Projet de Fin d'Année 21 février 2008

Cahier des charges

Développement logiciel dans le cadre de la génération de processeurs très hautes-performances

Client : Bertrand LE GAL

Encadrant pédagogique : Philippe DUCHON

BENITTO Adnane
BUISSON Rémi
EL AFRIT Mohamed Amine
GATTI Stéphanie
PORTRAT Benoît
RATOUIT Thomas
STORDEUR Ludovic

Table des matières

Table des figures

1 Présentation du projet

1.1 Contexte

Dans le laboratoire IMS[1] , l'équipe "Conception de systèmes numériques" dans laquelle travaille le client Monsieur Bertrand Le Gal, développe actuellement un outil générant des processeurs très haute performance. Un cas d'utilisation de l'outil est le suivant : en entrée, l'utilisateur vient écrire du code source (Fichier MatLab[2]) MatLab[3] , qui sera ensuite traduit via un parser Java en un ou plusieurs graphes. Celui-ci sera ensuite utilisé dans la conception du processeur que l'on souhaitait produire. Le passage du graphe au processeur est un processus automatisé. Il s'effectue par le biais de la génération d'un fichier VHDL[4] décrivant la structure matérielle, qui servira ensuite à créer des portes logiques câblées contenues dans un FPGA[5] afin de relier les différents éléments de la mémoire du processeur.

L'outil, nommé "GraphLab", offre des étapes de conception identiques aux phases de compilation logicielle, la différence étant que la finalité est d'obtenir un processeur avec des performances optimisées. En effet, l'équipe de développement de l'outil est plutôt orientée vers la fabrication de processeurs selon les besoins et les demandes, c'est-à-dire avec des fonctionnalités pouvant varier à chaque conception. Chaque processeur est dédié à un calcul spécifique. Par conséquent, le processeur est obtenu après les différentes phases suivantes :

- écriture du code source MatLab,
- analyse de ce code via un parser Java,
- traduction en un ou plusieurs graphes,
- génération d'un fichier VHDL,
- création de portes logiques câblées, et
- création de liens entre les différents éléments de la mémoire pour parvenir à l'obtention d'un processeur,

doit être un processeur très-haute-performance, c'est-à-dire dont les caractéristiques sont optimisées et adaptées au cas développé.

La figure 1.1 représente l'enchaînement de ces étapes.

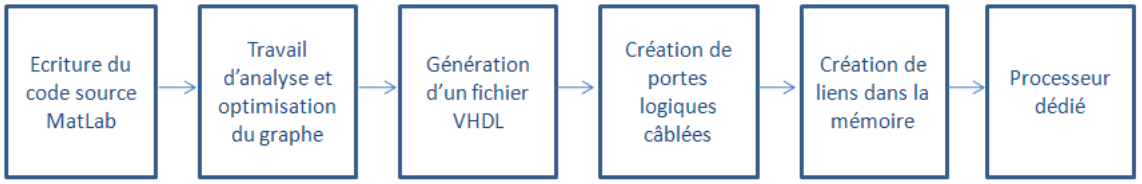


Fig. 1.1 – Les différentes étapes de développement dans lesquelles s'insère l'outil GraphLab

[1] cf. Glossaire
[2] cf. Glossaire
[3] cf. Glossaire
[4] cf. Glossaire
[5] cf. Glossaire

Le point de départ de l'outil est que l'utilisateur dispose d'un code MatLab décrivant des expressions arithmétiques. Un tel code présent en entrée représente les instructions que devra réaliser le processeur dédié final. Par exemple, si l'expression arithmétique **a+b+c** est passée en entrée, cela signifiera que le processeur dédié final possèdera, après les phases de traitement, une ressource de calcul de type additionneur.

De plus, il faut dissocier l'obtention d'un graphe de sa visualisation.

Après analyse du code MatLab, il y a obtention d'un modèle formel de calcul qui n'est pas directement visualisable. Pour l'observer à l'écran, il faudrait faire appel à un outil de visualisation de graphe, comme par exemple, la fonction *plot* sous MatLab.

Pour réaliser la traduction du code MatLab en architecture matérielle, deux fonctionnalités différentes ont été développées dans le cadre d'un autre projet l'année dernière :
- un parser de code source MatLab sommaire,
- une possibilité de mise à jour automatique pour l'utilisateur.

Notre travail consiste à faire évoluer l'outil en ajoutant de nouvelles propriétés et en optimisant certains points, tout en respectant la nature et la cohérence des éléments déjà développés.

Les besoins du client se situent à trois niveaux de l'outil :
- au niveau du parser MatLab,
- au niveau du graphe obtenu après analyse, et
- au niveau de l'interface visuelle.

Plus précisément, il s'agira donc de reprendre le parser MatLab en analysant toute la sémantique, et également en ajoutant la possibilité d'optimiser le graphe obtenu, par détection de simplifications via les propriétés des opérateurs usuels. Ces propriétés pourront être, par exemple, l'élément neutre pour un opérateur, ce qui conduira à différentes optimisations. Par exemple, en considérant une variable a et l'opération d'addition, l'élément neutre étant le 0, $a + 0$ sera optimisé en a, et il en est de même pour l'opération de multiplication où l'élément neutre est 1 : l'expression $a * 1$ sera optimisée en a.

En considérant par exemple l'expression arithmétique $y = a * 0 + b * 1 + 0 + c$, le graphe correspondant est visualisable sur la figure 1.2, ainsi que le graphe optimisé via des simplifications mathématiques.

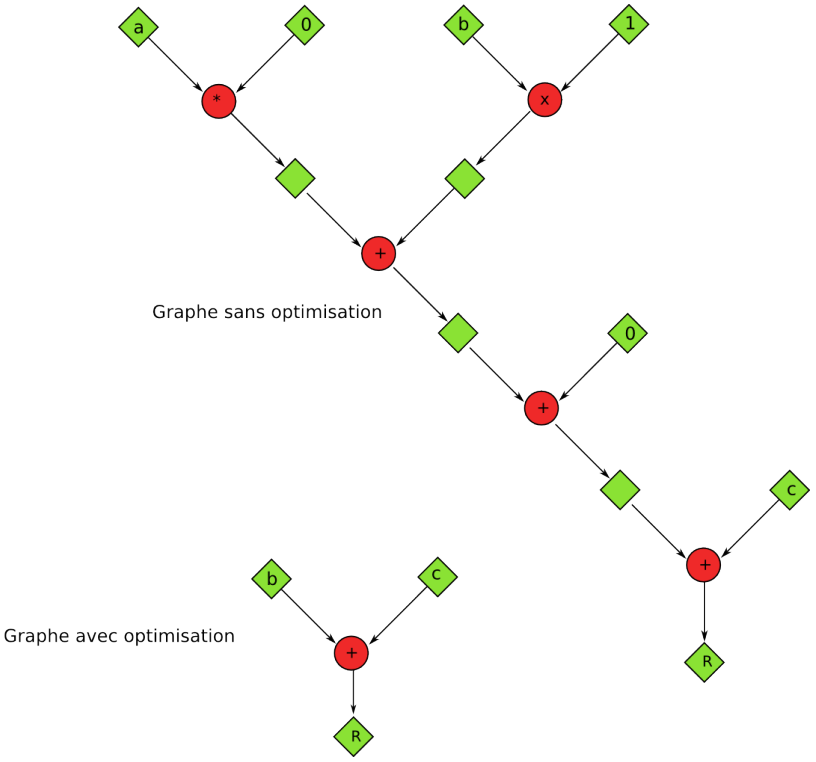


Fig. 1.2 – Graphe correspondant à l'équation $y = a * 0 + b * 1 + 0 + c$ ainsi que le graphe optimisé, correspondant à l'expression arithmétique optimisée $y = b + c$

Nous voyons clairement sur ces figures l'intérêt d'effectuer de telles optimisations sur le graphe.

Nous devrons également considérer le problème de l'ajout ou l'appel de nouveaux opérateurs au langage MatLab ainsi que de propriétés les caractérisant. Pour cela, nous pourrons être amenés à définir un nouveau langage de description de ceux-ci. Notre intervention consistera également en l'optimisation du graphe obtenu, et ce, en passant de la forme factorisée d'un graphe à sa forme développée et inversement, grâce aux propriétés que possèdent les opérateurs, telle l'associativité pour l'opération d'addition.

Considérons par exemple l'expression arithmétique factorisée $y = (a + b) * c$. Cette expression peut être développée en l'expression **y=(a*c)+(b*c)**. La figure 1.3 montre les graphes obtenus, selon que l'on considère la forme factorisée ou développée de cette expression.

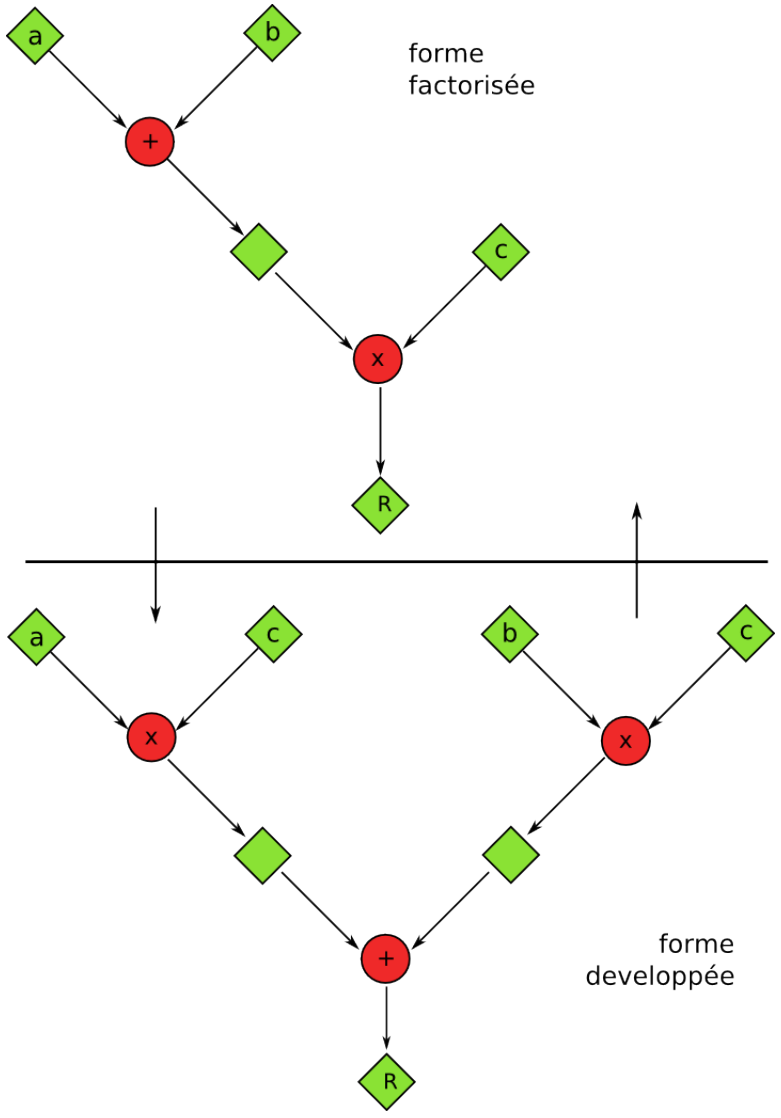


Fig. 1.3 – Formes factorisée et développée du graphe

Nous voyons sur ces graphes, que le nombre d'opérations et l'ordre d'exécution des opérations n'est pas le même.

