), c.-à-d. où un programme est représenté comme un graphe d'acteurs indépendants qui communiquent par des canaux. Les débits de communication des filtres étant connus à la compilation, le compilateur est à même de générer un scheduling statique du programme comme montré dans [6]. [7] présente trois optimisations majeures : optimisation du scheduling de manière à limiter les défauts de cache d'instructions, fusion de filtres adjacents (inlining) et gestion optimale des buffers pour les canaux FIFO. Enfin, StreamIt propose un mécanisme particulièrement intéressant, introduit dans [8], pour faire transiter des flots de contrôle entre des filtres adjacents ou non, dans le sens du flot de données aussi bien que dans le sens inverse. Cette propriété représente un point particulièrement intéressant qui semble très complexe à mettre en œuvre dans une approche asynchrone. Une introduction au langage peut être trouvée dans [9].

StreamIt représente certainement le projet le plus actif et le plus abouti pour réaliser des applications de streaming. Ayant développé une application en se basant sur du code StreamIt (cf. section 5.1 page 45), nous avons donc pu étudier de près ce langage. Cependant le compilateur n'étant pas encore disponible, il ne nous a pas été possible de valider les performances du langage. La manière dont a

été codé l'analyseur syntaxique et la lecture des fichiers MPEG-2 nous amène tout de même à nous interroger quant à l'efficacité du code C généré. Le fait qu'aucune donnée quantitative sur les performances dans [10] n'ait été présentée confirme en partie notre intuition.

Un problème réside cependant dans le typage faible des interfaces des filtres. Seules des primitives `pop(number_of_data)`, `peek(number_of_data)` et `push(data)` pour respectivement récupérer des données en entrée, lire des données dans le canal d'entrée sans les supprimer et pousser des données en sortie. Une nouvelle version du compilateur introduit des structures de données de type record.

Spidle

Spidle [11][2], dont le développement a démarré presque simultanément à StreamIt, reste très proche dans ses objectifs, à savoir un langage basé sur les flots permettant de réaliser des optimisations et vérifications spécifiques. La notion de module autorise un découpage d'une application en filtres réutilisables avec une interface typée. Contrairement à StreamIt, Spidle permet de découper le flot d'entrée d'un filtre en nommant explicitement ses composantes et non en prenant simplement en entrée un nombre fixé de données. Un graphe de dépendance entre les filtres est généré et permet de compiler un scheduling statique. De même que dans StreamIt, il est possible de réaliser des composants (filtres) composites (dit hiérarchiques dans StreamIt). Le compilateur vérifie la correction de l'architecture.

2.6 Les langages basés sur le Join calcul

Cette section présente deux langages basés sur le Join calcul [12]. Nous reviendrons section 3.2.1 page 23 sur ce modèle de calcul plus en détail[3]. Join Java et Polyphonic C# proposés respectivement dans [13] et [14] sont des extensions de Java et C# qui masquent l'utilisation de messages et de threads derrière deux constructions du langage :

- En Polyphonic C#, le mot clé `async` qui remplace **void** pour le type de retour d'une fonction représente la première extension. Comme la syntaxe le laisse supposer, l'invocation d'une méthode `async` provoque un appel de fonction asynchrone dans un autre thread d'exécution. La méthode étant asynchrone, aucune valeur ne pourra être retournée.
- La seconde extension se nomme *chord* et définit un pattern de synchronisation. Au lieu d'avoir un prototype de fonction « standard » tel que

[2] À noter que les critiques faites à StreamIt dans [11] ne sont plus d'actualité et les possibilités de Spidle restent, à notre sens, inférieures à celles offertes actuellement par StreamIt.

[3] Il est à noter que l'implémentation originelle du Join calcul a été faite sur Caml et porte le nom de JoCaml.

public `string Get()`, on peut exprimer une conjonction de prototypes, comme par exemple **public** `string Get()` & **public** `async Put(string s)` dont la sémantique est la suivante : le corps de la fonction ne sera exécuté que lorsque toutes les fonctions composant le pattern auront été appelées.

Une restriction du langage veut qu'il n'y ait qu'une seule méthode synchrone par *chord*. Le compilateur se charge de réaliser les verrous nécessaires. Il convient ensuite de programmer l'exclusion mutuelle en se basant sur ce paradigme. De nombreux exemples sont donnés dans [14], montrant ainsi que l'on peut élégamment et concisément coder des problèmes standards de la programmation parallèle à l'aide de ce modèle. Les performances de Polyphonic C# sont globalement comparable à du code écrit « à la main » en C# pour une exécution sur une machine comportant plusieurs threads physiques.

JoinJava présente les mêmes caractéristiques excepté qu'il propose plusieurs politiques pour prioriser les *chords*. L'implémentation du pattern matcher qui permet de vérifier si l'invocation d'une fonction réalise le pattern de synchronisation diffère légèrement.

Nous reviendrons également dans la section 3.2.1 page 23 sur la manière dont sont compilés ces patterns de synchronisation.

2.7 Quelques exemples

L'exemple schématisé figure 2.2 page 17 cherche à illustrer certains des langages discutés précédemment. Le filtre C_1 est une source qui produit en permanence des données vers C_2 et C_3. Ces derniers effectuent une multiplication des données en entrée et les poussent vers C_4 qui somme les entiers en provenance de C_2 et C_3.

Le code figures 2.1 page suivante, 2.2 page 17 et 2.3 page 18 donne une implémentation possible respectivement en StreamIt, Esterel et Polyphonic C#.

On remarquera que dans la version StreamIt, les filtres ne sont pas dépendants entre eux. `C1` n'a pas connaissance du fait qu'il soit connecté à `C2`. Ceci peut être pratique mais pose des problèmes de typage des interfaces comme nous l'avons évoqué plus haut. Un filtre composite supplémentaire `C1C2` est nécessaire pour réaliser le multiplexage/démultiplexage du flux.

Le programme en Esterel mérite quelques explications. Il est possible de découper le code en modules comme dans les exemples précédents, ce que nous ne faisons pas ici. Le signe || sépare des sections de code parallèles et ; des sections séquentielles. `emit` émet un signal, `await` attend un signal et `every S` **do** `inst end every` déclenche les instructions `inst` sur réception du signal `S`. L'expression [`await O3` || `await O4`] représente donc une attente parallèle des signaux `O3` et `O4`. Si l'on émet `O1`(2) `O2`(3) les signaux `O3`(4) `O4`(9) `R`(13) vont être produits simultanément.

```
void -> void pipeline AppliTest {
  add C1;
  add C2C3;
  add C4;
}

void -> int filter C1 {
  work push 2 {
    while (true) {
      push(2);
      push(4);
    }
}

int -> int splitjoin C2C3 {
  split roundrobin(1,1);
  add C2;
  add C3;
  join roundrobin(1,1);
}

int -> int filter C2 {
  work pop 1 push 1 {
    push(pop() * 2);
  }
}

int -> int filter C3 {
  work pop 1 push 1 {
    push(pop() * 3);
  }
}

int -> void filter C4 {
  work pop 2 {
    print(pop() + pop());
  }
}
```

Listing 2.1 – Code StreamIt pour l'application exemple.

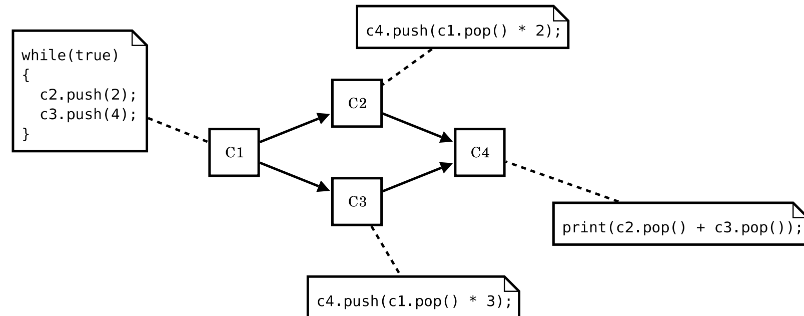


FIG. 2.2 – Schéma de l'application exemple.

```
module AppliTest:
inputoutput O1 : integer, O2 : integer, O3 : integer, O4 : integer;
output R : integer;
  every O1 do
    emit O3(?O1 * 2);
  end every
||
  every O2 do
    emit O4(?O2 * 3);
  end every
||
  loop
    [await O3 || await O4];
    emit R(?O3 + ?O4);
  end loop
end module
```

Listing 2.2 – Code Esterel pour l'application exemple.

```
class C1 {
  private C2 c2;
  private C3 c3;
  public C1(C2 c2, C3 c3) {
    this.c2 = c2; this.c3 = c3;
  }
  async start() {
    while (true) {
      c2.push(2);
      c3.push(4);
    }
  }
}

class C2 {
  private C4 c4;
  public C1(C4 c4) { this.c4 = c4; }
  async push(int i) { c4.push1(i * 2); }
}

class C3 {
  private C4 c4;
  public C1(C4 c4) { this.c4 = c4; }
  async push(int i) { c4.push2(i * 3); }
}

class C4 {
  async push1(int i1) && async push2(int i2) {
    Console.Out(i1 + i2);
  }
}

class AppliTest {
  C4 c4 = new C4();
  C3 c3 = new C3(c4);
  C2 c2 = new C2(c4);
  C1 c1 = new C1(c2, c3);
  c1.start();
}
```

Listing 2.3 – Code Polyphonic C# pour l'application exemple.

	Concurrence	Synchro.	Structures de données	Architecture logicielle	Productivité
Threads	*	*	****	****	**
Framework parallélisme	***	**	****	****	***
DSL Streaming	****	****	**	***	****
Basé sur le Join calcul	****	****	****	****	***
Langages synchrones	****	****	*	*	**

TAB. 2.1 – Synthèse et évaluation des différents langages pour chaque critère sur une échelle de 1 (non adapté) à 4 (totalement adéquate).

2.8 Synthèse

Les modèles à bases de threads et les frameworks de programmation parallèle présentés ne disposent pas de construction intrinsèque au langage pour les patterns de synchronisation. Ceci se révèle assez gênant puisque le mode « push » représente la manière la plus naturelle d'écrire une application de streaming.

Les DSL pour le streaming et les langages synchrones vus plus haut nécessitent une compilation globale qui n'autoriserait pas de reconfiguration dynamique.

Enfin, les langages basés sur le Join calcul proposent une abstraction attrayante de la concurrence mais qu'il n'est pas forcément trivial d'utiliser. Le fait d'autoriser des appels synchrones dans les *chord* ne permet pas non plus d'analyser le nombre de threads potentiellement démarrés de manière sous-jacente par le runtime.

