IUT Nice Côte d'Azur
Département Informatique / LP SIL
41 bd Napoléon III
06206 Nice cedex 3

Mohamed Hadj Djilani
Maître de stage : Fabrice Peix
Tuteur IUT : Léo Donati

RAPPORT DE STAGE

Licence Professionnelle Systèmes Informatiques et Logiciels spécialité Imagerie Numérique et Informatique

Stage effectué à l'INRIA Sophia Antipolis, équipe-projet MASCOTTE

mai - août 2008

REMERCIEMENTS

Je tiens à remercier tout le personnel de MASCOTTE, pour son accueil, sa sympathie, sa disponibilité et ses conseils.
Et tout particulièrement je remercie M. Michel Syska, enseignant en LP SIL et membre de MASCOTTE, pour m'avoir permis d'effectuer ce stage, ainsi que M. Fabrice Peix, mon tuteur, ingénieur à MASCOTTE, pour son encadrement, ses explications et la confiance qu'il m'a accordée tout au long du stage.

RÉSUMÉ

J'ai effectué mon stage de fin de formation LP SIL au sein de l'équipe-projet de recherche MASCOTTE à l'INRIA Sophia Antipolis. L'équipe MASCOTTE travaille sur la conception des réseaux de télécommunications. Ma position au sein de cette structure était celle d'un technicien. Tout au long du stage, j'ai travaillé sur MascOpt, un des logiciels développés par MASCOTTE. MascOpt est basiquement un ensemble d'outils développés en Java et qui traitent des problèmes d'optimisation réseau.
En bref, mon stage consistait dans un premier temps à intégrer un composant logiciel supplémentaire à MascOpt. Ce composant est une interface pour l'utilisation via MascOpt d'une librairie de solver de programmes linéaires (CLP/CBC). Pour cela j'ai travaillé avec différents outils dont la librairie CLP/CBC pour la programmation linéaire et JNI pour l'implémentation native de classes Java.
Dans un deuxième temps j'ai travaillé avec l'environnement Maven. C'est un outil de gestion, de distribution, de documentation et d'installation de projets de manière efficace, claire, réutilisable et extensible. Il s'agissait de convertir en projet Maven le composant développé durant la première partie du stage et d'envisager comment procéder pour étendre cette conversion au logiciel MascOpt.

ABSTRACT

I have done my end LP SIL internship within the MASCOTTE team-project, at INRIA Sophia Antipolis. The MASCOTTE team works on telecommunication network design. My post in this organization was technician.
All along the internship period, I worked on MascOpt, which is one of the softwares developed by MASCOTTE. Basically, MascOpt is a set of Java written tools concerning network optimization problems.
In brief my training period first deal with an integration of a new software component in MascOpt. This component is an interface for using a linear program solver (CLP/CBC) via MascOpt. To do that I worked with several tools, both of which were CLP/CBC library for linear programming and JNI for native implementation of Java classes.
Secondly, I worked on Maven. This is a tool for management, releasing, documentation and installation of projects in a efficient, clear, reusable and extensible way. The matter was to convert the software component I had developped before in a Maven format project and to consider to do the same thing with MascOpt software.

SOMMAIRE

INTRODUCTION

Mon sujet de stage étant d'intégrer un solver CLP/CBC de programmes linéaires au logiciel MascOpt, ainsi que de « maveniser » le projet résultant, mes objectifs au début du stage étaient les suivants :

- Me familiariser avec les nouveaux outils et domaines de travail : les librairies CLP, CBC, MascOpt et l'API JNI nécessaires à l'aboutissement du projet qui m'a été confié. Mais aussi Maven dont je voulais profiter pour acquérir de nouvelles connaissances en matière de génie logiciel.
- Revoir et confirmer les connaissances et compétences acquises, notamment en LP SIL ; avec l'utilisation de l'environnement de développement Eclipse, des langages de programmation Java, C et C++. Revoir et si possible approfondir mes connaissances concernant la programmation linéaire que nous avions étudiée en premier semestre de LP SIL.

Cela étant, je voulais également en apprendre un peu plus sur MASCOTTE, me faire une idée des activités qui y sont menées. Et enfin profiter d'une expérience dans un milieu de recherche, qui de prime abord me paraissait intéressant, ce qui s'est d'ailleurs avéré comme tel.