Dans la forme factorisée, deux opérations sont effectuées : l'addition de a et b dont le résultat est mis dans une variable intermédiaire, puis la multiplication de ce résultat intermédiaire par c.

Dans la forme développée, trois opérations sont effectuées : l'addition de a et c dont le résultat est mis dans une variable intermédiaire, l'addition de b et c dont le résultat est mis dans une autre variable intermédiaire, puis l'addition de ces deux résultats intermédiaires.

Du fait que, dans le graphe représentant la forme développée de l'équation $y = (a+b)*c$, trois opérations sont effectuées, alors que deux seulement sont effectuées dans celui représentant la forme factorisée de cette équation, nous ne pouvons pas en déduire, a priori, l'impact sur la vitesse d'exécution du calcul. Tout dépendra des fonctionnalités possédées par le processeur. Les caractéristiques des graphes seront différentes selon le circuit sur lequel elles seront implantées.

En fonction du cas d'utilisation, il est préférable d'implanter un graphe plutôt qu'un autre.

En effet, si nous considérons l'expression arithmétique suivante : $y = a + b + c + d$, le graphe obtenu serait celui en figure 1.4.

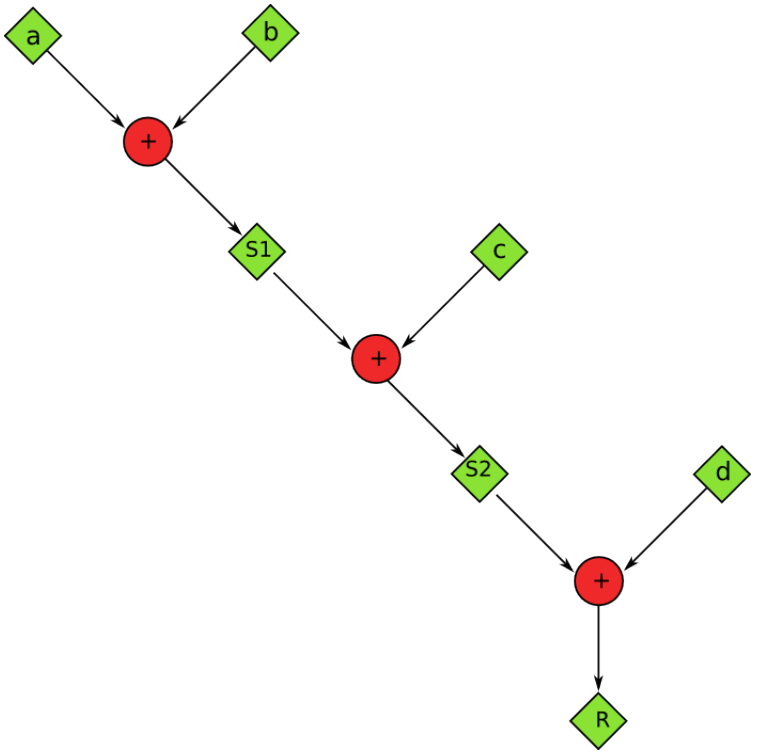


Fig. 1.4 – Graphe représentant l'équation $y = a + b + c + d$

Or en utilisant la propriété d'associativité de l'opérateurs d'addition, l'expression précédente est équivalente à $y = (a + b) + (c + d)$, ce qui induit le graphe présenté en figure 1.5.

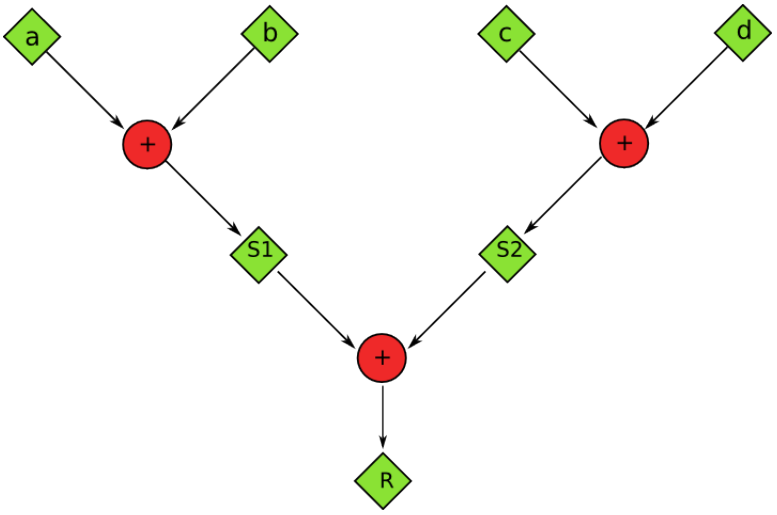


Fig. 1.5 – Graphe représentant l'équation $y = (a + b) + (c + d)$

Dans le cas où le processeur dédié final possède deux additionneurs ou plus, le graphe représentant l'équation $y = (a + b) + (c + d)$ sera utilisé car il induira une vitesse d'exécution du processeur supérieure.

En effet, le graphe 1.4 décrit trois opérations d'addition successives, qui ne pourront donc pas être réalisées en même temps : il faudra donc trois cycles d'horloge pour obtenir le résultat attendu. A l'inverse, le graphe 1.5 présente seulement deux opérations d'addition pouvant s'effectuer dans le même temps (dans le cas considéré où le processeur possède deux additionneurs au moins), ainsi qu'une autre opération d'addition :

il faudra donc seulement deux cycles d'horloge pour obtenir un tel résultat.

Mais ces informations sont à modérer par le fait que l'équation $y = (a + b) + (c + d)$ fait intervenir deux variables intermédiaires au même moment, tandis que l'équation $y = a + b + c + d$ en fait intervenir deux mais à des dates différentes. En effet, le graphe 1.5 présente deux variables intermédiaires $S1$ et $S2$ qui ne partagent pas le même registre, du fait que leurs durées de vie sont recouvrables. A l'inverse, le graphe 1.4 présente deux variables intermédiaires qui n'en sont pas vraiment, car elles peuvent partager le même registre, du fait que leurs durées de vie ne se recouvrent pas.

1.2 Organisation du projet

Le projet de travail sur le thème "Développement logiciel dans le cadre de la génération de processeurs très haute-performance" s'articule autour de la maîtrise d'ouvrage (MOA[6]) et de la maîtrise d'oeuvre (MOE[7]).

La maîtrise d'ouvrage correspond au client, à savoir Monsieur Bertrand LE GAL. Le client a émis des besoins concernant l'outil GraphLab et souhaité voir s'articuler un projet ; ce dernier ayant pour but de faire évoluer l'outil via le développement de certaines fonctionnalités, comme l'analyse syntaxique du parser MatLab, et des améliorations, comme l'optimisation du graphe généré en sortie.

La maîtrise d'oeuvre est constituée par les différents membres de notre groupe, à savoir :

BENITTO Adnane
BUISSON Rémi
EL AFRIT Mohamed Amine
GATTI Stéphanie
PORTRAT Benoît
RATOUIT Thomas
STORDEUR Ludovic.

Notre tâche sera de développer, parmi les différents besoins émis par le client, les fonctionnalités principales retenues selon le temps et les moyens dont nous disposons. Celles-ci auront été déterminées en accord avec lui. Une série de tests sera également effectuée lors du développement des différents modules.

Notre travail sera effectué sous la supervision d'un encadrant technique : Monsieur Philippe Duchon.

1.3 Environnement du projet

1.3.1 Existant fonctionnel

Ce logiciel contient des fonctionnalités qui sont déjà implantées :

- Un parser basique de code *MatLab* permettant de transformer une suite d'instructions *MatLab* en graphe.
- Plusieurs fonctionnalités permettant de manipuler les graphes : sauvegarder, transformer, afficher des informations sur celui-ci, «««< .mine l'ordonnancement des noeuds du graphe, au plus tôt *ASAP* ou au plus tard *ALAP*. ASAP[8] ASAP[9] ======= l'ordonnancement des noeuds du graphe, au plus tôt ASAP[10] ou au plus tard ALAP[11] . »»»> .r158
- Un système de mise à jour automatique de l'outil et des plugins.

[6] cf. Glossaire
[7] cf. Glossaire
[8] cf. Glossaire
[9] cf. Glossaire
[10] cf. Glossaire
[11] cf. Glossaire

– Et encore un certain nombre d'opérations élémentaires.

1.3.2 Existant technique

L'ensemble du code source du programme est en *Java*.

Le noyau

Le client a choisi d'organiser son logiciel autour d'un noyau le plus petit possible, autour duquel viennent se greffer des modules qui ajoutent de nouvelles fonctionnalités. Ceci permet de garantir au maximum l'évolutivité du logiciel tout en limitant au maximum les régressions. En effet, si un module montre des dysfonctionnements lors de son exécution, il suffit de désactiver le chargement de celui-ci sans mettre en péril la stabilité du logiciel entier.

De plus, cela permet à des développeurs qui n'ont pas de nécessité de connaître le coeur de l'outil de pouvoir tout de même ajouter des fonctionnalités par l'intermédiaire des modules.

Enfin, cette conception modulaire offre un avantage du point de vue juridique. En effet, les sources du noyau sont placées sous une licence qui oblige la diffusion du code source de ce dernier. Au contraire, les modules sont plus permissifs à ce niveau dans la mesure où seule l'archive **jar** résultant de la compilation du module est nécessaire pour pouvoir utiliser ce dernier. Autrement dit, cela permet à un développeur d'ajouter une fonctionnalité au logiciel au travers d'un module sans être obligé de fournir les sources de son travail.