Le tableau 2.1 synthétise les différents modèles suivant les axes proposés. Concernant la concurrence et la synchronisation, nous classons les langages en fonction de l'effort que le programmeur devra fournir pour paralléliser une application. Les notes pour les structures de données et l'architecture découlent directement des langages sous-jacents ou de leurs propriétés spécifiques pour le domaine du streaming. La productivité devrait certes être meilleure pour des langages dédiés pour autant que leur syntaxe soit déjà connue (StreamIt est par exemple proche du Java). Le classement des langages synchrones est assez subjectif du fait que le paradigme de programmation est peu commun dans le monde des applications multimédia.

Partie

3

Approche

Cherchant à poser un compromis entre les différents langages présentés dans la partie précédente, notre approche repose sur le modèle de programmation à base de composants qui présente une abstraction adéquate pour la mise en œuvre d'applications de streaming comme montré dans [1] pour le cas particulier d'un décodeur audio/vidéo. Nous donnerons un aperçu du modèle utilisé, Fractal, ainsi que la chaîne d'outils extensibles associées, ThinkADL, sur laquelle nous avons réalisé l'implémentation. La section suivante présentera notre modèle de concurrence, inspiré du Join calcul, où une application sera composée de composants actifs (c.-à-d. tournant dans son propre thread d'exécution) communiquant par le biais de canaux FIFO qu'il est possible de synchroniser à l'aide de constructions syntaxiques proches des *chord* de Polyphonic C#. Un exemple conclura cette partie.

3.1 La programmation à base de composants

3.1.1 Présentation

Les composants logiciels ont émergé au début des années 90 suite à un besoin grandissant en matière de réutilisabilité logicielle. [15] donne une définition d'un composant logiciel comme étant « une unité de composition qui peut être déployée de manière indépendante et qui se prête à la composition par une tierce partie. » On trouve dans cette approche la notion de *reconfiguration dynamique*, c.-à-d. la possibilité de remplacer un composant au sein d'un système sans interruption et sans nécessiter une recompilation globale de l'application. S'ajoute à cela la notion d'*interfaces* de manière à spécifier explicitement les opérations fournies mais aussi

requises par le composant (c'est en cela aussi que les modèles à composants vont au-delà des modèles objets), réalisant ainsi une séparation entre interface et implémentation. Des langages de description d'architecture (ou *ADL* pour *Architecture Description Language*) permettent de réaliser une application par la liaison d'un ensemble de composants.

Une explication synthétique des enjeux de la programmation à base de composants est argumentée dans [16].

3.1.2 Le modèle Fractal

Fractal est un modèle de composant général et peu restrictif reposant sur des bases mathématiques rigoureuses. Il fournit un canevas logiciel (API) indépendamment de toute implémentation. Plusieurs implémentations existent, en particulier Julia (pour le Java) et Think (multi-langages). Une description plus poussée peut être trouvée dans [17]. Nous donnerons ici des éléments tirés du cours de Sacha Krakowiak [18].

Dans un premier temps, différencions les deux types de composants du modèle :

- les *primitifs*, composés d'un contenu, c.-à-d. du code exécutable fonctionnel (éventuellement existant), ainsi que d'une membrane constituée d'un ensemble de contrôleurs à même de pouvoir fournir des propriétés non-fonctionnelles comme de l'introspection.
- les *composites*, agrégeant un ensemble de composants primitifs ou composites et permettant de donner au système une vue uniforme des applications à différents niveaux d'abstraction.

A noter qu'il existe également la notion de composant partagé entre plusieurs composites qui offrent une commodité pour modéliser les ressources tels que des allocateurs mémoires.

Les interactions entre un composant et le monde qui l'entoure se réalise uniquement par le biais d'interfaces. On différencie les interfaces dites *clientes*, requises par le composant, des interfaces dites *serveurs*, fournies par celui-ci. Les interfaces dans Fractal sont fortement typées ; elles implantent un *type d'interface* définissant les méthodes qui la composent.

L'architecture d'une application se définit au moyen d'un langage, Fractal ADL, possédant la propriété d'être extensible en partie grâce à sa syntaxe XML. Le fichier ADL définit des instances de type de composants et relie les interfaces (c.-à-d. des instances de type d'interface) par un mécanisme de liaison (*binding*), connectant chaque interface cliente vers une interface serveur. On trouve des liaisons primitives dont une implémentation en C peut par exemple représenter une interface cliente comme un pointeur vers une interface serveur. Également, des liaisons dites composites permettent de lier des composants au travers d'autres composants (par exemple, suivant un pattern de proxy avec *stub* et *skeleton*).

3.1.3 La chaîne d'outils ThinkADL

A l'origine, THINK est une implémentation en C du modèle Fractal destinée à la construction de noyaux de système d'exploitation configurable [19]. Récemment, une nouvelle chaîne d'outils a été développée conjointement par STMicroelectronics, l'INRIA (SARDES) et France Télécom R&D. Le compilateur ThinkADL prend en entrée des fichiers de description d'architecture (ADL) référençant des descriptions d'interfaces (IDL) et génère le code approprié. De par sa conception en composants, cette chaîne d'outils reste facilement extensible, et permet de s'adapter à un nombre non limité de langages. Actuellement, un support pour le C, le C++ et le Java est proposé.

Pour une description complète de l'infrastructure, le lecteur est invité à se reporter à [20].

3.2 Contribution à l'infrastructure Think ADL

Nous visons à proposer un modèle de programmation adéquat pour la programmation d'applications de type streaming, potentiellement parallélisables. Ce modèle est basé sur des composants qui constituent des domaines d'encapsulation de comportement, et qui interagissent avec leur environnement uniquement au travers de leurs interfaces. Ainsi définis, les composants nous fournissent des éléments logiciels "autonomes" et répartissables possédant des dépendances explicites vers leur environnement. Nous augmentons la description des composants d'un comportement collaboratif, c.-à-d. de quelle manière ils réagissent avec leur environnement au travers leurs interfaces. Nous utilisons ces descriptions pour générer automatiquement le gestionnaire d'interactions sous forme de machine à états adaptée à des environnements d'exécution donnés. Nous découplons alors l'implémentation des composants de la façon dont ils sont exécutés (un thread, multiple threads, répartis sur plusieurs machines) et comment ils reçoivent et envoient des données.

Pour atteindre cet objectif, nous nous basons sur le modèle Fractal et son ADL, fournissant uniquement une vue structurelle du logiciel, que nous augmenterons avec des descriptions de collaborations comme décrit ci-dessus. Le langage de collaboration (ou d'interaction) implanté est basé sur le Join calcul que nous introduirons dans un premier temps.

3.2.1 Extension apportée : modèle événementiel inspiré du Join calcul

Une description complète du Join calcul se trouve dans [12]. Nous ne ferons que survoler informellement ce modèle destiné à la programmation concurrente et distribuée pour en dégager sa sémantique. Celle-ci repose sur une *reflexive chemical abstract machine* (CHAM). L'état du système est représenté par une « soupe

chimique »[1] contenant des définitions actives ainsi que des processus en cours d'exécution. Des règles de réductions définissent des réactions qui peuvent potentiellement se produire lorsque tous les termes de la partie gauche de la règle sont présentes dans la « soupe ». La réaction, s'exécutant, a pour effet de supprimer les termes de gauche et de générer les termes de droite de la règle.

Nous allons illustrer ceci par un exemple simple. Prenons la règle suivante :

$$R_1 = push(image)|pret(decodeur) \triangleright decodeur(image)$$

Nous appellerons *pattern de synchronisation* (*join pattern*) la partie à gauche du signe $\triangleright$, et *réaction* la partie droite. Admettons que *push*(*i1*), *push*(*i2*) et *pret*(*mpeg2dec*) soient présents dans la « soupe » ; alors la règle R_1 va s'appliquer de manière non déterministe. Le calcul ne précise pas si le message *push*(*i1*) ou *push*(*i2*) doit être pris en compte. En supposant que *push*(*i1*) soit utilisé, la « soupe » contiendra après réduction *push*(*i2*) et *mpeg2dec*(*i1*).

Considérons maintenant que notre système contienne une règle supplémentaire

$$R_2 = push(image_1)|push(image_2) \triangleright fusionne(image_1, image_2)$$

Avec le même contenu initial que dans l'exemple précédent, nous nous apercevons que R_1 aussi bien que R_2 peuvent être exécutées. Là non plus, le Join calcul ne précise pas laquelle des deux règles appliquer, seulement qu'une et une seule doit l'être.

De l'exposé informel de cette sémantique, nous dégageons qu'il devient assez délicat de programmer sans résoudre ces problèmes d'indéterminisme. Ce constat a également conduit les concepteurs de Join Java à augmenter le calcul des propriétés suivantes. :

- Les règles peuvent être ordonnées, c.-à-d. que le pattern matcher regardera le pattern de synchronisation de la première, puis de la deuxième et ainsi de suite jusqu'à trouver celui qui correspond (*match*) avec les messages présents dans la « soupe ». Dans notre exemple, si R_2 et R_1 avaient été définies dans cette ordre, le contenu initial aurait provoqué le déclenchement de R_2.

- Les messages ne sont pas envoyés parallèlement mais séquentiellement et les règles s'appliqueront sur le premier message posté. En admettant que *push*(*i2*) ait été envoyé avant *push*(*i2*) dans le premier exemple, R_1 s'appliquera sur *push*(*i2*).

Si ce modèle de calcul se prête bien a une implémentation où existent des messages asynchrones et des messages synchrones, il reste peu approprié lorsqu'il n'existe que des messages asynchrones. Nous gardons dans notre modèle de calcul les deux propriétés énoncées ci-dessus. Prenons maintenant l'exemple suivant dans

[1] Il s'agit de la terminologie employée dans [12].

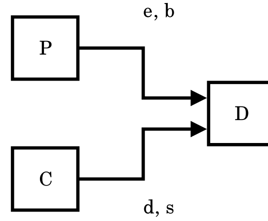


FIG. 3.1 – Exemple d'application de streaming minimale comportant un parseur P, un décodeur D et une commande C.

le cadre d'un décodeur vidéo pour mieux soulever les problèmes qui surviennent. Le système, présenté figure 3.1, est composé de trois composants un lecteur/parseur de fichier P, un décodeur D et une commande C. Le parseur envoie au décodeur deux types de messages : des en-têtes de séquences, notés e, et des morceaux d'images associés à ces en-têtes, notés b pour *bloc*. C transmet à D des données de contrôle : d pour décoder et s pour stopper le décodage.

Prenons les trois règles suivantes pour fixer le comportement du système :

$$R_1 = d()|s() \triangleright$$
$$R_2 = d()|e() \triangleright r_1()|d()$$
$$R_3 = d()|b() \triangleright r_2()|d()$$

La première règle se contente de consommer les messages sans rien produire. La deuxième correspond au traitement d'un entête par r_1 et produit d qui permet de poursuivre le décodage à moins que s n'ait été produit.