C'est pourquoi dans ce rapport je commencerai par présenter MASCOTTE, son organisation, ses activités.

J'aborderai ensuite les éléments nécessaires à la compréhension de la première partie de mon stage, à savoir l'intégration du solver CLP/CBC à MascOpt, ainsi que la mise en oeuvre concrète de ce travail, étape par étape.

Enfin, dans la dernière partie je présenterai Maven, et le travail que j'ai eu à faire en seconde partie de stage avec cet outil ; principalement convertir au format Maven le projet que j'avais développé en première partie.

I PRÉSENTATION DE LA STRUCTURE D'ACCUEIL : MASCOTTE

1. MASCOTTE en plusieurs points

- MASCOTTE est l'acronyme de Méthodes Algorithmiques, Simulation, Combinatoire et OpTimisation des TÉlécommunications.

- MASCOTTE est un projet de recherche qui a pour objet la conception des réseaux de télécommunications. Le but de ce projet est notamment d'établir des méthodes et des outils algorithmiques, à la fois théoriques et appliqués, à travers les axes de recherche suivants :
  - Algorithmique, mathématiques discrètes et optimisation combinatoire.
  - Algorithmique des communications.
  - Dimensionnement de réseaux (optiques WDM[1], MPLS[2], embarqués, radio WiFi WiMax et satellites).
  - Simulation de systèmes complexes.
  - Protection et partage de ressources.
  - Réseaux logiques (*overlay computing*).

- MASCOTTE est un partenariat INRIA-I3S :
  - L'Institut National de Recherche en Informatique et en Automatique (INRIA) de Sophia Antipolis – Méditérranée, est un acteur majeur du réseau de recherche et du campus STIC du bassin méditérranéen. En effet, ses sites sont localisés dans les villes de Marseille, Montpellier et dans la communauté d'agglomération de Sophia Antipolis, qui sont parmi les plus grandes technopoles européennes.

    L'INRIA de Sophia Antipolis – Méditérranée compte à peu près 400 scientifiques, une trentaine d'équipes de recherche, de nombreux partenariats régionaux, industriels et internationaux. Il est notamment à l'origine de la création de 15 start-up issues de travaux de recherche.