Ainsi, le noyau se contente d'implanter les fonctionnalités suivantes :

– La détection et le chargement des modules. Concrètement, ces modules se présentent sous la forme d'une archive **jar** et doivent être placés dans un dossier spécifique afin d'être détectés.
– Une structure de données générique permettant de représenter des graphes. Celle-ci sera notamment utilisée par le parser MatLab pour créer le graphe résultant de l'analyse d'une fonction MatLab ou bien lors de l'affichage des graphes.
– La console d'interaction entre l'utilisateur et le logiciel. Il est à noter qu'en elle-même, la console n'implante aucune commande dans la mesure où ces dernières sont développées sous forme de modules.

Les sources du noyau sont localisées dans le répertoire **GraphLab** à partir de la racine des sources. Au vu de la nature des fonctionnalités que nous allons développer, nous ne devrions pas avoir à modifier le code du noyau.

Les modules

Comme expliqué précédemment, les modules, que l'on peut également appeler des *plugins*, sont développés en marge du noyau évoqué ci-dessus. Ils viennent greffer sur ce dernier les fonctionnalités qui sont directement utilisables par l'utilisateur. Parmi elles, on peut citer :

– L'ensemble des commandes utilisables dans la console (*ls*, *plot*, ...).
– Le parser MatLab.

Ainsi, les fonctionnalités liées à l'optimisation des graphes que nous allons développer seront implantées en tant que modules à cet endroit.

Actuellement, les sources des modules sont localisées dans deux emplacements :

Les sources des modules encore inclus dans le corps de l'outil sont localisées dans le répertoire **GraphLab/src/modules**. A terme, il conviendra de sortir ces modules de l'arborescence du noyau.

Les sources des modules sortis du coeur de l'outil sont localisées dans le répertoire **PlugIns**. Les sources de chaque plugin sont situées dans un sous-répertoire du nom du plugin. Une fois un module développé, on peut utiliser celui-ci en en faisant une archive **jar** et en déplaçant cette dernière dans un dossier spécifique qui se trouve dans les sources du noyau.

Du point de vue des sources, les modules sont complètement dissociés les uns des autres. Cela permet entre autre à des développeurs de créer en parallèle leur propre module sans devoir à un moment donné procéder à une fusion des sources. Ils auront juste à déposer l'archive **jar**, contenant les classes compilées de leur module, dans le dossier prévu à cet effet.

Diagramme de classe du noyau

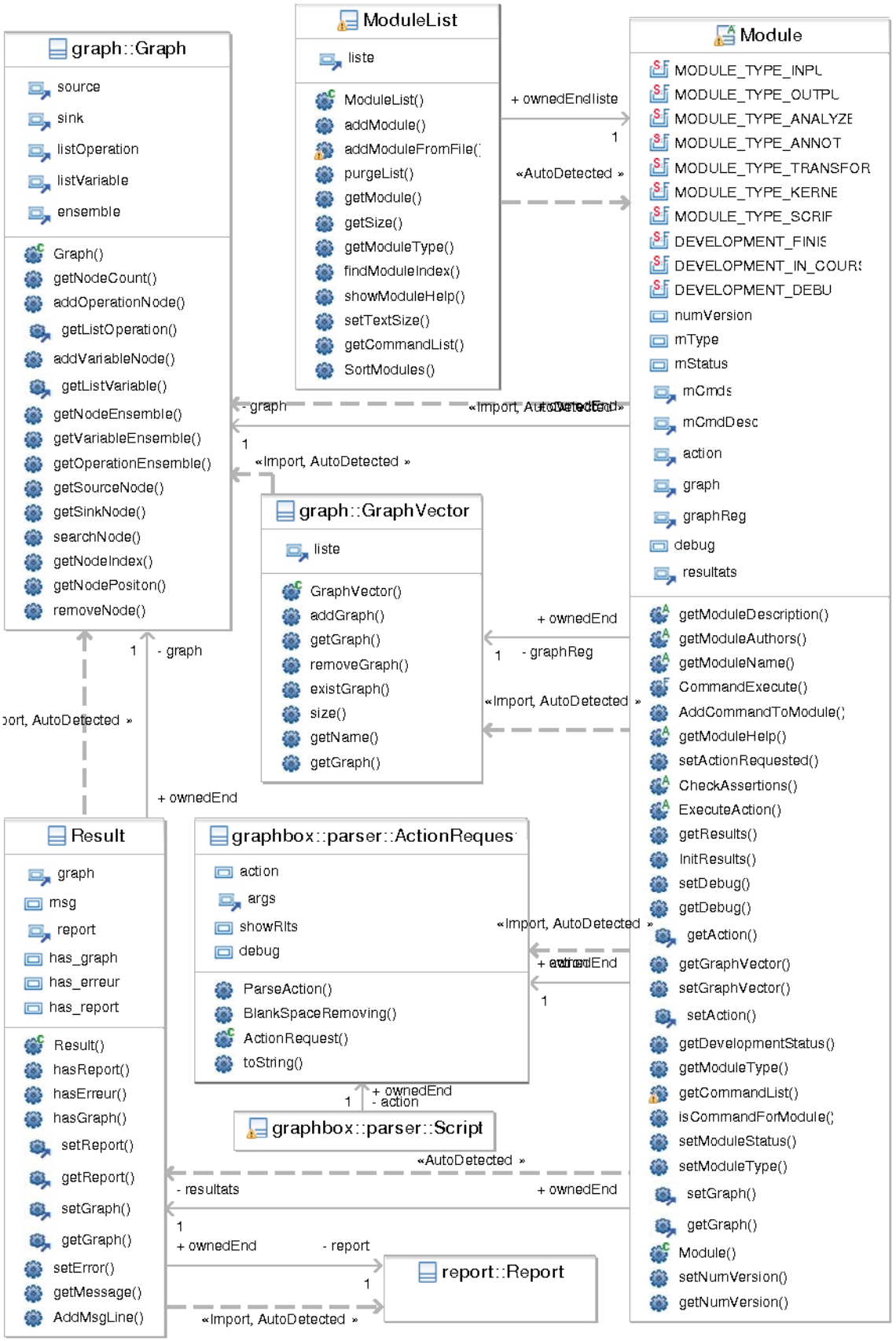


Fig. 1.6 – Diagramme des classes du noyau

Description du Diagramme de classe 1.6

La classe *Module* est celle dont toutes les autres classes héritent ; elle est donc très importante. De plus, elle peut contenir des informations importantes sur l'avancée du développement du code avec des notions telles que *fini*, *en cours de développement*, *en cours de débogage*.

Dans la classe *Result* se trouvent :
- Le graphe résultant de la transformation,
- Une série de points,
- Les textes affichés à l'écran.

En lançant l'outil, tous les modules présents dans cette classe se rajoutent automatiquement dans la liste.

La classe *GraphVector* est le vecteur des graphes secondaires. Ce sont les graphes correspondant aux fonctions auxiliaires rencontrées dans le graphe principal. Le code de ces fonctions auxiliaires se situe dans un fichier d'extension *.m*, portant le nom de la fonction.

1.3.3 Description des besoins techniques

Le langage Java

La programmation de cette application sera réalisée en langage Java pour maintes raisons :
- la partie existante de l'outil est déjà réalisée en *Java* et pour éviter tout conflit et problème provenant de l'utilisation de différents langages de programmation dans la même application, nous avons décidé de garder le même langage,
- le langage *Java* est compatible avec toute plateforme, c'est-à-dire un programme développé en *Java* peut s'exécuter sur n'importe quelle machine disposant d'une machine virtuelle *Java* (appelée *J.V.M.*). Ainsi, un applicatif développé en *Java* répond aux besoins d'un marché bien plus important que s'il avait été développé dans un autre langage objet sans recompilation, c'est-à-dire sans distribution binaire particulière.
  Pour programmer cette application, nous avons besoin d'un environnement *JDK* version 1.6,
- le langage *Java* est assez fiable. Il dispose d'un ramasse-miettes (*garbage collector*) qui évite les fuites mémoire.

Le serveur SVN SubVersioN

Subversion est un logiciel de gestion des versions des fichiers d'un projet. La gestion de version consiste à maintenir l'ensemble des versions d'un logiciel.

Subversion s'inscrit comme un outil indispensable dans le processus de développement de grands projets. Parmi les fonctionnalités proposées, l'archivage de documents et de code source se révèle très utile car il permet notamment de synchroniser le travail effectué par les différents développeurs, de garder une trace sur l'évolution de leurs produits et de servir de support de communication entre les acteurs du projet.

SVN est un dépôt sur lequel différentes actions peuvent être effectuées, par le biais de commandes listées dans l'aide de SVN.

Ces actions sont de différentes sortes. Il est possible d'ajouter un fichier (ou un répertoire) et de mettre à jour un fichier déjà présent sur le dépôt. Ainsi, le SVN permettra l'archivage des fichiers, et grâce à son système de versions, permettra d'afficher des informations quant aux modifications qui ont eu lieu sur le dépôt, et ce, pour chaque version, toute modification dans le SVN entraînant la création d'une nouvelle version.

Il est également possible de laisser des commentaires lors d'une modification, ce qui se révèle très utile pour les autres développeurs car ces données pourront les informer sur l'avancée des travaux.

Le principal atout du logiciel SVN est de permettre un travail collaboratif en offrant une centralisation des sources afin de les rendre disponibles à l'ensemble des acteurs d'un projet, tout en assurant un suivi en conservant un historique des différents fichiers.

Au vu de ces informations, nous avons mis en place un *SVN* pour faciliter les mises à jour des sources et pour l'organisation de notre travail.

Environnement de développement NetBeans

L'environnement de développement n'a pas été imposé par le client. Le choix de l'environnement était libre.