En admettant que P produise la séquence de messages de type

$$\langle e_1, b_1, b_2, \ldots, b_n, e_2, b_{n+1}, b_{n+2}, \ldots, b_m, \ldots \rangle$$

puis que C envoie une commande d pour débuter le décodage, avec le modèle de calcul précédent, on conserverait bien l'ordre pour les entêtes et pour les blocs, mais indépendamment les uns des autres. Il est impossible de savoir quel entête est associé à un bloc donné.

Pour pallier ce problème, nous introduisons une notion de canaux sur lesquels pourront être agrégés différents types de messages et qui garantira une politique FIFO. Ainsi, dans l'exemple précédent, et en utilisant un même canal pour faire transiter les e et les b, nous garantissons un ordre total sur les messages en entrée de D en provenance de P. Dans le cadre de l'implémentation que nous proposons autour du compilateur Think ADL, nous considérerons qu'une interface serveur

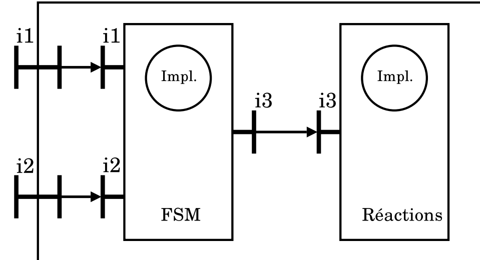


FIG. 3.2 – Vue d'un composant actif composite avec sa FSM et son sous-composant implémentant les réactions.

représente un canal sur lequel on peut éventuellement connecter plusieurs clients[2]. Remarquons que produire des messages sur un même canal en provenance de composants actifs différents ne présentent à priori pas d'intérêt puisque l'ordre dans lesquels ces messages arrivent n'est pas déterministe.

De manière à simplifier la compréhension de ce modèle de calcul, nous imposerons une limitation sur les patterns de synchronisation en spécifiant qu'un pattern ne peut contenir des types de message d'une même interface. Par exemple, si une interface i peut recevoir des messages $a()$ et $b()$, il ne sera pas possible d'écrire un pattern de synchronisation contenant a et b. Nous reviendrons sur ce point dans la définition complète du langage section 4 page 29.

3.2.2 Intégration dans l'ADL Fractal sur un exemple

L'exemple 3.2 présente un composant actif très simple. Le composant fourni les interfaces `i1` et `i2` ayant les types décrits par les IDL figure 3.1 page suivante sur lesquelles peuvent transiter respectivement des messages contenant des images de deux types à décoder. Le comportement collaboratif est donné dans un fichier de règles `behavior.rules` (suivant l'attribut `class` du composant `fsm`) reproduit listing 3.3 page 28. Enfin, nous donnons le fichier C d'implémentation du composant `reactions` dans le listing 3.4 page 28.

Dès que le composant a reçu une image dans `i1` et une autre dans `i2`, la réaction `decode(img1, img2)` est invoquée. Si la FSM reçoit un flux d'images sur `i1` et rien sur `i2`, elle les stocke en interne jusqu'à réception d'images sur `i2`.

Le composant primitif `fsm` contient la machine à états dont le comportement a été brièvement décrit dans la section précédente. La réaction est déportée dans un

[2]Il convient de bien dissocier interface et type d'interface. Le type d'interface définit les méthodes (ou messages). L'interface est une instance d'un type. Un composant peut donc avoir plusieurs interfaces du même type mais l'ordre des messages entre elles sera indéterminé.

```
public interface PImageStream {
  void pushPImage(PImage img);
}

public interface BImageStream {
  void pushBImage(BImage img);
}

public interface Decoding {
  void decode(PImage img1, BImage img2);
}
```

Listing 3.1 – Interfaces utilisée dans le composant exemple.

autre composant `reactions`. Le compilateur que nous avons réalisé se charge de générer le code propre au composant `fsm` en fonction du fichier de règles décrivant les patterns de synchronisation.

Remarquons que ce composant `fsm` aurait plutôt sa place dans la membrane du composant `comp`. Cependant, il n'est pour l'instant pas possible de modifier la membrane de sorte que les interfaces externes de cette membrane diffèrent des interfaces internes. Une telle fonctionnalité est à l'étude dans la prochaine version de Fractal (v3).

```
<definition name="comp">
  <interface name="i1" signature="PImageStream" role="server" />
  <interface name="i2" signature="BImageStream" role="server" />

  <component name="fsm">
    <interface name="i1" signature="PImageStream" role="server" />
    <interface name="i2" signature="BImageStream" role="server" />
    <interface name="i3" signature="Decoding" role="client" />
    <content class="behavior" language="join" />
  </component>

  <component name="reactions">
    <interface name="i3" signature="Decoding" role="server" />
    <content class="reactions" language="C" />
  </component>

  <binding client="this.i1" server="fsm.i1" />
  <binding client="this.i2" server="fsm.i2" />
  <binding client="fsm.i3" server="reactions.i3"/>
</definition>
```

Listing 3.2 – Description ADL du composant exemple.

```
i1.pushPImage(img1) & i2.pushBImage(img2) => i3.decode(img1, img2);
```

Listing 3.3 – Définition du fichier de règles.

```
DECLARE_DATA {
  // Données privées du composant
};

#include <think.c>

void METHOD(i3, decode)(PImage img1, BImage img2) {
 // Implémentation
}
```

Listing 3.4 – Implémentation en C de la réaction.

Partie

4

Mise en œuvre

4.1 Définition du langage ThinkJoin

Nous aborderons successivement dans cette section la syntaxe du langage tel que nous l'avons implémenté, les règles sémantiques qui le régissent et la sémantique donnée sous forme algorithmique qui sera illustrée et commentée.

4.1.1 Syntaxe

Nous donnons figure 4.1 page suivante la grammaire du langage. De même que pour Polyphonic C# et Join Java, nous employons le signe `&` et non `|` comme dans la syntaxe originelle du Join calcul. Notons que les commentaires à la C++ sont acceptés mais ne sont pas représentés dans la grammaire pour des raisons de lisibilité. Les identificateurs (terminal *idf*) sont des expressions régulières identiques à celle des identificateurs de l'IDL de Think.

Il est important de remarquer que grâce à cette syntaxe, nous séparons l'aspect interaction du composant de son implémentation en référençant une méthode d'une interface pour la réaction, et ce, contrairement à Polyphonic C# et Join Java qui fusionnent ces deux concepts.

4.1.2 Règles sémantiques

Nous énonçons l'ensemble des règles sémantiques :

1. le premier identificateur d'une méthode correspondant à un message doit être celui d'une interface serveur du composant.

Définition	→	*Règle* +
Règle	→	*Pattern* `'=>'` *Réaction* `';'`
Pattern	→	*Message* (`'&'` *Message*)∗
Message	→	*Méthode*
Réaction	→	*Méthode* | `'empty'`
Méthode	→	*idf* `'.'` *idf* `'('` *idf* ∗ `')'`

FIG. 4.1 – Grammaire BNF du langage ThinkJoin.

2. le premier identificateur d'une méthode correspondant à une réaction doit être celui d'une interface cliente du composant.
3. le second identificateur d'une méthode, après le point, est l'identificateur de la méthode dans l'interface.
4. les identificateurs inclus entre les parenthèses d'une méthode sont les paramètres formels de la méthode, mais n'ont pas nécessairement le même identificateur que dans la définition de l'interface.
5. le nombre de paramètres formels d'une méthode doit être le même que celui de la méthode correspondante définie dans le type d'interface.
6. les paramètres formels des méthodes pour un pattern donné doivent porter des identificateurs différents. Par exemple, `i1.m1(a, a) => i2.r1(a);` n'est pas autorisé, mais `i1.m2(a) => i2.r2(a, a);` est légal.
7. les identificateurs des paramètres formels de la méthode de la réaction doivent correspondre à des identificateurs des paramètres formels des méthodes des messages du même pattern. `i1.m1(a) => i2.r1(b);` n'est pas légal du fait que `b` ne correspond à aucun paramètre des messages du pattern.
8. les types des arguments de la méthode de la réaction doivent correspondre avec les types des arguments des messages.
   Si nous avons la règle `i1.m1(a) => i2.r1(a);` et si les méthodes ont les prototypes **void** `m1(`**int** `a)` et **void** `r1(string a)`, la règle n'est pas légale.
9. les méthodes ne doivent pas retourner de valeur. `i1.m1(a) => i2.r1(a);` avec **int** `m1(`**int** `a)` et/ou **int** `r1(`**int** `a)` n'est pas autorisée.
10. un pattern ne peut pas contenir des messages d'une même interface. La règle `i1.m1(a) & i1.m2() => i2.r1(a);` n'est donc pas valide.
11. une méthode d'une interface donnée ne peut apparaître qu'une seule fois dans un pattern. `i1.m1() & i1.m1() => i3.r1();` n'est pas acceptée, mais `i1.m1() & i2.m1() => i3.r1();`, `i1` et `i2` étant instance du même type d'interface, l'est du fait que les interfaces sont différentes. Cette règle est incluse dans la règle précédente mais nous préférons la séparer étant donné que nous envisageons de la relâcher

pour implémenter des patterns dits non-linéaires. Nous reviendrons sur cette propriété section 4.5 page 43.

12. le nombre de messages contenus dans l'ensemble des patterns est limité par une constante. Un message étant caractérisé par un couple interface/méthode, et en prenant les règles `i1.m1() & i2.m1() => i2.r1();` et `i2.m1() => i2.r1();`, `i1` et `i2` étant du même type, le nombre total de messages est 2. Ceci constitue seulement une restriction due à notre implémentation. Elle est actuellement fixée à 64.

Enfin, signalons la production d'un avertissement lors d'un cas d'inclusion. Prenons l'exemple suivant : `i1.m1() => i3.r1(); i1.m1() & i2.m2() => i3.r1();`. Les patterns étant lus dans l'ordre de leur déclaration, l'arrivée d'un message `m1` lancera toujours la première règle privant ainsi la seconde de s'exécuter.

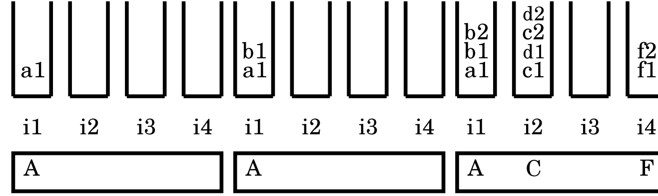
4.1.3 Sémantique

L'arrivée des messages pour chaque interface est séquentialisée, mais le comportement pour l'arrivée de deux messages sur des interfaces différentes n'est pas défini.