1 WDM : Wave Division Multiplexing (Multipexage par par longueur d'ondes).

2 MPLS : Multi-Protocol Label Switching.

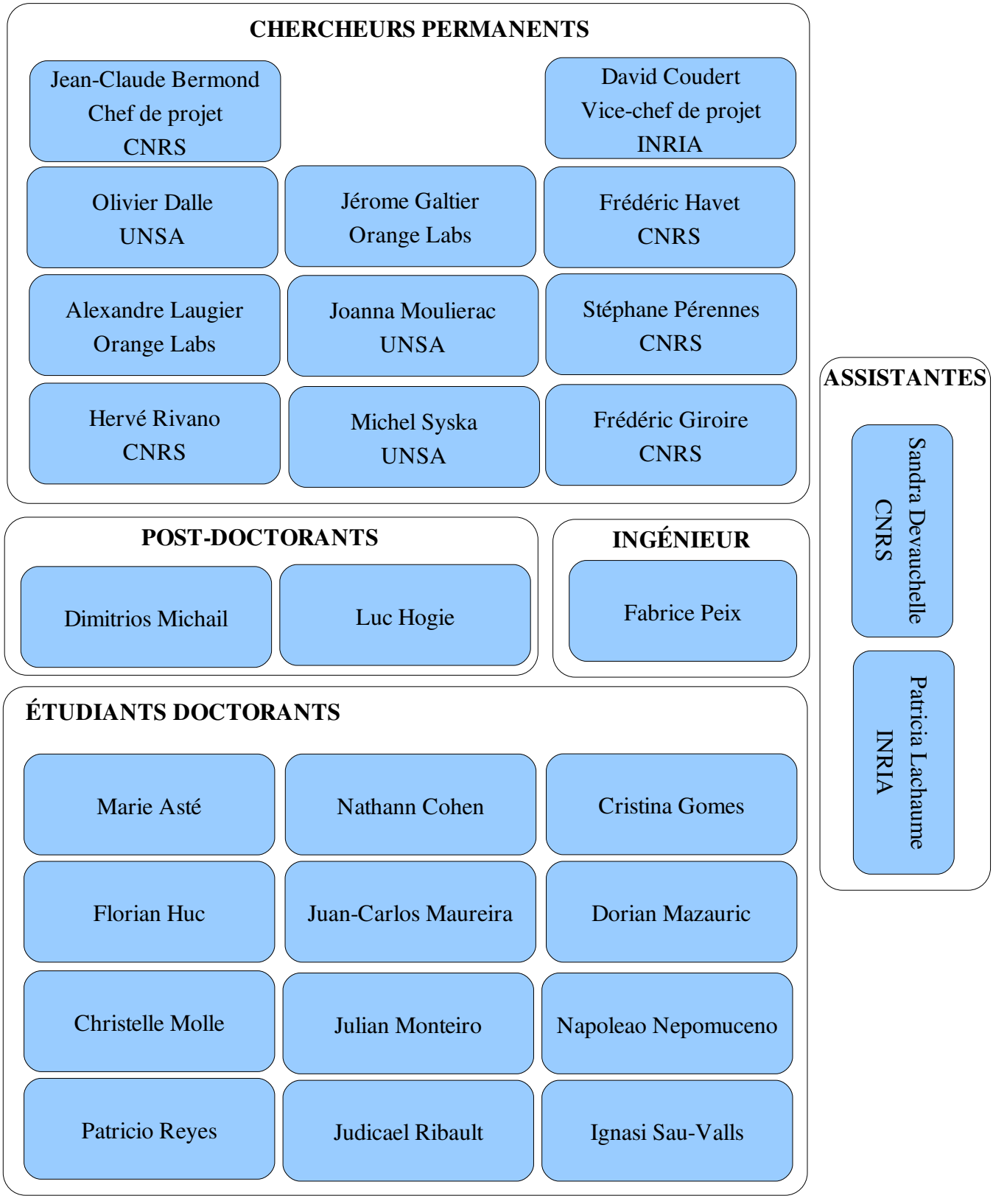
- Le laboratoire Informatique Signaux et Systèmes Sophia-Antipolis (I3S), est une Unité Mixte de Recherche (UMR) entre l'Université de Nice-Sophia Antipolis (UNSA) et le Centre National de la Recherche Scientifique (CNRS).

- MASCOTTE prend pleinement place dans les thématiques du pôle mondial de compétitivité « Solutions communicantes sécurisées », dont l'INRIA est membre. Avec environ 330 acteurs dans l'industrie, les services, et la recherche, ce pôle situé en PACA, vise à créer une synergie des différents domaines de compétences en microélectronique, logiciel et télécommunications. Avec comme enjeux pour la région PACA, de devenir leader du marché mondial des solutions communicantes, qui se développe très rapidement et d'apparaître comme une région à fort potentiel technologique pour attirer les laboratoires du monde entier.

- MASCOTTE c'est aussi une collaboration avec l'école informatique de Simon Fraser University de Vancouver, avec de nombreux échanges depuis les années 90. Voici comment l'équipe MASCOTTE décrit cette collaboration sur son site web :

  La collaboration passée a eu pour objectif principal d'appliquer une expertise commune en mathématiques discrètes, et en particulier en théorie des graphes, aux problèmes de conception de réseaux (principes reliant le degré d'un réseau, son diamètre et son nombre de sommets, propriétés structurelles et aux questions liées à la diffusion de l'information dans les réseaux. Sur le plan théorique, elle a contribué a comprendre les phénomènes de diffusion et d'échange total. Sur un plan plus pratique, elle a mis en perspective l'importance des hypothèses de modélisation (commutation de paquets, routage wormhole, réseaux par bus avec comme domaine d'applications le parallélisme). Les deux projets ont à peu près en même temps réorienté leurs thématiques vers la modélisation et la résolution des problèmes issus des réseaux de télécommunications et investi plus dans les relations industrielles. Au sein de l'école d'informatique de SFU a été crée en septembre 2001 un nouveau groupe (projet) qui travaille de fait sur les mêmes sujets que MASCOTTE. Si durant ces dernières années MASCOTTE a eu tendance à collaborer plus avec des partenaires industriels et des partenaires européens, l'équipe de SFU reste par la qualité de ses chercheurs et les thématiques développées comme la plus voisine de nous et un excellent partenaire pour une équipe associée. Plusieurs chercheurs de MASCOTTE (en particulier ceux recrutés récemment) souhaitent aller régulièrement à Vancouver et de manière réciproque plusieurs chercheurs canadiens souhaitent profiter d'années sabbatiques pour venir ici.