Nous avons choisi de travailler avec *NetBeans* car cet I.D.E.[12] possède plusieurs fonctionnalités très utiles au développeur :

- une large variété de modules est disponible pour supporter plusieurs langages de programmation dont *Java*, qui est le langage qui sera utilisé lors du développement de l'outil,
- NetBeans est très intéressant pour le travail avec le langage Java car cet I.D.E. est développé par *Sun* qui est la maison-mère du Java. Ainsi, toute modification dans le J.D.K[13] . apparaît beaucoup plus rapidement sur NetBeans que sous Eclipse, son principal concurrent. Les nouvelles fonctionnalités développées pour le J.D.K. sont intégrées avant même que la version finale ne soit mise en circulation, le J.D.K. étant un logiciel édité par Sun pour le développement d'applications en Java,
- de nombreuses fonctionnalités y sont intégrées,
- NetBeans reconnait des balises. Par exemple, l'utilisation de la balise "TODO" par un développeur, permet ensuite à l'I.D.E. de rechercher dans tout le source les balises "TODO". Ceci permettra de retrouver rapidement l'endroit du code où il restait encore du travail à effectuer. Il existe de même des balises "BUG", qui permettent de rappeler où se situe le code à déboguer,
- NetBeans offre aux programmeurs une grande facilité de développement, en prenant en charge des difficultés et des parties secondaires du projet : il agit notamment comme un éditeur de textes évolué. Il inclue en particulier les propriétés de complétion de code, de refactorisation[14] , des Java Hints[15] , un navigateur XML, ...

[12] cf. Glossaire
[13] cf. Glossaire
[14] cf. Glossaire
[15] cf. Glossaire

2 Description du projet

2.1 Le périmètre du projet

Le développement de l'outil générique GraphLab permettra essentiellement une réalisation de synthèse d'architecture, ou une compilation logicielle transposée en vue d'un développement matériel. L'assise de cet outil ainsi que d'autres fonctionnalités (un module parser *MatLab* rudimentaire et un module de mise à jour automatique) sont déjà implantés. Notre apport consiste en la mise en oeuvre de davantage de fonctionnalités et en le développement de quelques unes déjà présentes :

- Développement en Java du parser existant de code source *Matlab*.
- Transformation de graphes et optimisations mathématiques.

Le premier point traite d'une fonctionnalité qui permet de transformer un code source en graphe. Notre intervention vise à la faire évoluer et à accroître la sémantique utilisable dans les codes d'entrée.

Le second point a pour but de proposer, grâce à des techniques mathématiques, des méthodes permettant de simplifier les graphes.

Ainsi, le périmètre du projet s'inscrit dans le cadre général du développement de *GraphLab*, et l'ensemble, dans un processus de développement logiciel.

La figure 2.1 présente le diagramme de périmètre. Nous y représentons les différents acteurs qui interagissent avec le système étudié. Dans ce diagramme, l'utilisateur doit jouir de toutes les fonctionnalités offertes par ce logiciel, c'est-à-dire du parser *MatLab* et des fonctionnalités existantes de manipulation de graphe.

2.2 Description générale du projet

Le projet s'articule autour de trois axes principaux :
- le parser MatLab,
- l'algorithme d'optimisation de graphe,
- le visuel : cette partie consisterait à mettre à jour l'ergonomie de l'outil.

Ce projet mixe un nombre important de connaissances que nous avons acquises dans les domaines de la conception logicielle. En effet, il permet d'accroître notre expérience en algorithmique, en compilation et en I.H.M.[1] . Pour le module de visualisation des courbes, pour les membres du groupes, nous pouvons même parler de découverte car au cours des projets divers que nous avons eu à réaliser, il était question de développement d'outils "en mode console".

Ceux-ci était développés de manière procédurale. Nous nous proposons ici de développer un logiciel objet, méthode intrinsèquement liée à Java. L'U.M.L.[2] nous fournit une méthode de conception d'architectures logicielles objet.

[1] cf. Glossaire
[2] cf. Glossaire

Ce langage de modélisation nous fournit des diagrammes qui informent de manière claire et précise sur la structuration d'une application. Ainsi, nous nous proposons de décrire les diagrammes de Paquetage[3] , de cas d'utilisation et de séquence associés à ce projet.

En figure 2.1 est présenté un diagramme de paquetage. Il présente les dépendances entre paquetages, c'est-à-dire les dépendances entre ensembles de définitions. Le symbole $\oplus$ signifie "contient". Ici, le paquetage **GraphLab** contient les classes **Optimiseur de graphes**, **Parser MatLab**, **Console GraphLab** et **Graphe**. L'utilisateur n'interagit qu'avec la console de GraphLab. Chacune de ces classes est représentée par un ensemble de trois champs. Le premier indique le nom de la classe. Dans le second sont listés les attributs de la classe (ici, ce champs est laissé vide pour simplifier la lecture). Le troisième liste les méthodes de la classe. Le + devant ces méthodes est un opérateur de portée qui indique qu'elles sont publiques, c'est-à-dire accessibles de toute autre classe.

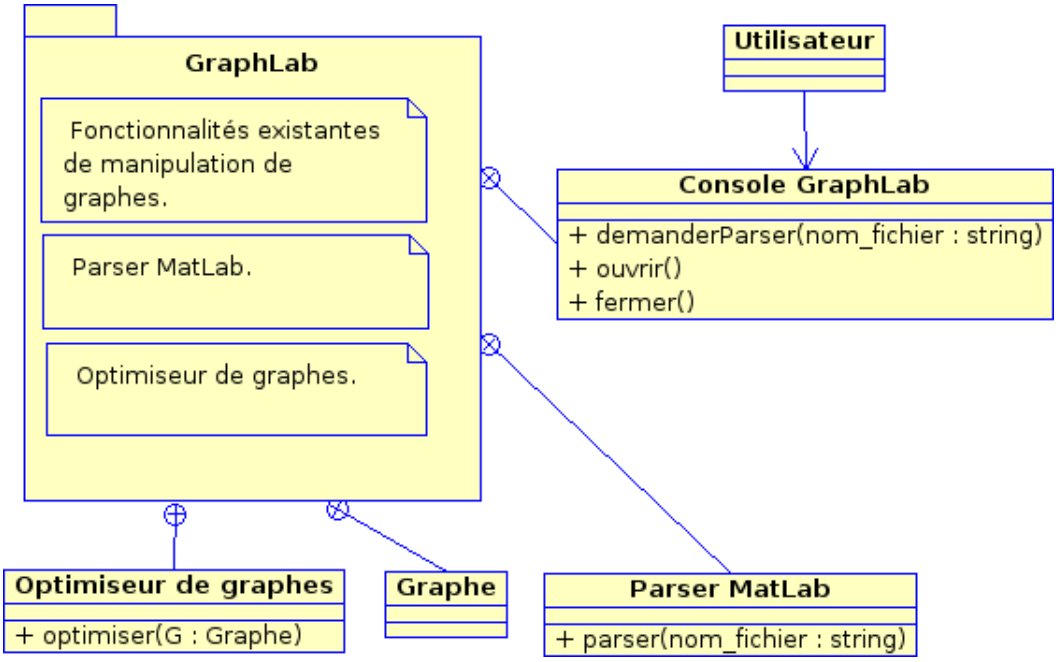


Fig. 2.1 – Diagramme de périmètre (ou de paquetage)

La classe correspondant à la console GraphLab possède les méthodes ouvrir(), fermer() et demandeParser(). La première permet d'ouvrir une console, c'est-à-dire une fenêtre dans laquelle s'affiche un prompt. La seconde a pour fonction de fermer la console ouverte précédemment. La troisième prend en paramètre un nom de fichier et demande à parser le code MatLab contenu dans ce fichier. En sortie de cette dernière fonction, un graphe est obtenu. Cette analyse du code MatLab sera effectuée par le biais de la méthode parser() contenu dans la classe **Parser MatLab**.

Les deux fonctionnalités contenues dans le paquetage **GraphLab** qui concernent l'analyse du code MatLab déjà présente dans l'outil avant notre valeur ajoutée, à savoir l'utilitaire de manipulation de graphes et le parser MatLab, sont décrites au travers des classes **Console GraphLab**, **Parser MatLab** et **Graphe**.

La troisième fonctionnalité, l'optimisation de graphes, est réalisée via des appels à une instance de la classe **Optimiseur de graphes**, et plus précisément, via l'appel à la méthode optimiser() qui prend un graphe en entrée sur lequel elle applique des techniques d'optimisation.

Bien évidemment, ce diagramme de périmètre n'est pas exhaustif en terme de nombre de méthodes et d'attributs. De plus, il correspond à une idée que nous nous faisons du squelette du logiciel en terme de conception logicielle. Il ne constitue donc en aucun cas un réprésentation exacte du code existant. Cette dernière est présentée en section 1.3.2. Il permet d'avoir une vision globale des fonctionnalités déjà présentes,

[3] cf. Glossaire

sur lesquelles s'appuie notre projet.

En figure 2.2 est présenté un diagramme de cas d'utilisation. Il permet d'identifier les possibilités d'interaction entre le système et les acteurs (intervenants extérieurs au système), c'est-à-dire toutes les fonctionnalités que doit fournir le système. Ici, nous voyons que l'utilisateur peut interagir avec le système via trois fonctionnalités importantes. Il peut en effet demander l'analyse de son code source MatLab, tout en ayant éventuellement défini au préalable un ou plusieurs nouveaux opérateurs. A partir d'un code source analysé, il peut aussi générer un graphe optimisé, visualisable par un utilitaire intégré au logiciel GraphLab.

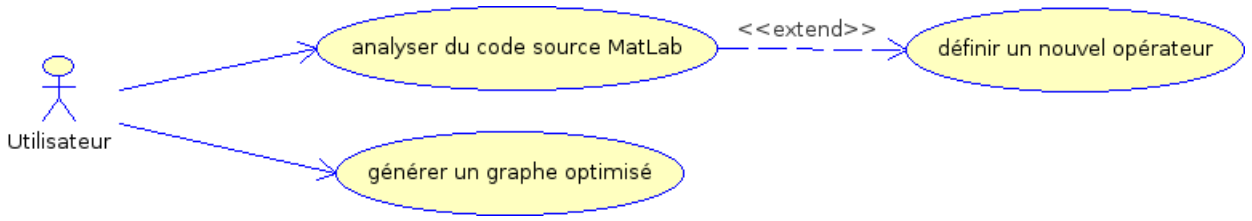


Fig. 2.2 – Diagramme de cas d'utilisation

En figure 2.3 est présenté un diagramme de séquence. Il représente de manière séquentielle le déroulement des traitements et des interactions entre les éléments du système et/ou de ses acteurs. Ici, une utilisation normale du logiciel pour l'utilisateur est, premièrement, d'ouvrir la console du logiciel GraphLab. Ensuite, il demande à ce que son fichier source MatLab soit analysé. L'analyseur ayant effectué son travail, traite le graphe obtenu afin de l'optimiser (développement/factorisation, simplifications arithmétiques ...). Enfin, l'utilisateur ferme la console ouverte.