Pour simplifier, nous considérerons dans cette sous-section que les messages sont sans paramètre et que la réaction prend en paramètre des messages complets et non certains paramètres des messages comme décrit paragraphe 4.1.1 page 29.

Nous donnons une description algorithmique de l'exécution figure 4.1 page 33. Commençons par quelques définitions de type. `queue_type` correspond à une structure de donnée de queue sur lesquelles nous aurons les opérations `enqueue(queue, msg)`, `dequeue(queue)`, `head(queue)` et `isempty(queue)` pour respectivement ajouter un élément en fin de queue, retirer et retourner l'élément en tête de queue, accéder à l'élément en tête sans le supprimer et enfin, savoir si la queue est vide ou non. On suppose que `head(queue)` retourne -1 lorsque la queue est vide. `message_type` est le type d'un message. Cet entier correspond au couple interface/méthode mentionné plus haut et est unique pour la définition des règles. Le type `reaction_type` est une structure comprenant un pointeur de fonction ainsi que la liste des queues dans lesquelles aller chercher les messages qui lui correspondent. Le dernier type, `pattern_type`, correspond à un pattern et comprend un tableau de types de message ainsi qu'une réaction associée.

Sur réception d'un message, ce dernier est ajouté à la queue correspondant à l'interface sur laquelle il est envoyé. Est créé un tableau `status` comprenant le type de message en tête de chaque queue. On parcourt ensuite tous les patterns, dans l'ordre de leur déclaration jusqu'à en trouver un qui soit contenu dans le statut. Dans le cas où l'on en trouve un, on récupère les messages des queues correspondant à la réaction puis l'on invoque la fonction associée. Tant que l'une des queues ayant



(A) Après réception de $\langle a_1 \rangle$ (B) Après réception de $\langle b_1 \rangle$ (C) Après réception de $\langle f_2 \rangle$

FIG. 4.2 – Exemple d'exécution de l'algorithme 4.1. Au dessous des quatre interfaces est représenté le statut.

été modifiée contient encore des messages, on recommence le processus à partir du premier pattern, et ainsi de suite jusqu'à ce qu'aucun pattern ne soit détecté.

Le mot clé `empty` représente une réaction vide ; en revanche, les messages sont bien consommés dans les queues pour respecter la sémantique présentée ci-dessus.

Dans le cas particulier où l'on a un seul type de message par interface, on tombe dans le cas du Join calcul et la boucle **while** n'est plus nécessaire.

Nous allons faire tourner cet algorithme sur un exemple simple. Prenons les règles

```
i1.A() & i3.E() => i5.r1(); //P1
i1.B() & i2.C() => i5.r2(); //P2
i2.D() & i4.F() => i5.r3(); //P3
```

et supposons que l'on produise les messages suivants

$$\langle a_1, b_1, c_1, d_1, b_2, c_2, d_2, f_1, f_2, e_1 \rangle$$

Nous serons dans l'état représenté figure 4.2(a) après la réception de a_1, et ainsi de suite jusqu'à f_2 comme le montrent les figures suivantes. Aucun des patterns n'est inclus dans le statut, par conséquent, aucune réaction n'est déclenchée.

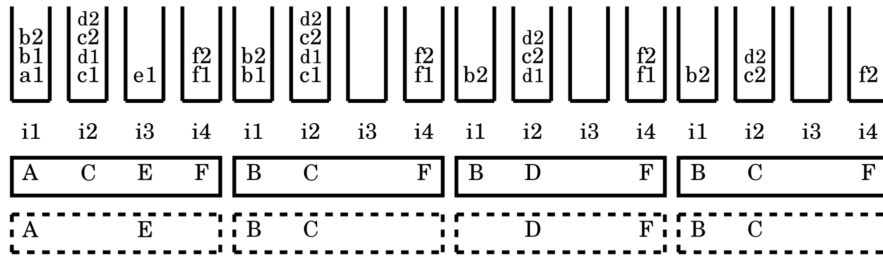
L'arrivée de e_1 va provoquer un déclenchement en chaîne des réactions. Comme on le voit dans la figure 4.3(a) page 34, la pattern P_1 match. Les premiers messages des queues de i_1 et i_3 seront supprimés et r_1 sera exécutée.

Ce modèle de calcul peut cependant conduire rapidement à deadlocks. Prenons les règles suivantes

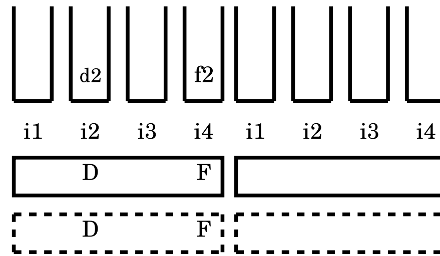
```
i1.A() & i2.C() => i3.r();
i1.B() & i2.D() => i3.r();
```

```
const int I; // nombre d'interfaces
const int P; // nombre de patterns

typedef int message_type;
typedef struct {function_type func; int queues[]} reaction_type;
typedef struct {index_type pattern[]; reaction_type reaction}
  pattern_type;

pattern_type patterns[P];
queue_type queues[I];

bool contains(message_type[] pattern, message_type[] status) {
  // retourne true si et seulement si tous les éléments
  // de pattern sont contenus dans status
}

void receive_message(int itf, message_type msg) {
  enqueue(queues[itf], msg);

  bool cont;
  do {
    message_type status[I] =
      {head_type(queues[0]), ..., head_type(queues[I])};
    cont = false;
    for (int i = 0; i < P; i++) {
      if (contains(patterns[i], status)) {
        int reaction_queues[] = patterns[i].reaction.queues;
        message_type args[reaction_queues.length];

        for (int j = 0; j < reaction_queues.length; j++) {
          args[j] = dequeue(queues[reaction_queues[j]]);
          if (!isempty(queues[reaction_queues[j]]))
            cont = true;
        }
        patterns[i].reaction.func(args);
        break;
      }
    }
  }
  while(cont);
}
```

Listing 4.1 – Algorithme décrivant la réception d'un message dans le langage ThinkJoin.



(A) Après réception de $\langle e_1 \rangle$, exécution de r_1 (B) Après exécution de r_1 (C) Après exécution de r_2 (D) Après exécution de r_3



(E) Après exécution de r_2 (F) Après exécution de r_3, état final

FIG. 4.3 – Exemple d'exécution de l'algorithme 4.1 (suite). Sous les quatre interfaces figurent le statut (rectangle continu) et le pattern détecté (rectangle en pointillés).

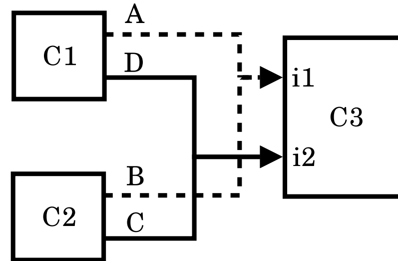


FIG. 4.4 – Exemple d'architecture pouvant mener à des deadlocks.

selon l'architecture donnée figure 4.4. L'ordre de la production des messages de C_1 et de C_2 n'est pas déterministe si bien que si C_3 reçoit la suite de message $\langle a, b, d, c \rangle$, aucune des règles ne pourra jamais être exécutée.

4.2 Compilation

4.2.1 Extension du compilateur pour le *front end*

Ainsi que nous l'avons évoqué dans l'approche, la syntaxe de l'ADL Fractal repose sur le langage XML, dont la grammaire des documents peut être décrite dans une DTD (*Document Type Definition*)[1].

La première étape pour étendre le compilateur Think ADL consiste à augmenter la DTD de l'ADL Fractal de manière à reproduire la grammaire abstraite dérivant de celle présentée au début de cette partie (c.-à-d. la même grammaire mais où les terminaux n'apparaissent plus). Nous avons donc du code proche de celui-ci :

```
<!ELEMENT rules ( rule )*>
<!ELEMENT rule ( pattern reaction )>
```

Grâce à une construction intéressante des DTD, nous réalisons une correspondance entre les types de noeuds du document XML définis par la DTD et des interfaces Java représentant les noeuds de l'AST :

```
<? add ast="rules" itf="join.ast.Rules" ?>
<? add ast="rules" itf="join.ast.RuleContainer" ?>
```

Ces interfaces définies dans les packages Java se verront implémenter par des classes générées dynamiquement par le compilateur. Quelques conventions syntaxiques sont nécessaires, notamment pour créer des conteneurs.

[1] Il existe d'autres langages comme *XML Schema Definition* pour décrire la grammaire, mais les DTD ont l'avantage d'être très simple à écrire.

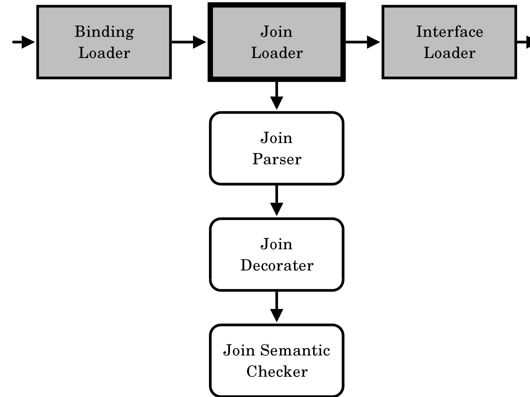


FIG. 4.5 – Intégration du loader Join dans la chaîne de load de Think ADL.

En utilisant ce mécanisme totalement identique à celui de l'ADL Fractal standard, nous pourrions directement programmer dans notre langage en XML, le compilateur se chargeant de créer l'AST automatiquement. Pour des raisons pratiques de lecture et d'écriture, nous préferons déporter ce code dans un fichier séparé comme nous l'avons vu section 3.2.2 page 26 en suivant la grammaire donnée au début de cette partie.

Vont être insérés dans la « chaîne de load » du compilateur, des composants permettant d'analyser ce fichier de règles, de créer l'arbre abstrait (*Join Parser*), de le décorer (*Join Decorater*) et d'effectuer les vérifications des règles sémantiques données section 4.1.2 page 29 (*Join Semantic Checker*). La figure 4.5 illustre l'intégration de nos composants dans le compilateur.

L'analyse lexicale et syntaxique se fait au travers de JavaCC qui est un compilateur de compilateur $LL(k)$. Il convient donc parfaitement pour une grammaire aussi simple que la nôtre et reste plus agréable à utiliser qu'un parser $LALR(1)$ comme *yacc*. Une fabrique nous permet de créer simplement les noeuds de l'AST.

Les classes de base pour les noeuds de l'arbre abstrait possèdent des attributs pour la décoration, mais non typés. Du fait que nous n'ayons pas accès à celles-ci (nous ne définissons que des interfaces, le compilateur générant les classes), nous avons créé un package pour typer fortement toutes les décorations. Il s'agit de la partie la plus importante qui facilite grandement la génération de code. Par exemple, on réalise dans cette phase des liens entre les noeuds *Message* d'une règle et le le noeud *Method* d'un IDL.