2. L'organigramme de MASCOTTE



3. Les contrats et actions de recherche

Pour plus de précisions sur les activités de MASCOTTE, les contrats en vigueur sont présentés ci-dessous. Il s'agit d'indiquer en bref, la thématique de recherche du contrat et les éventuels participants autres que MASCOTTE, etc.

- Action Collaborative de Recherche (ARC) Capacité de Réseaux radio MAillés (CARMA) :
  Cette ARC a été initiée début 2007, avec un budget de 100 000 euros pour deux ans. Elle associe MASCOTTE, aux équipes de recherche ARES (INSA[3] Lyon et INRIA Rhône Alpes), Drakkar (Laboratoire Informatique de Grenoble) et POPS (INRIA Lille). Au sein de MASCOTTE, ce sont David Coudert et Hervé Rivano qui participent à cette ARC.
  Cette ARC s'intéresse à la capacité des réseaux maillés. Sur base théorique, elle vise dans un premier temps à développer des outils pour la modélisation et l'évaluation de la capacité d'un réseau maillé. Dans un second temps, elle veut optimiser cette capacité par des protocoles *cross-layer* (travaillant sur couches réseau et physique). Enfin, valider ces protocoles par simulation et expérimentation.

- IST/FET[4] Algorithmic Principles for Building Efficient Overlay Computers (AEOLUS) :
  AEOLUS est un projet européen débuté en 2005. Des universités de nombreux pays européens y participent ; Allemagne, Chypre, Belgique, Espagne, Grèce, Italie, Suisse, République Tchèque, ainsi que l'institut d'informatique Max-Planck en Allemagne et la société estonienne de recherche Cybernetica. Les membres MASCOTTE actifs sur ce projet sont Olivier Dalle et Hervé Rivano.
  Deux éléments pivots sont étudiés dans ce projet. Le *global computer* qui est en fait un groupe de serveurs, fournissant des services sous forme de puissance de calcul, d'espace disque, et de ressources informationnelles. Et l'*overlay computer* qui est une sorte de machine virtuelle permettant l'accès aux ressources du *global computer*.
  Ce projet comporte une partie théorique qui consiste à étudier les problèmes et algorithmes pour *overlay computers* exécutés sur *global computers*, à concevoir des outils de programmation pour *overlay computers*, avec des algorithmes fiables, ainsi que des méthodes permettant les communications de types sans fil et mobile pour les *overlay computers*.
  La partie pratique vise à développer un prototype d'*overlay computer* implémentant les fonctionnalités définies à l'issue de la partie théorique.

3 INSA : Institut National des Sciences Appliquées.

4 IST/FET : Information Society Technologies (thématique prioritaire de recherche au niveau européen), Future and Emerging Technologies (englobe des projets de recherche européens sur les technologies d'avenir).

- COST[5] 293 GRAphs and ALgorithms in communication networks (GRAAL) :
  GRAAL est un projet européen débuté à la fin de l'année 2004. Il implique des universités, des instituts et des organisations de recherche dans de nombreux pays européens : Angleterre, Belgique, Danemark, Espagne, France, Grèce, Hongrie, Italie, Norvège, Suède, Slovaquie, Slovénie, et non européens : États-Unis (Iowa) et Israël. Des sociétés basées en Europe y participent aussi : Alcatel, Ericsson Research, France Telecom (dont Jérome Galtier, membre MASCOTTE est le représentant), NEC Network Laboratories, Nokia Siemens Networks. David Coudert est le représentant MASCOTTE pour ce projet.
  GRAAL est un projet multidisciplinaire qui touche aux réseaux et qui vise à établir une collaboration rapprochée entre la recherche appliquée et la recherche fondamentale. C'est-à-dire des connaissances et compétences en matière de réseaux de communication (réseaux sans fils ad-hoc, multicouches, et dorsaux à fibres optiques...) et en mathématiques (mathématiques discrètes, algorithmique, optimisation, calcul distribué). Ces dernières permettant de résoudre les problèmes posés par les premières. Le but de cette collaboration est de développer les générations futures de réseaux de communication (réseaux multimédia et réseaux de données, permettant l'accès immédiat à toute sorte d'information dans un environnement mobile). Cette collaboration prend effet dans des ateliers de discussion (*workshops*), dans des missions à court terme et dans la diffusion et le partage de résultats de recherche.