Les flèches en pointillés constituent des retours de fonctions, c'est-à-dire que la méthode appelée informe la méthode appelante sur l'état du traitement à la sortie de cette fonction.

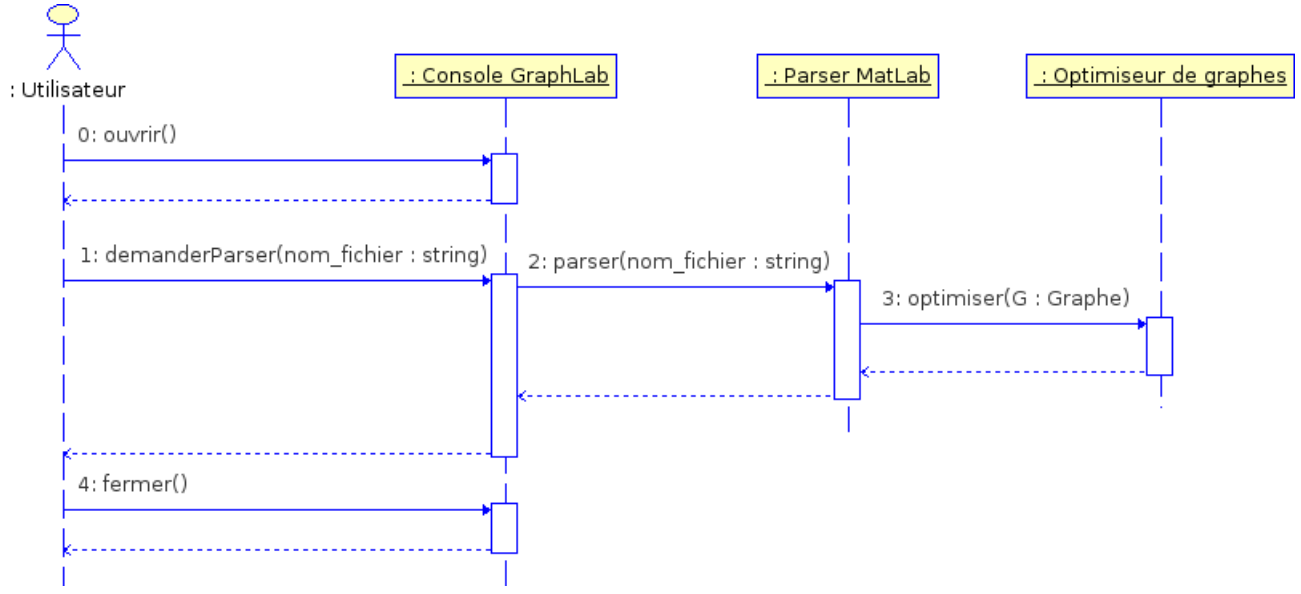


Fig. 2.3 – Diagramme de séquence

2.3 Description des besoins fonctionnels

Lors de nos rendez-vous avec le client, il est apparu toute une liste de fonctionnalités à modifier ou à mettre en place. Il est à noter que cette section présente les besoins fonctionnels énoncés par le client. Il ne s'agit en aucun cas d'un engagement.
Le client a exprimé trois types de besoins fonctionnnels classés par ordre de priorité :

- ceux concernant le Parser MatLab sont prioritaires pour le client,
- ceux touchant à l'optimisation des architectures de processeurs résultantes sont fortement souhaités,
- ceux concernant la partie IHM de GraphLab sont moins importants.

Dans l'état actuel, le Parser MatLab est plutôt basique. De ce fait, il existe un ensemble d'améliorations et de nouvelles fonctionnalités pouvant lui être apporté. Globalement, les fonctionnalités du parser sont prioritaires vis à vis des autres souhaits. Voici la liste des fonctionnalités à mettre en place :

1. Dans un processeur créé par l'outil GraphLab, il est impossible de construire des structures itératives. De ce fait, il est nécessaire de dérouler chacune des boucles rencontrées dans le code source MatLab. Il faut les transformer en suites d'instructions linéaires. Néanmoins, ceci n'est possible que parce que les boucles sont déterministes. Cela signifie que chacun des paramètres des boucles est connu lors de la compilation. Par conséquent, il est possible de connaître le nombre de fois qu'une boucle va être exécutée. Cette gestion est déjà partiellement implantée. Le besoin exprimé par le client est de prendre en compte tous les cas de structures itératives dans la sémantique utilisable.
2. Il s'agit de prendre en compte toute la sémantique qui n'est pas encore prise en charge par le parser. Par exemple, une spécificité du langage MatLab n'est pas encore prise en compte par le parser : la syntaxe d'accès à un élément d'un tableau et la syntaxe d'appel d'une fonction sur un argument sont les mêmes. Si z est un tableau, z(0) correspondra à un accès à la première case de z. Si z est une fonction à un argument, alors z(0) correspondra à l'appel de la fonction z sur l'argument entier 0. Cependant, le langage MatLab n'admet aucune collision entre les noms de fonctions et de tableaux. Il s'agit d'étendre la sémantique en reconnaissant les appels de fonctions et les accès dans un tableau.
3. Le client souhaite que GraphLab gère des fonctions déclarées dans des fichiers annexes. C'est-à-dire que lorsque le parser trouve une fonction qu'il ne connaît pas, il commence par créer un noeud correspondant à la fonction, puis il effectue une recherche dans le dossier où est stocké le code source. S'il trouve un fichier MatLab (portant l'extension **.m**) possédant le même nom que la fonction, alors il crée le graphe associé à cette fonction. Ce graphe secondaire devra être fourni au même moment que le graphe principal. Ainsi, un répertoire constitue un projet car il contient le code source de la fonction principale ainsi que les codes sources des fonctions auxiliaires. Par la suite, une commande pourra permettre d'insérer les graphes secondaires dans le graphe principal afin d'obtenir un graphe complet. Le besoin fonctionnel exprimé par le client est que la déclaration de fonctions auxiliaires soit faite dans des fichiers annexes de même nom. Il demande aussi que l'utilisateur ait la possibilité de remplacer les fonctions auxiliaires par leur représentation sous forme de graphe.
4. Une des particularités intéressantes du langage MatLab est qu'il peut gérer des fonctions à plusieurs sorties. Par exemple,

```
1 function [x,y,z]=f(a)
2 x = a - 1;
3 y = a + 1;
4 z = 2 * a;
```

   Lors de l'instructions [a, b, c] = f(5), les trois sorties de la fonction f sont stockées respectivement dans les variables a, b et c. Pour le moment, cette fonctionnalité n'est pas prise en charge par GraphLab. Le client demande à ce que l'utilisateur puisse employer des fonctions à plusieurs sorties dans son code MatLab.
5. Un des points les plus importants sur lequel a insisté notre client est la détection ainsi que la gestion des délais apparaissant dans les applications de traitement du signal. Par exemple, lors de l'instruction a = a + 1, il faut que, au cycle d'horloge suivant cette instruction, la nouvelle valeur de a soit à jour.

Cela peut paraître simple mais, sur un processeur, ce n'est pas le cas. La valeur de a est stockée dans un registre et les calculs sont effectués dans des registres séparés. Or, le registre contenant la valeur de a doit être mis à jour avant que ne commencent les calculs suivants. Il y a déjà eu une première tentative d'implantation de cette fonctionnalité mais elle n'a pas abouti. La difficulté vient du fait qu'il n'existe pas forcément de syntaxe explicite permettant d'identifier de telles variables.

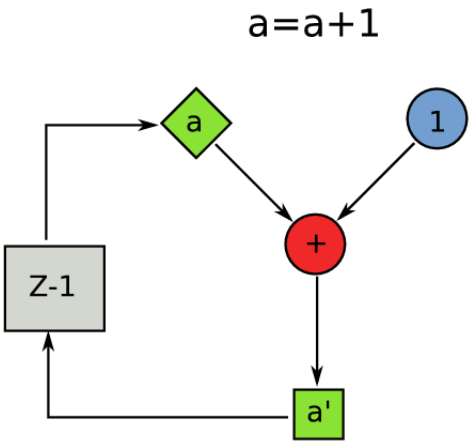


Fig. 2.4 – Exemple de modélisation d'un délai

Comme les variables ne sont pas réutilisables lors du calcul, le résultat de cette opération est stocké dans une variable séparée, ici a'. Il faut réinjecter le contenu de a' dans a avant le calcul suivant. Cette opération est modélisée par un noeud $Z-1$. Le client souhaite voir apparaître ces délais dans les graphes sous forme de noeuds particuliers.

6. La complexité des fonctions utilisées dans le domaine du traitement numérique, par exemple, évolue très rapidement. GraphLab doit pouvoir suivre cette évolution. La seule limite à la complexité des fonctions gérables par l'outil doit être la taille de la mémoire allouable sur la machine. Par exemple, le code MatLab suivant :

```
1    for i=1:10
```

pourrait devenir :

```
1    for i=1:1000000
```

De ce fait, le client a insisté sur le fait qu'il ne fallait en aucun cas borner la complexité des fonctions du code source MatLab pour que GraphLab puisse supporter de telles évolutions.