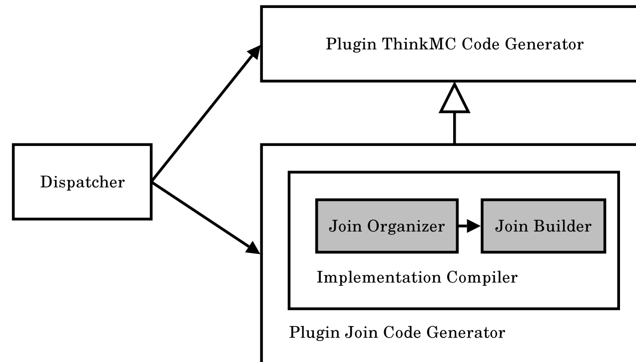


FIG. 4.6 – Plugin de génération de code pour le langage Think Join.

4.2.2 Machine à états et génération de code

La génération de code s'effectue dans Think ADL par un mécanisme de *plugin* qui permet d'éviter de modifier le compilateur pour implémenter la prise en charge d'un nouveau langage. L'avantage de ce procédé vient aussi du fait que les plugins sont chargés à la volée par un dispatcher, ce qui réduit par conséquent la taille en mémoire du compilateur.

Notre plugin est dérivé du plugin de génération de code ThinkMC (qui correspond au langage C étendu de macros facilitant la programmation). La figure 4.6 montre les deux composants qu'il a fallu développer et dont le principal est le builder.

Partant d'un AST construit, validé et décoré, il nous faut maintenant générer le code avec le composant builder du plugin.

La compilation des patterns de synchronisation (*join-patterns*) est l'objet de [21] qui introduit une formalisation à l'aide d'automates puis plusieurs techniques pour les compiler et optimiser le code généré. On peut retenir deux points importants.

Tout d'abord, la procédure que nous avons nommée `contains` dans l'algorithme 4.1 page 33 peut être assez coûteuse si l'on doit itérer sur deux ensembles pour savoir si le pattern est inclus dans le statut. Une façon plus astucieuse consiste à considérer des vecteurs de bits (des entiers longs) où chaque bit correspond à un type de message. Nous serons donc limité à la taille d'un entier long mais ceci s'avère largement suffisant en pratique puisqu'il nous permet d'utiliser 64 types de message par composant. Grâce à cette technique, l'inclusion se vérifie en une seule opération `!((p & s) ^ p)`.

Ensuite, l'itération sur tous les patterns jusqu'à trouver celui qui peut corres-

pondre au statut demeure loin d'être optimale. En revanche, le fait d'associer à chaque type de message les patterns qui lui corresponde permet de réaliser un gain de temps précieux, d'autant plus qu'en pratique, un type de message n'est présent que dans un ou deux patterns. Par contre, notre modèle de calcul étant différent du Join calcul, nous ne pouvons appliquer cela directement. En effet, pour la première itération de la boucle **while**, cela ne pose pas de problème. Là où cela ne va plus, c'est lorsqu'un pattern a été détecté et qu'une réaction a été déclenchée. Nous avons donc besoin de raffiner quelque peu notre algorithme pour l'améliorer. La nouvelle version se trouve figures 4.2 page ci-contre et 4.3 page 40. Notons que si la queue contenait déjà des messages alors aucun pattern ne peut-être déclenché.

Une première implémentation générait le code de la FSM et des queues. Cependant, le code se complexifiant à force d'optimisations, nous avons préféré le déporter dans une librairie C orientée objet inlinée[2]. Le diagramme 4.7 page 41 donne une vue de l'architecture de la FSM. Plutôt que de gérer une queue par interface pouvant contenir des messages de différents types, nous avons choisi d'implanter une queue par type de message puis de les grouper par le biais d'une structure de données `queue_group`. Ceci tient au fait que les optimisations sont simples à gérer de cette manière. Un grand soin a été apporté à la gestion de la mémoire et des structures de queue qui tiennent un rôle critique dans l'exécution.

4.3 Modèle d'exécution

Jusqu'à présent, nous avons étudié le comportement de la machine à états sur réception d'un message, où, lorsqu'un pattern match sur le statut, la réaction est exécutée immédiatement. Ce comportement n'est pas souhaitable si nous désirons réaliser du parallélisme. Le code du composant envoyant le message va exécuter la méthode de la FSM. Celle-ci devra donc comporter des verrous de manière à pouvoir être *thread-safe*[3]. La réaction, plutôt que d'être déclenchée, sera enregistrée auprès d'un scheduler connecté à la FSM. Le scheduler est partagé par plusieurs composants actifs mais n'est pas nécessairement unique dans l'application.

Nous avons ajouté à la librairie Kortex un composant scheduler basé sur les *pthreads* gérant un pool de threads. L'interface `Registration` de ce scheduler comprend une méthode **void** `register_reaction(Executable component, any id)`. Les FSM implémente l'interface `Executable`, dont le scheduler invoquera **void** `execute(any id)`

[2]Nous n'avons pas pris le parti de faire une version composants de cette librairie du fait des performances moyennes que cela aurait pu engendrer. Actuellement, le compilateur Think ADL n'optimise pas les appels de fonction pour les composants ayant une unique instance statique et passe par des tables de fonctions virtuelles.

[3]Du code est dit *thread-safe* si son comportement est correct lors d'appels simultanés par différents threads d'exécution. Par exemple, lorsque le code manipule des données partagées dans un modèle threadé, il convient de placer des verrous pour réaliser une exclusion mutuelle.

```
// nombre d'interfaces, de patterns et de types de message
const int I, P, M;

typedef int message_type;
typedef struct {function_type func; int queues[]} reaction_type;
typedef struct {bit pattern[M]; reaction_type reaction}
  pattern_type;

pattern_type patterns[M][];
queue_type queues[I];
bit status[M] = {0, ..., 0};

bool contains(bit pattern[M], bit status[M]) {
  return !((pattern & status) ^ pattern);
}

void receive_message(int itf, message_type msg) {
  bool was_empty = isempty(queues[itf]);
  enqueue(queues[itf], msg);
  if (!was_empty) return;(
  vector_type new_heads;
  status[msg] = 1;
  add(new_heads, msg);

  if (scan(itf, msg, new_heads))
    while(true) {
next:
      for (int k = 0; k < new_heads.length; k++)
        if (scan(itf, new_heads[k], new_heads))
          goto next;
      return;
    }
}
```

Listing 4.2 – Algorithme optimisé décrivant la réception d'un message dans le langage ThinkJoin.

```
bool scan(int itf, message_type msg, bit new_heads[M]) {
  for (int i = 0; i < patterns[msg].length; i++) {
    if (contains(patterns[msg][i], status)) {
      int reaction_queues[] = patterns[msg][i].reaction.queues;
      message_type args[reaction_queues.length];
      bool cont = false;
      for (int j = 0; j < reaction_queues.length; j++) {
        args[j] = dequeue(queues[reaction_queues[j]]);
        if (!isempty(queues[reaction_queues[j]])) {
          cont = true;
          status[reaction_queues[j]] = 1;
          if (!contains(new_heads, reaction_queues[j])
            add(new_heads, reaction_queues[j]);
        }
        else {
          status[reaction_queues[j]] = 0;
          remove(new_heads, reaction_queues[j]);
        }
      }
      patterns[msg][i].reaction.func(args);
      return cont;
    }
  }
  return false;
}
```

Listing 4.3 – Algorithme optimisé décrivant la réception d'un message dans le langage ThinkJoin (suite).

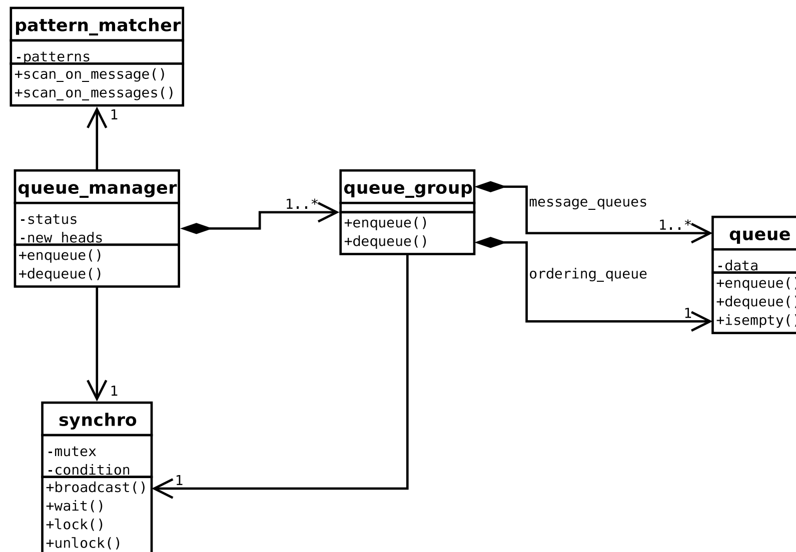


FIG. 4.7 – Diagramme UML de la librairie FSM.

pour exécuter la réaction au moment venu.

Nous illustrons ce processus dans la figure 4.8 page suivante. Des messages sont envoyés sur i_1 et i_2. La FSM détecte le pattern indiqué sur le schéma dès réception de `pushBImage(img2)`. La réaction est enregistrée auprès du scheduler. Notons que le paramètre `id` de la méthode `register_reaction` représente une sérialisation de la méthode et des arguments. Lorsque le scheduler décide d'appeler la réaction, il invoque `execute` sur la FSM en donnant le même argument `id`, qui sera désérialisé pour finalement appeler `decode`.

Le nombre de threads logiques à lancer lors du démarrage est paramétrable. Le scheduler créera une file d'attente de réactions pour chaque composant qu'il gère, en garantissant une politique FIFO et le fait qu'une et une seule réaction ne peut être exécutée à un instant donné pour un composant.

Plusieurs politiques de scheduling entre les composants sont proposées dont une à base de *timestamps* garantissant un ordonnancement globale des réactions (ce qui n'est pas optimal) et une autre de type *round robin*.