- ANR[6] « Jeunes Chercheurs » Optimisation et Simulation pour l'Étude des Réseaux Ambiants (OSERA) :
  OSERA est un projet de 36 mois. Au sein de MASCOTTE, il rassemble David Coudert, Olivier Dalle et Hervé Rivano. Les équipes OASIS, RAINBOW et RECIF du laboratoire I3S prennent part ou sont consultées pour ce projet. OSERA prend parfaitement place dans le contexte des projets AEOLUS et GRAAL présentés ci-dessus.
  En effet, OSERA traite des problématiques d'optimisation de systèmes de télécommunications mobiles ambiants, actuellement en fort développement (ils constituent l'avenir de l'Internet). OSERA étudie les problèmes de dynamique de ce type de réseaux qui a la caractéristique d'être instable. Pour cela des outils théoriques sont développés : algorithmes d'optimisation (dynamique et distribuée) et de combinatoire, de routage dynamique fiable, pour le dimensionnement et l'exploitation de ce type de réseaux. Pour l'aspect pratique, des outils d'analyses, de validation, d'intégration, et d'exploitation des solutions théoriques sont réalisés. Avec utilisation de techniques de programmation linéaire (avec la librairie MascOpt) et de simulation à évènements discrets.

5 COST est l'acronyme de european COoperation in the field of Scientific and Technical research.

6 ANR : Agence Nationale de Recherche est Groupement d'Intérêt Public mis en place en 2005, dont INRIA et CNRS sont membres, qui vise à favoriser au niveau national, la recherche appliquée et fondamentale, l'innovation et les partenariats entre secteurs public et privé.

Le but du projet OSERA est finalement, d'être à même de proposer une solution optimale (au niveau du coût, donc du dimensionnement), constituée de protocoles, d'algorithmes et d'outils logiciels fiables et de qualité. C'est également la réalisation de simulations reproductibles de réseaux dynamiques à grande échelle.

- ANR Safe P2P-based Reliable Architecture for Data Storage (SPREADS) :
  SPREADS est un projet d'une durée de 36 mois, débuté en janvier 2008 et devant prendre fin en décembre 2010. C'est un partenariat entre MASCOTTE, UbiStorage SA représentée notamment par un ancien membre de MASCOTTE : Sébastien Choplin, LACL[7] de l'université Paris XII, l'équipe NS[8] d'EURECOM (associée avec le CNRS) située à Sophia Antipolis, et l'équipe mixte REGAL (LIP6/INRIA Rocquencourt). Ce partenariat ANR est sponsorisé par le pôle de compétitivité « Solutions Communicantes Sécurisées ».
  Les membres MASCOTTE actifs sur ce projet sont : Olivier Dalle, Michel Syska, Stéphane Pérennes, Philippe Mussi, Luc Hogie, Julian Monteiro, Juan-Carlos Maureira, Judicael Ribault et Fabrice Peix.
  L'objectif de ce projet est d'étudier et de concevoir un système de sauvegarde basé sur des réseaux à grande échelle de type P2P dynamique. Avec des exigences de fiabilité et de confidentialité. Pour cela les problèmes étudiés sont : spécification et vérification formelle de protocoles distribués et de protocoles de communications, optimisation de placement de données, sécurisation de protocoles, gestion de données protégées, réalisation et vérification de fonctions auto-organisées de stockage de données, systèmes à accès en écriture multiple, codes correcteurs, simulation et systèmes distribués de très grande taille.