Le deuxième type de fonctionnalité concerne les optimisations. Il s'agit d'un ensemble d'optimisations ayant pour but d'améliorer les architectures de processeurs résultant de la compilation du code MatLab. Le client a imposé que ces optimisations s'effectuent distinctement de l'étape de parsing, dans un module bien distinct. Elles ne peuvent en aucun cas être effectuées dans le parser, elles doivent être faites lors d'une étape postérieure pour que celles-ci soient indépendantes du parser. Ce module permettra d'effectuer une série de transformations sur les graphes. Pour l'ensemble des optimisations, il serait bien de pouvoir estimer le temps nécessaire pour effectuer les opérations afin que l'utilisateur puisse se rendre compte du temps à attendre. Voici la liste des optimisations proposées par le client :

1. Le client veut avoir à disposition une description facilement modifiable de règles de simplifications s'appliquant sur les opérateurs. Le client doit pouvoir modifier ces règles comme il le souhaite. Par exemple, l'expression :

```
1    y = a + b * 1 - c + 0 * (d + e)
```

devra se simplifier en :

```
1    y = a + b - c
```

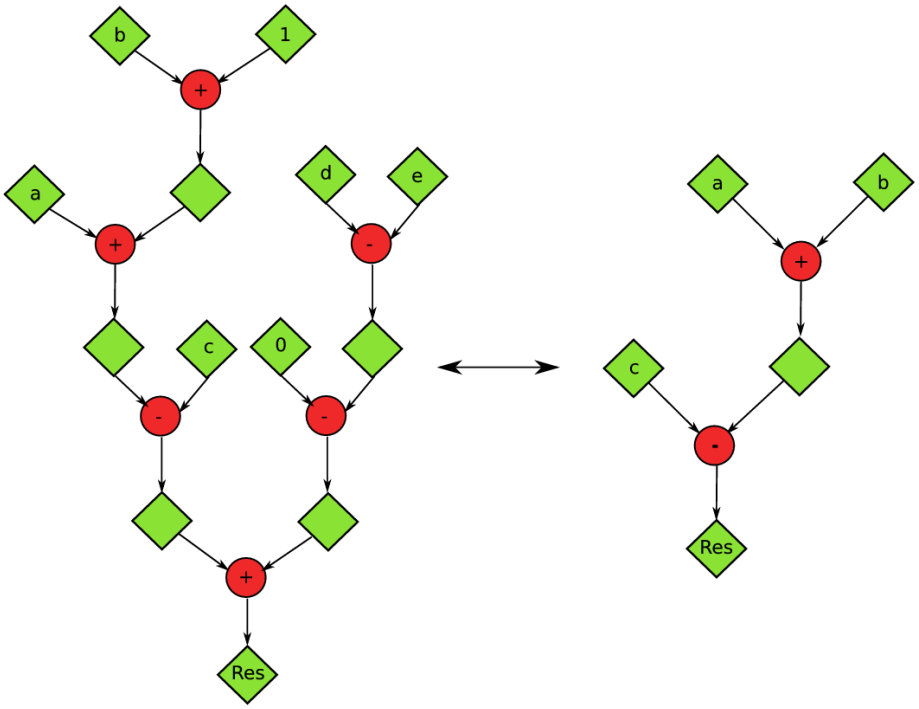


Fig. 2.5 – Exemple de simplification

Il s'agit d'une fonctionnalité dont doit pouvoir bénéficier tout utilisateur de GraphLab. Pour cela, il sera peut-être nécessaire de définir un nouveau langage. Cependant, le nombre de règles mathématiques que nous serons amenés à rajouter pendant la vie de l'outil est très limité. On peut donc se poser la question de l'intérêt d'une interface. Une interface n'est donc pas nécessaire, sous réserve que la syntaxe du langage de spécification soit suffisamment simple. Ainsi, le client souhaite pouvoir préciser un ensemble de règles de simplification élémentaire pour des opérateurs pré-existants ou de nouveaux opérateurs.

2. Il faut pouvoir paralléliser les expressions lorsque celles-ci utilisent des opérateurs associatifs entre eux et que l'ordre dans lequel s'effectuent les opérations n'est pas imposé à l'aide d'un parenthésage. Par exemple,

```
1    z = a + b + c ; z = z + d
```

doit être mis sous la forme

```
1    z = (a + b) + (c + d)
```

qui maximise le nombre d'opérations effectuées en même temps et non sous la forme

```
1    z = ((a + b) + c) + d
```

qui effectue linéairement le calcul. Cette technique permet de gagner quelques cycles d'horloge.

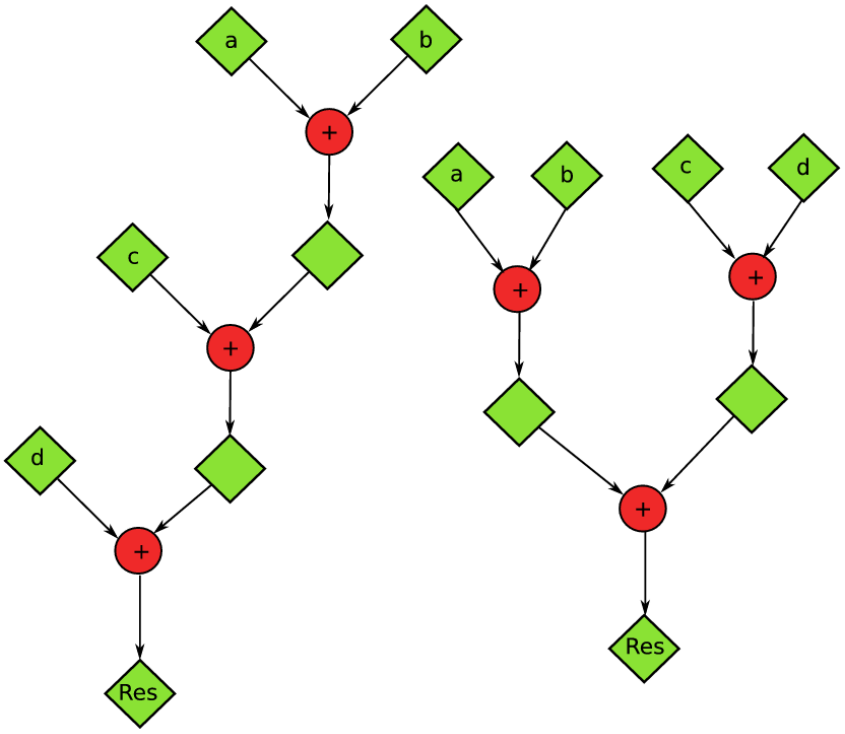


Fig. 2.6 – Calcul parallélisé et calcul linéaire

Le client souhaite pouvoir compresser le graphe des entrées vers la sortie quand un opérateur basique (+, -, * ou /) est utilisé plusieurs fois en utilisant les règles d'associativité de cet opérateur afin de limiter le nombre de cycles d'horloge nécessaire.

3. Le client souhaite aussi pouvoir passer de la forme factorisée d'un graphe à sa forme développée. En effet, le graphe représente des expressions arithmétiques qui peuvent parfois être factorisées ou développées. Ces transformations au niveau arithmétique ont leurs équivalents au niveau des transformations du graphe. Ce sont ces dernières que le client nous demande de mettre en place. En effet, dans le graphe, il se peut qu'un motif se répète de façon inutile. On peut factoriser ce motif de la même manière que pour une expression arithmétique. Par exemple, si on prend le cas de la fonction calculant l'expression arithmétique :

```
1    a * b + a * c
```

, celle-ci peut être factorisée en

```
1    a * (b + c)
```

. Inversement, un graphe doit aussi pouvoir être développé.

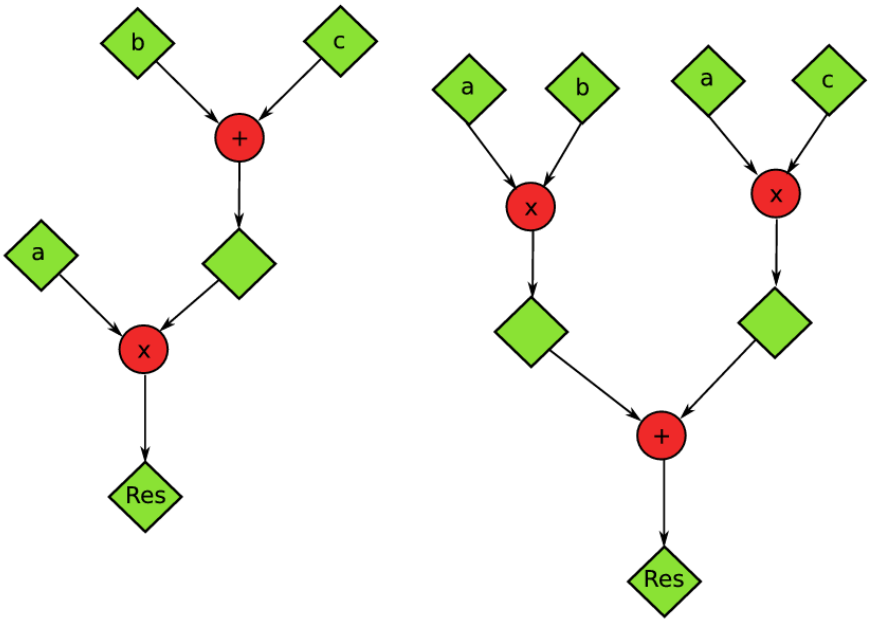


Fig. 2.7 – Formes factorisée et développée

Par exemple, la forme factorisée permet de maximiser le nombre d'opérations pouvant être effectuées en parallèle alors que la forme développée minimise le nombre de variables temporaires. Une des deux formes sera choisie selon le contexte. Le client souhaite ainsi pouvoir implémenter des processeurs possédant des caractéristiques différentes.

Le troisième type de fonctionnalité concerne l'interface de GraphLab. Voici la liste des fonctionnalités graphiques souhaitées par le client :

1. Le besoin fonctionnel exprimé par le client est que l'interface de débogage qui permet d'afficher les graphes autorise une visualisation hiérarchique des graphes. En effet, le fait de pouvoir regrouper un ensemble de calculs sous un noeud commun que l'on peut déplier permettrait d'augmenter considérablement la complexité des graphes affichés par l'outil de débogage. Par conséquent, le débogage des fichiers sources MatLab serait facilité pour le client.
2. Dans la même optique, le client envisageait aussi le développement et l'implantation d'un algorithme de placement des noeuds du graphe. Cet algorithme améliorerait la compréhension de graphes possédant un nombre de noeuds important.
3. Le client proposait aussi de développer en Java un prototype d'IHM complétant l'interface existante.
4. Le shell étant basique, le client souhaitait que diverses améliorations et nouvelles fonctionnalités soient ajoutées. Ces fonctionnalités n'ont pas été abordées en détail lors des réunions,
5. GraphLab possède un outil graphique permettant de visualiser des mesures statistiques effectuées sur les graphes. Cet outil peut lui aussi accueillir de nouvelles fonctionnalités. Par exemple, à l'heure actuelle, il est impossible d'afficher deux graphiques dans le même espace. Il s'agit d'une des améliorations envisagées. Ici aussi, le client souhaitait que de nouvelles fonctionnalités soient ajoutées. Cependant, ce point n'a pas été abordé en détail.

Cette partie présentait les trois axes de travail dont nous a fait part le client lors des différentes réunions. Les fonctionnalités sur lesquelles nous nous engageons sont expliquées en détail dans la partie qui suit, concernant les prestations attendues.

3
Prestations attendues

3.1 Présentation des prestations attendues

Suite à la description des besoins fonctionnels évoqués précédemment, voici la liste des fonctionnalités que nous nous engageons à développer dans les contraintes de moyens et de temps qui nous sont accordées.