L'implémentation la plus simple et la plus efficace est de lancer tous les threads au démarrage. Chaque thread récupère ensuite une réaction à exécuter en suivant la politique fixée puis une fois terminée, en cherche une autre et se met en attente s'il n'y a rien à faire. De façon à optimiser le scheduling, une option permet de définir des exécutions par bloc, c.-à-d. qu'un thread se saisit pour un composant donné

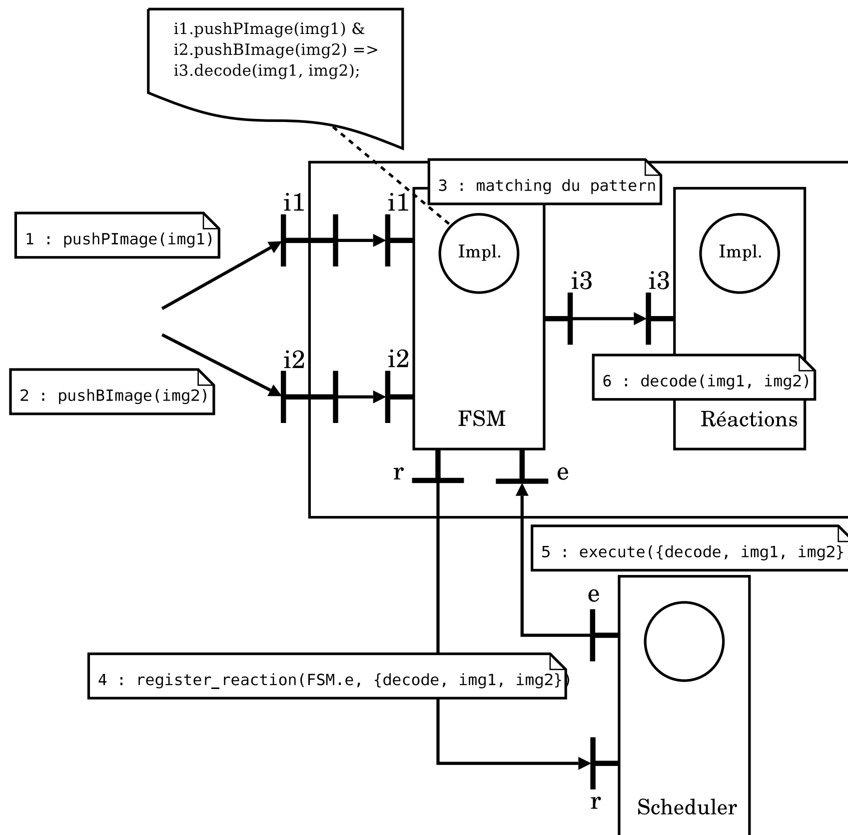


FIG. 4.8 – Enregistrement et exécution d'une réaction par le scheduler suite à la réception de deux messages.

(fonction de la politique) des n premières réactions en attente puis les exécute dans cet ordre.

Il est envisageable de le spécialiser pour une plateforme et de construire un système ad hoc pour l'application de streaming où ce scheduler gère directement des threads physiques sans l'intermédiaire d'un scheduler du système d'exploitation.

4.4 Optimisations et options de compilation

Dans cette première version du compilateur de patterns, nous implémentons quelques optimisations élémentaires :

1. une réaction `empty` ne procédera pas de la même manière et utilisera des primitives optimales pour les queues ;
2. les messages ne contenant pas d'arguments sont compilés vers des compteurs et non des structures de données complexes ;
3. si les interfaces ne contiennent qu'un seul type de message chacune, la boucle `while` de l'algorithme est supprimée ainsi que les conditions associées ;
4. les patterns triviaux (c.-à-d. lorsque pour une règle donnée, le pattern ne contient qu'un seul type de message et que ce type de message est le seul contenu dans l'interface à laquelle il appartient) ne passent pas par la FSM et les réactions sont donc directement enregistrées dans le scheduler.

Une option de compilation permet de définir un comportement synchrone pour le système, c.-à-d. que le scheduler n'est pas utilisé et que les invocations des réactions se font immédiatement à la détection du pattern. Il s'en suit que peuvent être supprimées toutes les primitives de synchronisation et notre modèle d'exécution peut être comparé en terme de performance au programme écrit traditionnellement en C.

Nous laissons la possibilité de réguler par un paramètre statique le flux sur les queues de message mais aussi du côté du scheduler sur le nombre de réactions à exécuter pour chaque composant. Il est clair que ces deux types de régulation peuvent être redondantes.

4.5 Évolutions envisagées

Nous envisageons un certain nombre d'optimisations, notamment ce qui touche à la gestion de messages ou de réactions par blocs. Des expérimentations nous ont permis de réaliser à quel point une telle technique peut avoir un impact sur les performances. Par exemple, il serait intéressant pour un composant de mettre en tampon l'ensemble des messages destinés aux autres composants, en conservant

l'ordre d'envoi, puis en transmettant par bloc ces messages. Cela aura non seulement l'avantage de pouvoir éventuellement copier par paquet dans les queues de la FSM, mais surtout, un seul verrou sera posé au niveau de la FSM, le coût des mutex pouvant devenir dramatiquement important pour des applications à grains fins.

Du point de vue des fonctionnalités, il semblerait intéressant de proposer des patterns non linéaires pour modéliser certains aspects fonctionnels. La compilation n'est toutefois pas aisée, et une refonte de la librairie sera certainement nécessaire.

Nous aurions souhaité également proposer un moyen de filtrer les messages au travers de leurs arguments dans les patterns de synchronisation, comme par exemple dans

```
i1.m1(a) & i2.m2(b) {a == b && b == 5} => i3.r(a, b)
```

Les concepteurs de Polyphonic C# rapportent une tentative d'implémentation mais se sont heurtés à des difficultés techniques. Un code naïf peut faire croître la complexité, actuellement en $O(PM)$[4] où P est le nombre de patterns et M le nombre de messages, ver $O(P \prod_i M_i)$ où M_i est le nombre de messages du type i. Des restrictions comme l'utilisation uniquement de conjonctions et l'interdiction de créer des conditions dépendant des arguments de messages différents permettraient d'ajouter cette fonctionnalité pour un coût d'exécution acceptable.

Si l'infrastructure pour la gestion du flux a été mise en place, il reste à tester son intérêt par des mesures de performances et il serait souhaitable de pouvoir gérer dynamiquement et automatiquement ces paramètres.

Notons finalement que l'aspect allocation mémoire, s'il a été correctement élaboré concernant les queues, a été laissé de côté pour les messages eux-mêmes. Comme suggéré dans [22], une allocation par *pool* reste une optimisation efficace qu'il nous semble indispensable de mettre en œuvre.

[4] Ceci est une complexité très grossière.

Partie

5

Évaluation

5.1 Application sur un décodeur MPEG-2

5.1.1 Choix de l'application

Choisir une application à implémenter pour une preuve de concept n'a rien de trivial. Si notre choix initial s'était porté sur un décodeur H.264, nous avons dû reculer devant la complexité de la norme et des implémentations disponibles. Une application de streaming trop simple, tel un décodeur AC3, ne nous aurait probablement pas permis de mettre à jour certains problèmes. En outre, il fallait pouvoir partir d'un code existant de façon à être guidé dans l'implémentation et à comparer les performances de notre système. Nous allons énoncé quelques éléments qui nous ont finalement conduit à nous pencher sur un décodeur MPEG-2.

Le lecteur est invité à se reporter à [23] pour une description synthétique et exemplifiée, et à [24] pour les spécifications complètes de la norme. Notons tout d'abord quelques considérations d'ordre méthodologique. La norme est décrite au travers de pseudo-code calqué sur l'implémentation de référence en C disponible sur [25]. Si cette manière d'aborder le standard nous incite à développer un code non structuré, les schémas dans le formalisme du début de la section 2 page 9 facilitent grandement la compréhension globale. Il est clair, au vu de cette lecture, que bénéficier d'un framework adapté au streaming permettrait d'unifier l'implémentation et la spécification.

Une équipe de STMicroelectronics a tenté la parallélisation d'un décodeur MPEG-2 dans [26] à l'aide d'un modèle threadé en réalisant à la fois un parallélisme de tâches et de données. Si les performances théoriques semblaient prometteuses,

un benchmark sur le simulateur SMP ST200 a montré qu'avec 4 processeurs, le gain n'atteignait que 1.5. Si ce chiffre paraît décevant, la cause est probablement à chercher dans le coût des appels systèmes des primitives de synchronisation et une gestion quelque peu hasardeuse des lectures de fichier.

Récemment, une implémentation en StreamIt de cette même application a été proposée dans [10]. Ceci nous a permis d'obtenir une seconde solution d'architecture parallèle. La différence avec notre approche réside cependant dans le fait que ce langage se prête plus volontiers à une décomposition à grains très fins des filtres, ce que l'on ne peut envisager avec les composants Fractal actuellement.

On trouve dans la littérature un certain nombre de propositions pour paralléliser ce type de décodeur. [27] et [28] sont deux approches centrées sur les données. [28] propose deux types de parallélisme : au niveau groupe d'images (GOP) ou niveau *slice*. Les deux solutions conduisent à des résultats comparables. Les auteurs remarquent tout de même qu'un parallélisme à grains plus gros (GOP) réduit les synchronisations mais engendre une consommation nettement plus importante de mémoire. La tendance s'inverse avec un grain plus fin tel que les *slice*. [27] va au-delà en montrant l'intérêt d'un parallélisme hiérarchique au niveau image et macroblocs. Plus récent, le framework Nizza [29], fixe des objectifs proches des nôtres, à savoir proposer un middleware orienté flot de données facilitant la programmation et proposant des optimisations de manière à satisfaire les contraintes de systèmes temps-réels. Malencontreusement, cet article reste flou sur les détails d'implémentation et ne présente au final que du parallélisme à gros grain sans synchronisation complexe.

5.1.2 Fonctionnement du décodeur MPEG-2

La figure 5.1 page suivante donne une vue schématique de notre décodeur. Les composants Header et Picture Decoder sont inclus dans le Parser et tournent dans un même monde synchrone. Nous avons choisi d'effectuer un parallélisme de tâches, sachant qu'un parallélisme sur les données pourrait avoir lieu en dupliquant ce schéma et en ajoutant quelques synchronisations, en particulier pour l'exportation des images une fois décodées.

De manière simplifiée, un flux MPEG-2 se décompose en une suite de séquences qui contiennent chacune une suite d'images comportant eux-mêmes un ensemble de macroblocs et de vecteurs de prédiction. Les messages notés `new_sequence`, `new_picture` et `new_macroblock` représente les en-têtes associés. `push_macroblock` et `push_mv` sont respectivement les données des macroblocs (c.-à-d. les parties de l'image encodés) et les vecteurs de prédiction.