- ARC BROCCOLI :
  Ce projet de recherche collaborative a débuté en 2008. Cette collaboration se fait par 3 équipes de recherche : ADAM de INRIA Futurs, ACMES de Télécom SudParis, et enfin MASCOTTE, avec en particulier la participation de Olivier Dalle (chef de projet) et de Judicael Ribault.
  Le but de ce projet est de concevoir une plate-forme pour la description, l'exécution, l'observation, l'administration et la reconfiguration d'architectures logicielles basées sur composants grande échelle. En particulier pour la création d'applications de simulation à évènements discrets distribuées sur des millions de noeuds de réseaux. Ce projet utilise le framework FRACTAL, et est directement à l'origine du logiciel Open Simulation Architecture (OSA) distribué par MASCOTTE.

7 LACL : Laboratoire d'Algorithmique, Complexité et Logique.

8 NS : Network Security.

- PArtition de GRaphes Orientés (PAGRO) :
  Pour ce projet de recherche MASCOTTE est associé avec l'équipe VAG du LIRMM, située à Montpellier. Voici les membres MASCOTTE participants à ce projet : Marie Asté, Jean-Claude Bermond, Frédéric Havet, Florian Huc, Christelle Molle, Stéphane Pérennes, Ignasi Sau-Valls.
  Ce projet traite de différentes questions ouvertes concernant le partitionnement de graphes orientés, comme manière d'établir la complexité de la structure d'un graphe.

- LAbel REduction for P2MP and MP2MP COmmunications (LARECO) :
  Débuté en 2008, ce projet associe MASCOTTE à l'institut et université de Girona en Catalogne. Les membres MASCOTTE travaillant sur ce projet sont David Coudert, Joanna Moulierac et Ignasi Sau-Valls.
  Le but de LARECO est d'étudier le problème de réduction du nombre de labels utilisés dans l'établissement de communications (Point-à-Point, Point-à-MultiPoint – P2MP, MultiPoint-à-Point et MultiPoint-à-MultiPoint – MP2MP) dans les réseaux GMPLS[9] et AOLS[10]. Dans ces réseaux, les labels sont utilisés par les noeuds pour décider du prochain saut de chaque paquet et impliquent ainsi des ressources et du temps de traitement. Le coût de AOLS augmente donc avec le nombre de labels, puisqu'un périphérique optique spécifique est nécessaire pour chacun des types de labels des noeuds. De ce fait, réduire le nombre de labels permettrait de réduire l'équipement nécessaire au niveau des noeuds. L'objectif principal du projet est d'établir des résultats de complexité, de concevoir des algorithmes exacts et d'approximation et de développer des algorithmes heuristiques.

- Enfin, Mascotte est impliqué dans le Contrat de Recherche Collaborative CORSO II avec France Telecom R&D. Initié en 2003, ce projet s'intéresse aux thèmes suivants : calcul de disponibilité et sécurisation, réseaux UMTS[11], réseaux maillés radio, conception de réseaux. Au sein de Mascotte ce contrat implique : Jean-Claude Bermond, David Coudert, Michel Syska, Stéphane Pérennes et Jérome Galtier.

9 GMPLS : Generalized Multi-Protocol Label Switching.

10 AOLS : All-Optical Label Swapping.

11 UMTS : Universal Mobile Telecommunications System, une des technologies de la téléphonie 3G.

II PREMIÈRE PARTIE DU STAGE : intégrer un solver CLP/CBC à MascOpt

MascOpt (MASCOTTE Optimisation) est un des logiciels développés au sein de l'équipe-projet MASCOTTE. C'est sur le développement d'une partie bien précise de ce logiciel que j'ai travaillé durant la première partie de mon stage. Il s'agissait d'intégrer un solver de programmes linéaires (CLP[12]/CBC[13]) à la librairie MascOpt.