3.1.1 Le parser MatLab

Fonctionnalités existantes

Le parser est à l'heure actuelle en mesure d'analyser un code source MatLab utilisant une sémantique limitée et d'en produire le graphe résultant. Il sait en particulier traiter les aspects suivants :

- les affectations de variable,
- l'évaluation d'expressions arithmétiques,
- l'appel à des fonctions dans des cas limités,
- l'utilisation et l'accès aux tableaux dans des cas limités,
- la définition de fonctions prenant un nombre multiple d'arguments mais ne renvoyant qu'une unique valeur,
- la détection et le dépliage des structures de contrôle de type **boucles** dans des cas limités et dont le nombre d'itérations est connu à la compilation.

Fonctionnalités à améliorer au niveau de l'analyse syntaxique

- Il faut améliorer le support des appels de fonctions et des accès aux tableaux. En particulier, il faudra pouvoir distinguer sans ambiguïté un appel de fonction d'un accès à une case d'un tableau, ces deux opérations ayant une syntaxe identique. Cependant, une variable de type 'tableau' doit obligatoirement être déclarée dans le corps de la fonction.
- Il faut améliorer la gestion des boucles. En particulier, le parser doit pouvoir gérer des boucles disjointes ou encore des boucles ayant un pas d'incrémentation ou de décrémentation différent de 1.
- Les fonctions devront pouvoir renvoyer plusieurs valeurs.

Fonctionnalités à améliorer au niveau de l'analyse sémantique

- Le parser devra pouvoir gérer les variables globales même si celles-ci n'ont pas été explicitement déclarées dans le fichier MatLab à l'aide du mot clé **export**.
- Le parser devra pouvoir détecter une dépendance de données de type "lecture avant écriture" sur une variable globale, conduisant à l'apparition d'un noeud spécifique noté **Z-1** dans le graphe.
- Le parser doit pouvoir détecter des appels de fonctions définies par l'utilisateur dans des fichiers MatLab annexes mais situés dans le même répertoire. Les fichiers porteront le nom de la fonction suivi de l'extension **.m** dont ils contiennent la définition. Chaque fichier ne devra alors contenir la définition

que d'une unique fonction. Ceci implique que chaque fichier aura un nom unique et qu'il ne sera pas possible de proposer plusieurs définitions d'une même fonction. Ceci exclue également toute forme de surcharge ou de polymorphisme dans les fonctions. Lorsque les prototypes de la fonction appelée et de la fonction définie sont différents, l'erreur sera notifiée au travers d'un retour de fonction. En aucun cas il n'y aura de warning ou d'interaction avec l'utilisateur vis à vis de cette erreur. Lors de la génération du graphe, le parser devra pouvoir automatiquement insérer dans celui-ci les sous-graphes correspondants à ces fonctions.

3.1.2 Transformation des graphes

Sous la notion de transformation de graphes se cachent en réalité des techniques à implanter qui ont pour but essentiel d'optimiser à divers niveaux le graphe obtenu. Il n'y a pour le moment aucune fonctionnalité existante dans ce domaine et, de ce fait, nous partons de rien.

Fonctionnalités à ajouter

- Il faudra pouvoir détecter des motifs locaux traduisant l'évaluation d'expressions arithmétiques sous forme factorisée ou développée et pouvoir modifier localement le graphe afin de passer d'une forme à l'autre et réciproquement. La détection de ces motifs se limitera aux expressions ne faisant intervenir que les quatre opérateurs arithmétiques usuels que sont l'addition, la soustraction, la multiplication et la division.
  Remarque : Bien qu'il eut été techniquement possible de réaliser ces optimisations au niveau du parser MatLab, nous sommes cependant obligés d'implanter celles-ci dans le module d'optimisation des graphes. En effet, un cas d'utilisation possible du logiciel est la possiblité de visualiser un graphe qui n'aurait pas été créé par le parser MatLab mais importé depuis une source externe. Dès lors, l'utilisateur pourrait tout à fait souhaiter effectuer une optimisation de ce graphe d'où la nécessité que ces optmisations soient présentes à ce niveau, le graphe étant un modèle formel.
- Il faudra pouvoir détecter des motifs locaux traduisant l'évaluation d'expressions arithmétiques dont tout ou une partie du résultat peut être connu à la compilation. De tels motifs devront alors pouvoir être simplifiés en conséquence. Ces simplifications étant des caractéristiques propres aux opérateurs, il faudra mettre en place un système permettant à l'utilisateur final de décrire ces règles de simplification pour chaque opérateur existant.
  Remarque : Pour les mêmes raisons que celles évoquées précédemment, nous sommes obligés d'implanter ces optimisations à ce niveau et non dans le parser MatLab.

3.2 Fonctionnalités non retenues

Voici la liste des fonctionnalités que nous avons décidé de ne pas développer, leur priorité étant jugée trop peu élevée au vu des contraintes auxquels nous sommes soumis :

3.2.1 L'interface Homme-Machine

- Nous ne développerons pas une gestion des espaces dans l'interpréteur de commandes. A l'heure actuelle, ces espaces doivent être précédés d'un caractère particulier afin d'indiquer à l'interpréteur que cet espace ne joue pas le rôle de séparateur d'arguments.
- Nous ne développerons pas d'interface graphique pouvant se substituer à l'interpréteur de commandes.

3.2.2 La visualisation des graphes

- A l'heure actuelle, le visualisateur est capable d'afficher correctement, c'est-à-dire en particulier sans arête s'entrecroisant, des graphes d'une taille modeste. Nous n'améliorerons pas les mécanismes d'af-

fichage afin que ces derniers puissent prendre en compte des graphes d'une complexité nettement plus importante.

3.2.3 L'affichage de statistiques sur les graphes

Actuellement, l'outil dispose d'un module, contrôlé au travers d'une interface graphique, permettant d'afficher des statistiques diverses et variées sur les graphes générés par le parser MatLab. Nous ne développerons pas davantage l'interface graphique afin d'offrir des fonctionnalités supplémentaires telle que la superposition de courbes sur un même graphique.

3.3 Répartition des tâches

La répartition du travail s'effectue de manière équilibrée. A chaque réunion, à tour de rôle en tant que secrétaire, un membre du groupe est chargé de rédiger un compte-rendu. Celui-ci sera ensuite vérifié, analysé et commenté par le groupe puis transmis à l'encadrant avec lequel aura eu lieu la réunion, à savoir l'encadrant scientifique (le client) ou l'encadrant technique.

Pour effectuer le travail de développement, le groupe sera scindé en deux binômes et un trinôme qui seront chargés de la rédaction d'un compte-rendu lors de chaque réunion avec le client et l'encadrant technique, réunion où les autres membres n'assisteront pas obligatoirement. Une réunion hebdomadaire est fixée pour le groupe complet, afin d'informer chacun de l'avancée du travail effectué.

Les fonctionnalités à développer seront réparties sur les différents groupes de travail (cf. figure C). Les points suivants seront traités lors du développement :

- au niveau du parser MatLab :
  1. Modifications :
     - amélioration du support des appels de fonction et des accès à un tableau,
     - détection des appels de fonctions définies par l'utilisateur,
     - traitement des fonctions à plusieurs sorties
     - support de structures de contrôle de type *boucle* et dépliage de ces structures lors de la création du graphe résultant
  2. Ajouts :
     - traitement de la notion de délai apparaissant dans une expression arithmétique et devant être signalée dans le graphe sous la forme d'un noeud *Z-1*
     - reconnaissance des fonctions ajoutées par l'utilisateur et création d'un graphe secondaire traduisant l'évaluation d'un appel de fonction grâce au code de cette même fonction présent dans un fichier (ne faisant pas partie de la bibliothèque MatLab)
- au niveau du graphe obtenu après analyse du code MatLab :
  1. Ajouts
     - détection de motifs locaux traduisant l'évaluation d'expressions arithmétiques sous forme factorisée ou développée, afin de passer d'une forme factorisée du graphe à une forme développée, et inversement
     - détection de motifs locaux traduisant l'évaluation d'expressions arithmétiques dont tout ou une partie du résultat pourra être connue à la compilation afin de simplifier le graphe

De plus, les fonctionnalités étant liées entre elles, les différents binômes et trinôme seront amenés à travailler sur plusieurs parties d'entre elles. Les réunions hebdomadaires permettant d'informer les membres de l'équipe de développement des avancées de chacun et des difficultés rencontrées le cas échéant, créeront une synergie au sein du groupe. Ainsi, ces synergies permettront à un binôme de pouvoir apporter son aide et sa compréhension à un autre binôme si nécessaire.

D'une manière générale, l'organisation du travail tendra à faire travailler chacune des personnes sur les différentes fonctionnalités, de façon à ce qu'elle puisse être confrontée à toute difficulté et qu'elle mette en place des solutions afin de la résoudre.

Glossaire

ALAP

As Late As Possible.

ASAP

As Late As Possible, le plus tard possible

ASAP

As Soon As Possible, le plus tôt possible

ASAP

As Soon As Possible.

FPGA

FPGA (field-programmable gate array, réseau de portes programmables in-situ) est un circuit logique programmable, ou réseau logique programmable, est un circuit intégré logique qui peut être reprogrammé après sa fabrication.

Fichier MatLab

Les fichiers MatLab manipulés sont des suites d'opérations mathématiques, arithmétiques et logiques.

I.D.E.

Un environnement de développement intégré (E.D.I. ou I.D.E. en anglais pour *Integrated Development Environment*) est un programme regroupant un éditeur de texte, un compilateur, des outils automatiques de fabrication, et souvent un débogueur.

I.H.M.

Interface Homme/Machine.

IMS

Le laboratoire de l'Intégration du Matériau au Système développe des actions de recherche originales et cohérentes dans les domaines de :
- la modélisation et l'élaboration des matériaux, des capteurs et des microsystèmes pour les dispositifs électroniques,
- la modélisation, la conception, l'intégration et l'analyse de fiabilité des composants, circuits et assemblages,
- l'identification, la commande, le diagnostic, le traitement du signal et des images, la supervision et la conduite des processus complexes et hétérogènes.
Les domaines d'application concernent principalement les télécommunications, les transports, la santé, l'énergie. Ils sont traités au travers de nombreux projets européens, nationaux ou régionaux.

J.D.K

Le Java Development Kit (couramment abrégé en J.D.K.) est l'environnement dans lequel le code Java est compilé pour être transformé en byte-code afin que la J.V.M. (machine virtuelle de Java) puisse l'interprêter.