Afin de donner une idée de la granularité, le tableau 5.1 page 48 indique la taille des en-têtes et des données du flux MPEG-2. Remarquons que ces chiffres sont donnés à titre indicatif et varient suivant l'implémentation. La granularité de

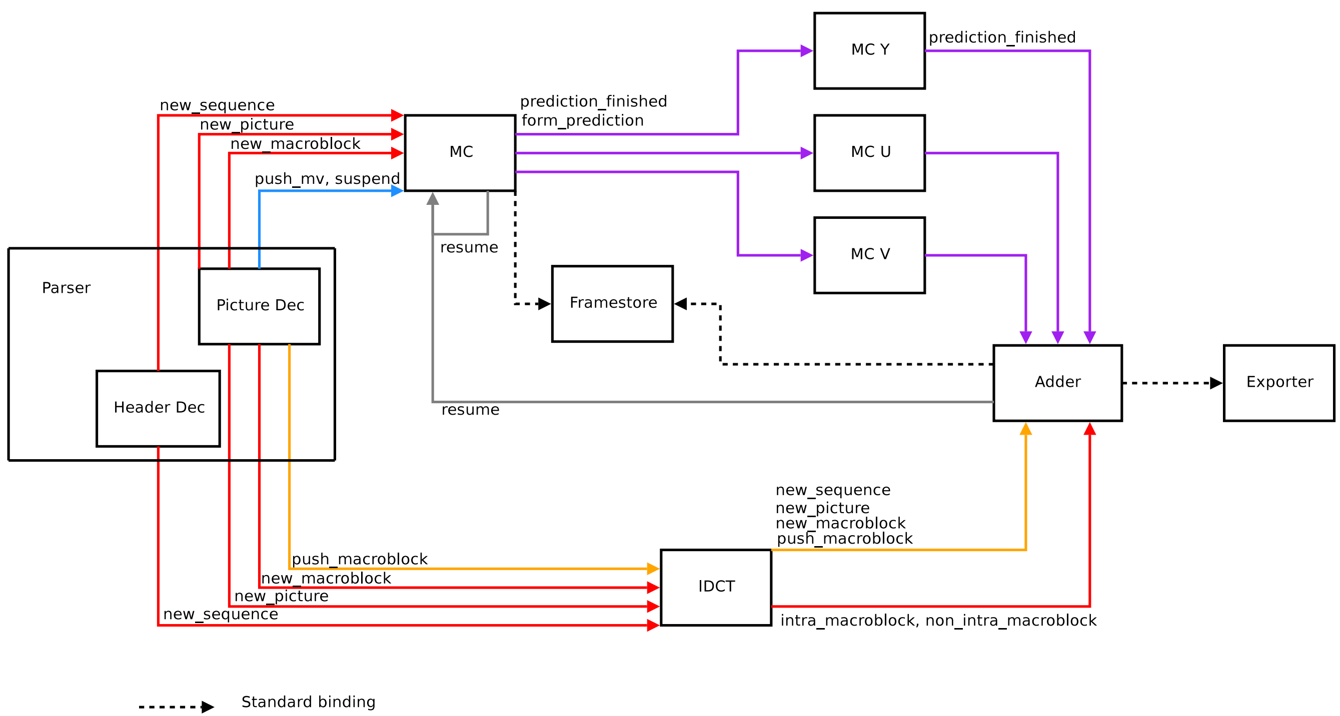


FIG. 5.1 – Architecture du décodeur MPEG-2.

Données	Taille (en octets)
En-tête de macrobloc	248
Données du macrobloc	6144
Vecteurs de prédiction	64
En-tête d'image	144
Ent-tête de séquence	400

TAB. 5.1 – Taille des données du flux MPEG-2.

notre architecture est au niveau macrobloc (dont la taille peu varié avec le type d'encodage utilisé).

Le *parser* lit et analyse syntaxiquement le flux en entrée. Les macroblocs sont intégralement redirigés vers le bloc *IDCT* (décodage) puis vers le composant *adder*. Les en-têtes sont dupliquée vers le filtre *MC* (pour *Motion Compensation*) et vers *IDCT*. Tous les macroblocs ne contiennent pas de vecteur de prédiction. Ces vecteurs servent à donner des références vers des parties d'images décodées précédemment qui seront additionnées par le *adder* avec les informations du macrobloc. *MC* répartit les calculs sur trois composants qui se chargent chacun d'une composante. Le composant *framestore* contient des données partagées par *MC* et *adder* qu'il convient de synchroniser grâce à un flux de contrôle (messages `resume` et `suspend`).

5.2 Plateformes

Notre décodeur compile pour trois plateformes matérielles différentes. La première d'entre elle est sans surprise Linux/Intel Multi-processeurs avec Hyper-Threading. Les deux machines à disposition avaient respectivement 2 threads physique / 1 processeur et 4 threads physiques / 2 processeurs. Nous utilisons la *glibc* et les *pthread*. Les deux autres ont été développées chez STMicroelectronics mais demeurent à l'état de prototype sur simulateur et FPGA. Il s'agit d'une plateforme MT (multi-threading qui correspond à l'hyper-threading d'Intel) et d'une plateforme SMP (multi-processeurs). L'avantage de ces deux derniers environnements est que l'on peut tester des applications sur n processeurs. La simulation est malheureusement très lente si l'on veut obtenir des mesures de performances (à noter que sur un simulateur, on n'introduit pas de biais dans la mesure, contrairement à du code intrusif comme avec *gprof*). Un runtime léger développé en interne implémente la librairie *pthread* et la transpose directement sur des primitives matérielles bas niveau.

5.3 Résultats

5.3.1 Aspects quantitatifs

Il convient de rester extrêmement prudent quant à l'analyse des performances de cette application. Le code n'est pas optimisé pour le matériel et de très nombreux défauts de cache d'instruction biaisent probablement les résultats.

Nous appellerons *version synchrone* de notre application, le programme compilé en mode mono-threadé sans utilisation de mutex (qui ne sont pas nécessaires dans ce cas) et sans scheduler (c.-à-d. que les réactions sont exécutées directement dès qu'un pattern est détecté dans la FSM).

Notre implémentation est basée sur le code de référence en C dont il reprend en grande partie le code fonctionnel. Après componentisation de l'application et refonte du code à l'aide des patterns de synchronisation, nous obtenons un programme 2.4 fois plus lent. Ceci se base sur une exécution sur simulateur ST231 pour une séquence vidéo de 32 images composées de 330 macroblocs chacunes. Nous considérons ici la version synchrone. Plusieurs raisons peuvent expliquer ce surcoût, notamment la gestion de la mémoire et l'allocation des messages sans utiliser de *pool*.

Nous mesurons ensuite le surcoût induit par l'introduction de l'exclusion mutuelle et du scheduler, ce qui nous amène à un code 1.7 fois plus lent que la version synchrone. Il serait judicieux de développer un scheduler basé sur des threads physiques et non des threads logiques pour s'affranchir du coût non négligeable de la librairie.

Des tests supplémentaires nous permettent d'évaluer le gain sur des plateformes MT et SMP. Pour chaque test, le code est compilé avec autant de threads logiques que de threads physiques à disposition. Nous obtenons les résultats du tableau 5.2. On remarque que la version sur 4 threads MT rattrape sur le surcoût de la couche de parallélisation.

	Coût de l'exécution (en millions de cycles)
Synchrone	900
1 thread	1500
4 threads MT	1250
4 threads SMP	880
7 threads MT	1580
7 threads SMP	?

TAB. 5.2 – Performances du décodeur MPEG-2 sur différentes architectures matérielles.

Soulignons le fait que nous sérialisons systématiquement les messages ce que ne fait pas le modèle threadé, mais qu'il devrait obligatoirement mettre en œuvre dans le cas de mémoires réparties.

5.3.2 Analyse qualitative

Dans cette approche, nous tirons tous les avantages de la programmation à base de composants. Les ADL permettent de donner une vue structurée de l'organisation architecturale d'une application. Ainsi, les évolutions pourront profiter de la réutilisation de code et il sera possible d'augmenter les fonctionnalités très simplement. Un avantage qui découle de ce paradigme est de pouvoir remplacer aisément les modules logiciels lourds en terme de calcul par des implantations matérielles dédiées.

Nous offrons une vue collaborative du système qui expose la manière dont les composants interagissent. Cela rend par exemple possible l'analyse des deadlocks potentiels. D'autre part, nous pourrions proposer une répartition des composants mais également une duplication de certains composants surchargés.

L'approche que nous défendons semble respecter les objectifs, à savoir proposer un paradigme de programmation simple pour les applications de streaming. Les composants nous autorise à transposer directement les schémas de principe de la norme. En outre, le mode « push » combiné à des patterns de synchronisation semble être l'abstraction la plus naturelle pour manipuler des flots de données.

Le programmeur focalise son travail sur les aspects algorithmiques de l'implémentation sans se soucier des aspects synchronisation et répartition. Par conséquent, le code produit est réutilisable.

Du point de vue des architectes, il est possible de revoir la configuration du système pour explorer différentes possibilités d'implantation et de répartition en modifiant seulement quelques lignes d'ADL[1].

5.4 Synthèse

L'implémentation d'un décodeur MPEG-2 se révèle être un bon exemple pour évaluer notre approche qui semble présenter un paradigme adéquate pour les applications de streaming parallèles. Le modèle composant apporte la souplesse architecturale nécessaire à un cadre industriel et notre modèle de synchronisation complète l'infrastructure par un moyen adapté pour gérer des flots de données.

Nous fournissons un cadre propice à l'automatisation de certains aspects qui pourrait, certes, être mieux optimisés manuellement. Cependant, si la maîtrise

[1] Nous pourrions aussi dire quelques clics si les outils proposaient une représentation graphique des fichiers de description d'architecture. Un logiciel supportant la précédente version de Fractal existait mais n'a pas encore été porté dans la chaîne d'outils Think.

de l'architecture et de la répartissabilité se confirme, nous pensons que l'on serait gagnant dans le cas d'applications hautement complexes tel qu'un encodeur H.264.

Pour conclure, rappelons que ce travail représente une première investigation qui visait principalement à démontrer la faisabilité de l'approche. Les aspects quantitatifs de l'évaluation sont à modérer par les nombreuses perspectives d'optimisation qui se sont ouvertes. Il conviendra de revenir de manière détaillée sur ce point une fois l'implémentation des optimisations terminée.

Partie

6

Conclusion

6.1 Bilan sur le projet

De cette étude, nous pouvons premièrement dégager un ensemble de points concluants.

- Cette première tentative de définition d'un modèle comportemental fournit une abstraction adéquate pour les applications streaming et permet de faciliter le parallélisme et la répartition. En outre, l'approche composant apporte toute la flexibilité dont ont besoin les industriels du secteur.
- Certaines pistes concernant l'optimisation du compilateur ont été testées avec succès et de nombreuses autres ont été envisagées pour de futurs développements.
- La réalisation d'un décodeur a pu être menée à bien grâce à notre modèle de programmation qui nous a en outre permis de l'exécuter de manière synchrone et parallèle.
- Enfin, nous pensons que cette abstraction facilite non seulement la programmation en lui retirant tous les aspects liés à la synchronisation, mais ouvre également la voie vers un véritable travail d'architecte consistant à répartir les composants sur des plateformes variées, et ce, en explorant aisément les différentes solutions.