Java Hints

Les Java Hints proposent un ensemble de fonctionnalités qui aident au développement Java, comme par exemple l'affichage de la liste des méthodes applicables à un objet.

MOA

MOA est un sigle, qui signifie maîtrise d'ouvrage en gestion de projets informatique

MOE

MOE est un sigle, qui signifie maîtrise d'oeuvre en gestion de projets informatique

MatLab

MatLab est à la fois un langage de programmation et un environnement de développement développé et commercialisé par la société américaine The MathWorks. MatLab est utilisé dans les domaines de l'éducation, de la recherche et de l'industrie pour le calcul numérique mais aussi dans les phases de développement de projets.

Paquetage

Un paquetage est un conteneur logique permettant de regrouper et d'organiser les éléments dans le modèle U.M.L..

U.M.L.

U.M.L. (en anglais Unified Modeling Language, « langage de modélisation unifié ») est un langage graphique de modélisation des données et des traitements. C'est une formalisation très aboutie et non-propriétaire de la modélisation objet utilisée en génie logiciel.

VHDL

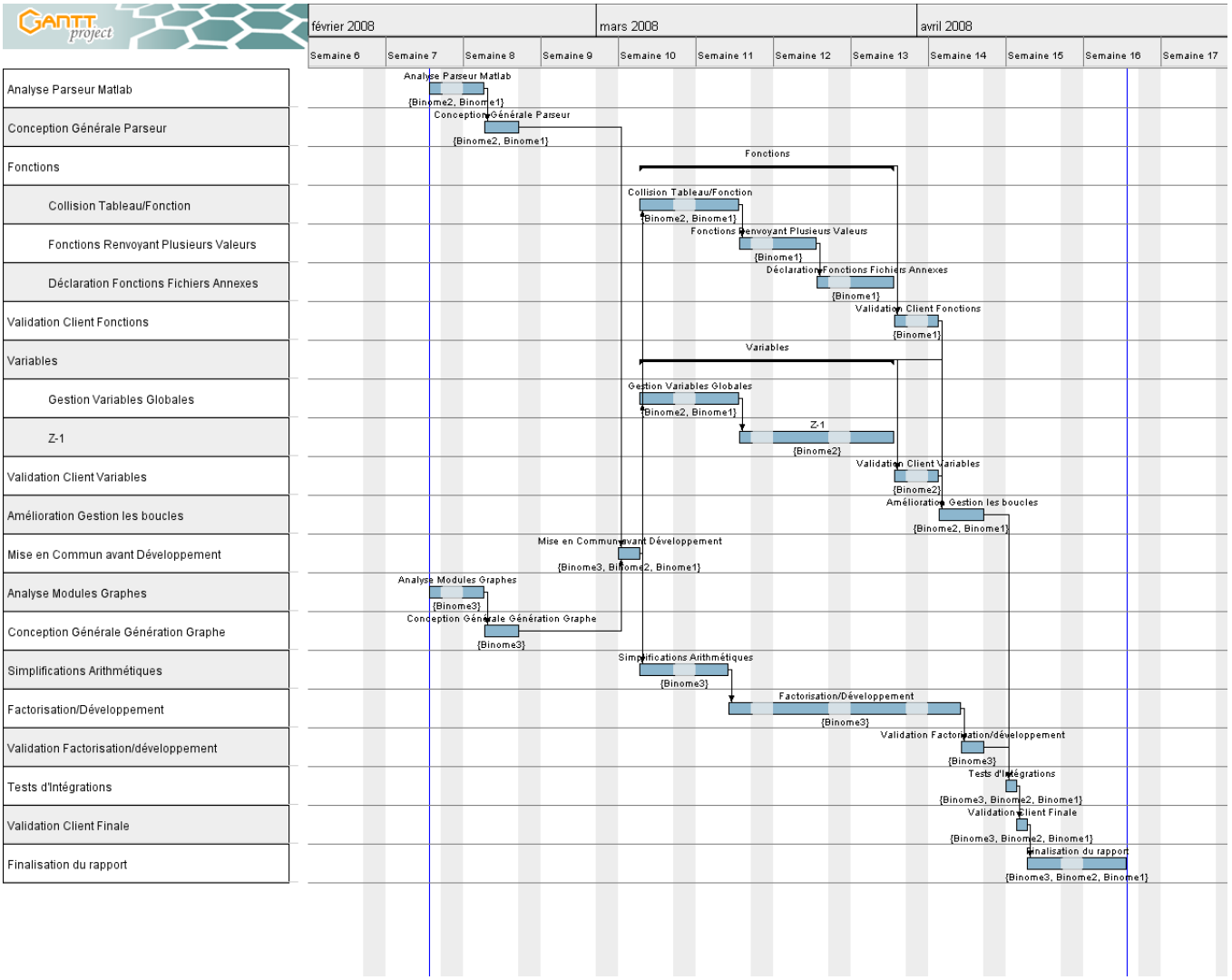
Le VHDL (abréviation de l'expression anglaise Very high speed integrated circuit Hardware Description Language) est un langage de description matériel destiné à décrire le comportement et/ou l'architecture d'un système électronique numérique.

refactorisation

La refactorisation (anglicisme venant de *refactoring*) est une opération de maintenance du code informatique.

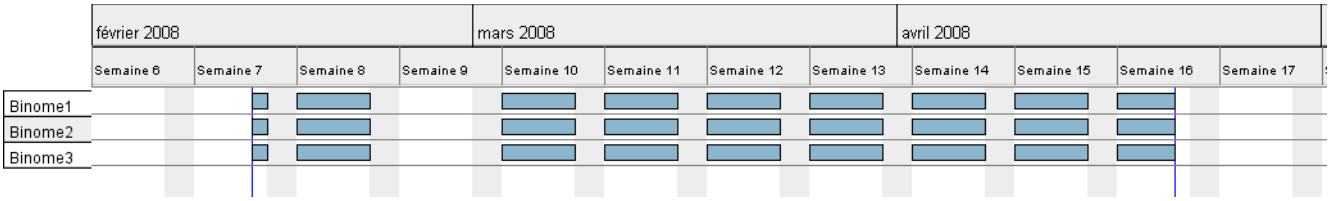
A

Diagramme de GANTT du projet



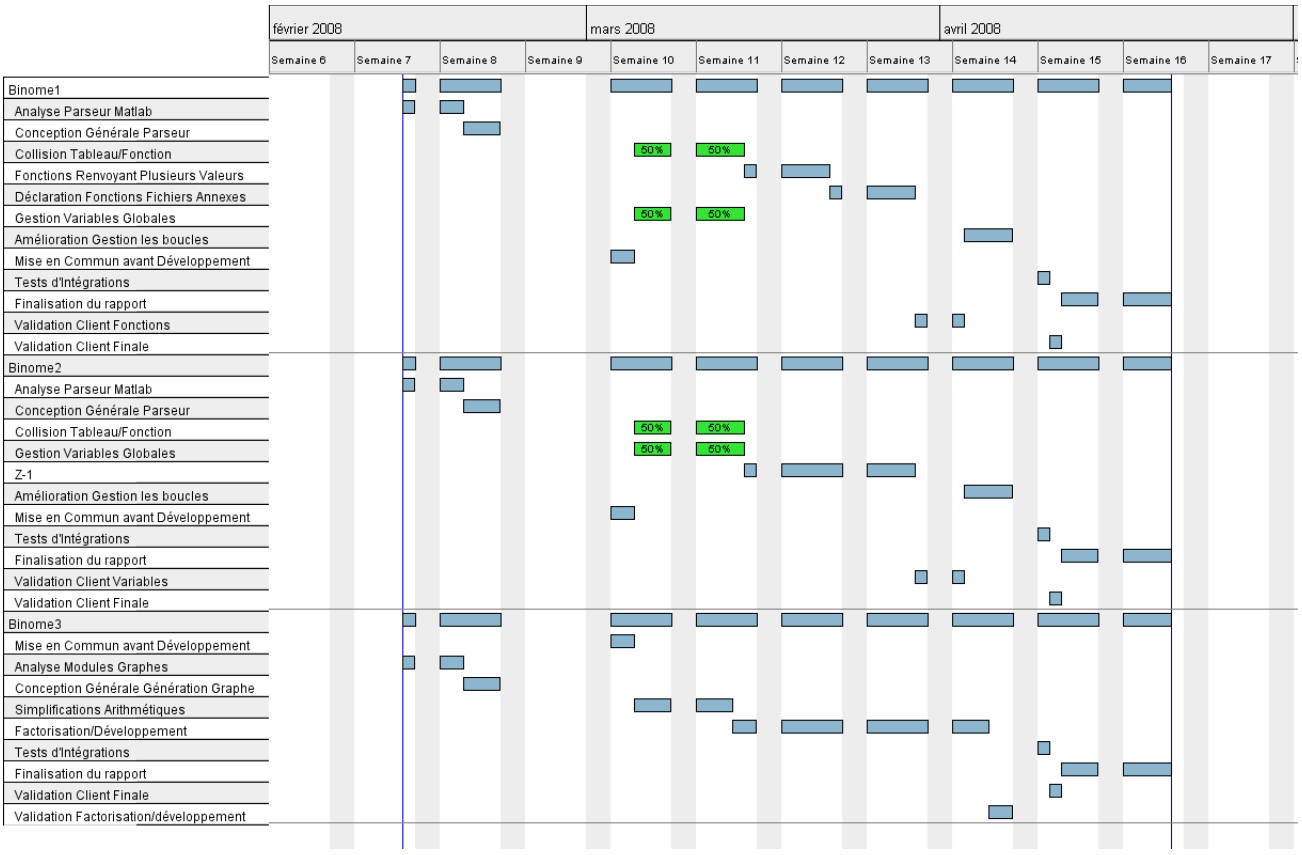
B

Diagramme des Ressources (réduit)



C

Diagramme des Ressources (complet)



Signatures

Fait à Talence, le 21 février 2008.

Je déclare avoir pris connaissance du présent cahier des charges et en agréer tous les termes.

En particulier j'accepte que l'équipe suivante, sous la responsabilité de P. DUCHON, réalise le projet décrit dans le présent document :

- BENITTO Adnane,
- BUISSON Rémi,
- EL AFRIT Mohamed Amine,
- GATTI Stéphanie,
- PORTRAT Benoît,
- RATOUIT Thomas,
- STORDEUR Ludovic.

le client (B. LE GAL),

l'encadrant pédagogique (P. DUCHON),

Signatures des deux partis précédées de la mention "Lu et approuvé".