Deux principaux points négatifs subsistent tout de même : d'une part, le modèle de calcul autorise trop facilement la création de deadlocks, et d'autre part, les performances de l'application de tests ne sont actuellement pas au rendez-vous.

6.2 Perspectives

De manière à palier les problèmes évoqués précédemment, il serait souhaitable de poser un formalisme plus clair du modèle de calcul de manière à supprimer les potentiels deadlocks. De même, quelques ajouts au langage sont désirables de façon à augmenter l'expressivité comme par exemple les patterns non linéaires et conditionnels.

Un point intéressant consisterait à étudier dans quelle mesure il serait envisageable, à partir de la description ADL et du code ThinkJoin, de détecter statiquement des problèmes comme des deadlocks ou des incohérences. Peut-être plus simplement pourrait-on proposer un moyen de vérifier des assertions, déclarées par le programmeur, notamment en lien avec l'ordre des messages. La reconfiguration dynamique des applications risque d'introduire une forte complexité vis-à-vis de la synchronisation. Encore une fois, un tel outil permettrait la vérification statique du comportement après reconfiguration.

La programmation événementielle n'étant ni triviale ni commune pour les développeurs d'applications de streaming, un bénéfice clair pourrait être tiré de la définition de patrons de conception pour la synchronisation. Nous en voulons pour exemple la manière dont a été gérée la barrière du composant MC dans le décodeur MPEG-2.

Comme cela a été réalisé dans StreamIt avec le *teleport messaging*, nous devrons nous demander jusqu'à quel point et à quel coût il est possible de faire transiter du flux de contrôle entre des composants non adjacents synchronisé avec les données. Ceci permettrait entre autres d'apporter une réponse au traitement des exceptions dans le flux multimédia.

Le modèle d'exécution nécessitera lui aussi des optimisations telle que la reconfiguration dynamique pour permuter d'une exécution parallèle vers une exécution synchrone lorsque ceci est plus rentable.

Il serait profitable de réaliser une série de tests pour valider les performances sur mémoires réparties. Nous partons de l'intuition que les gains par rapport au modèle threadé pourraient être plus remarquables que sur plateforme SMP, notamment à cause du coût de sérialisation des messages.

Enfin, l'investigation de la régulation du flux sur les canaux semble prometteuse. Des tests pourraient montrer l'incidence de diverses politiques de contrôle de flux dynamique sur l'exécution et sur l'utilisation de la mémoire.

Bibliographie

[1] Özcan (A. E.), Layaida (O.) et Stefani (J.-B.), « A component-based approach for MPSoC SW design : Experience with OS customization for H.264 decoding. », dans *ESTImedia*, p. 95–100, 2005.

[2] Lee (E. A.), « The problem with threads », *Computer*, vol. 39, n° 5, 2006, p. 33–42.

[3] Randall (K. H.), *Cilk : Efficient Multithreaded Computing*. Thèse de doctorat, MIT Department of Electrical Engineering and Computer Science, 1998.

[4] Galilée (F.), Roch (J.-L.), Cavalheiro (G. G. H.) et Doreille (M.), « Athapascan-1 : On-line building data flow graph in a parallel language », dans *PACT '98 : Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, p. 88, Washington, DC, USA, 1998. IEEE Computer Society.

[5] Benveniste (A.), Caspi (P.), Edwards (S.) *et al.*, « The synchronous languages twelve years later », *Proc. of the IEEE, Special issue on embedded systems*, vol. 91, n° 1, jan 2003, p. 64–83.

[6] Thies (W.), Karczmarek (M.) et Amarasinghe (S. P.), « StreamIt : A language for streaming applications », dans *CC '02 : Proceedings of the 11th International Conference on Compiler Construction*, p. 179–196, London, UK, 2002. Springer-Verlag.

[7] Sermulins (J.), Thies (W.), Rabbah (R.) et Amarasinghe (S.), « Cache aware optimization of stream programs », dans *LCTES '05 : Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and*

tools for embedded systems, p. 115–126, New York, NY, USA, 2005. ACM Press.

[8] Thies (W.), Karczmarek (M.), Sermulins (J.) *et al.*, « Teleport messaging for distributed stream programs », dans *PPoPP '05 : Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, p. 224–235, New York, NY, USA, 2005. ACM Press.

[9] « StreamIt cookbook ». 2004.

[10] Drake (M.), Hoffmann (H.), Rabbah (R.) et Amarasinghe (S.), « MPEG-2 decoding in StreamIt », dans *IPDPS*, 2006.

[11] Consel (C.), Hamdi (H.), Réveillère (L.) *et al.*, « Spidle : a DSL approach to specifying streaming applications », dans *GPCE '03 : Proceedings of the second international conference on Generative programming and component engineering*, p. 1–17, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[12] Fournet (C.) et Gonthier (G.), « The reflexive CHAM and the join-calculus », dans *POPL '96 : Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 372–385, New York, NY, USA, 1996. ACM Press.

[13] Itzstein (G. S.) et Jasiunas (M.), *On Implementing High Level Concurrency in Java*, vol. 2823/2003, p. 151–165. Springer Berlin / Heidelberg, 2003.

[14] Benton (N.), Cardelli (L.) et Fournet (C.), « Modern concurrency abstractions for C# », *ACM Trans. Program. Lang. Syst.*, vol. 26, n° 5, 2004, p. 769–804.

[15] Szyperski (C.), *Component software : beyond object-oriented programming.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[16] Aoyama (M.), « New age of software development : How component-based software engineering changes the way of software development ? », dans *International Workshop on Component-Based Software Engineering*, 1998.

[17] Bruneton (E.), Coupaye (T.), Leclercq (M.) *et al.*, « The fractal component model and its support in Java », *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, vol. 36, n° 11-12, 2006, p. 1257–1284.

[18] Krakowiak (S.). « Constructions d'applications parallèles et réparties, cours de Master Recherche 2 ». `http://sardes.inrialpes.fr/~krakowia/`, 2006.

[19] Fassino (J.-P.), Stefani (J.-B.), Lawall (J. L.) et Muller (G.), « Think : A software framework for component-based operating system kernels », dans *Proceedings of the General Track : 2002 USENIX Annual Technical Conference*, p. 73–86, Berkeley, CA, USA, 2002. USENIX Association.

[20] Guillaume (P.), Leclercq (M.), Özcan (A. E.) et Stefani (J.-B.). « Modular and retargettable code generation for an extensible ADL ». 2006.

[21] Fessant (F. L.) et Maranget (L.), « Compiling join-patterns », *Electronic Notes in Computer Science*, vol. 16, nº 2, 1998.

[22] Berger (E. D.), Zorn (B. G.) et McKinley (K. S.), « Reconsidering custom memory allocation », dans *OOPSLA '02 : Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, p. 1–12, New York, NY, USA, 2002. ACM Press.

[23] Rosengren (K.), « Modelling and implementation of an MPEG-2 video decoder using a GALS design path ». Rapport de DÉA, University of Linköping, 2006.

[24] « MPEG-2 specification ». google : "ISO/IEC 13818" ou `http://neuron2.net/library/mpeg2/neuron2.net/library/mpeg2/iso13818-2.pdf`.

[25] « MPEG-2 video codec reference C code ». `http://www.mpeg.org/MPEG/MSSG/`.

[26] Saha (K.), Mathur (M.) et Maiti (S.), « Parallelization of MPEG-2 video decoder in shared memory systems ». Rapport technique, STMicroelectronics, 2004.

[27] Chen (H.), Li (K.) et Wei (B.), « A parallel ultra-high resolution MPEG-2 video decoder for PC cluster based tiled display systems », dans *IPDPS '02 : Proceedings of the 16th International Parallel and Distributed Processing Symposium*, p. 30, Washington, DC, USA, 2002. IEEE Computer Society.

[28] Bilas (A.), Fritts (J.) et Singh (J. P.), « Real-time parallel MPEG-2 decoding in software », dans *IPPS '97 : Proceedings of the 11th International Symposium on Parallel Processing*, p. 197–203, Washington, DC, USA, 1997. IEEE Computer Society.

[29] Tanguay (D.), Gelb (D.) et Baker (H. H.), « Nizza : A framework for developing real-time streaming multimedia applications ». Rapport technique, HP Laboratories Palo Alto, 2004.

[30] Özcan (A. E.), Jean (S.) et Stefani (J.-B.), « Bringing ease and adaptability to MPSoC software design : A component-based approach. », dans *CASSIS*, p. 118–137, 2005.

Table des figures

2.1 Exemple d'une application de streaming. 10
2.2 Schéma de l'application exemple. 17

3.1 Exemple d'application de streaming minimale comportant un parseur P, un décodeur D et une commande C. 25
3.2 Vue d'un composant actif composite avec sa FSM et son sous-composant implémentant les réactions. 26

4.1 Grammaire BNF du langage ThinkJoin. 30
4.2 Exemple d'exécution de l'algorithme 4.1. Au dessous des quatre interfaces est représenté le statut. 32
4.3 Exemple d'exécution de l'algorithme 4.1 (suite). Sous les quatre interfaces figurent le statut (rectangle continu) et le pattern détecté (rectangle en pointillés). 34
4.4 Exemple d'architecture pouvant mener à des deadlocks. 35
4.5 Intégration du loader Join dans la chaîne de load de Think ADL. . 36
4.6 Plugin de génération de code pour le langage Think Join. 37
4.7 Diagramme UML de la librairie FSM. 41
4.8 Enregistrement et exécution d'une réaction par le scheduler suite à la réception de deux messages. 42

5.1 Architecture du décodeur MPEG-2. 47

Liste des tableaux

2.1 Synthèse et évaluation des différents langages pour chaque critère sur une échelle de 1 (non adapté) à 4 (totalement adéquate). 19

5.1 Taille des données du flux MPEG-2. 48
5.2 Performances du décodeur MPEG-2 sur différentes architectures matérielles. 49

Listings

2.1 Code StreamIt pour l'application exemple. 16
2.2 Code Esterel pour l'application exemple. 17
2.3 Code Polyphonic C# pour l'application exemple. 18
3.1 Interfaces utilisée dans le composant exemple. 27
3.2 Description ADL du composant exemple. 28
3.3 Définition du fichier de règles. 28
3.4 Implémentation en C de la réaction. 28
4.1 Algorithme décrivant la réception d'un message dans le langage ThinkJoin. 33
4.2 Algorithme optimisé décrivant la réception d'un message dans le langage ThinkJoin. 39
4.3 Algorithme optimisé décrivant la réception d'un message dans le langage ThinkJoin (suite). 40