Pour bien comprendre les motivations de ce travail, il faut savoir que MascOpt disposait déjà a priori de plusieurs solvers de programmes linéaires :
Ilog CPLEX qui est reconnu comme le leader dans le domaine, est très performant, fournit une API Java (nommée Concert), ce qui est parfaitement adapté à l'objectif de portabilité de MascOpt mais un désavantage réside dans le fait que cet outil est sous licence propriétaire. Désavantage il y a car l'utilisation de cet outil sur un réseau est limité par le nombre de licences d'exploitation que l'on s'est procuré, c'est-à-dire qu'il est difficile de distribuer la résolution de programmes linéaires. Sachant de plus que le coût des licences CPLEX est très élevé, surtout pour l'industrie. Enfin, restreindre MascOpt à l'utilisation de CPLEX va à l'encontre de la volonté qu'a l'INRIA de promouvoir les logiciels libres.
GNU Linear Programming Kit (GLPK) : par opposition à CPLEX, cet outil est « libre » (GNU General Public License) mais reste peu performant.

D'où l'intérêt d'utiliser les librairies CLP et CBC, solvers de programmes linéaires qui sont fournies par COIN-OR[14], qui sont « libres » (Common Public License) et qui permettent des performances[15] plus ou moins équivalentes à CPLEX. On aurait donc tous les avantages.

Cependant un problème réside dans ce choix : CLP et CBC sont des librairies/outils développés en C++, or MascOpt est développé en Java.

La solution à ce problème, et par là le principal objectif de mon travail consistaient à : utiliser la spécification Java Native Interface de Sun – qui en bref permet l'utilisation de code natif en Java, pour intégrer à MascOpt une interface d'utilisation de CLP et CBC, en respectant évidemment l'architecture de solver générique déjà mise en place dans la librairie MascOpt.

Dans cette partie, je présenterai donc l'analyse nécessaire et la réalisation de ce travail. Mais avant je présenterai avec un détail relatif les outils utilisés pour mener à bien ce développement, ainsi que les librairies MascOpt, CLP et CBC.

12 CLP : Coin-or Linear Programming.
13 CBC : Coin-or Branch and Cut.
14 COIN-OR : Computational Infrastructure for Operationnal Research ; http://www.coin-or.org.
15 Performances comparatives de solvers : voir le *benchmark* à l'adresse http://plato.asu.edu/ftp/milpf.html.

1. Présentation de MascOpt

1.1 Présentation générale

Le but principal du projet MascOpt est de fournir un ensemble d'outils traitant des problèmes d'optimisation réseau.

Voici quelques exemples de ces problèmes :

- routage (multiflot) d'un noeud réseau à un autre sur un réseau limité par les capacités associées à ses segments,
- routage avec contraintes de vulnérabilité (protection et restauration),
- multiplexage par *grooming* (par exemple avec les standards SDH[16] et WDM[17] sur réseaux optiques)...

MascOpt fournit à l'utilisateur :

- un modèle de données de graphes,
- des librairies pour analyser, charger, stocker ces données,
- des librairies pour le calcul de routage via des algorithmes connus ou des programmes linéaires,
- un outil de visualisation graphique des résultats,
- d'autres outils visant à faciliter le développement d'algorithmes traitant de problèmes réseaux...

MascOpt est portable, avec l'utilisation de formats XML et du langage Java. MascOpt est par ailleurs un logiciel libre distribué sous licence LGPL (GNU Lesser General Public License).

1.2 Les programmes linéaires

Comme indiqué précédemment, MascOpt fait une utilisation des programmes linéaires, pour la résolution de problèmes de routage. Il s'agit donc ici de définir simplement ce qu'est un programme linéaire et de donner quelques indications quant à ses caractéristiques et à sa résolution.

Un programme linéaire est un problème qui consiste à minimiser ou maximiser une fonction linéaire, dite fonction objectif, en respectant un certain nombre de contraintes (des inégalités ou des égalités) qui sont également linéaires.

16 SDH : Synchronous Digital Hierarchy.

17 WDM : Wave Division Multiplexing.

Un programme linéaire peut être fractionnaire, c'est-à-dire que les variables des termes des expressions du programme sont fractionnaires, mais il peut également être entier ou mixte.

Les algorithmes permettant sa résolution sont : *simplex*[18], *interior point*, *branch and bound* (spécialement pour les programmes en nombres entiers).

Voici un exemple de programme linéaire où il faut maximiser une fonction objectif, sous des contraintes qui sont des bornes supérieures sur des expressions linéaires avec des variables fractionnaires positives ou nulles.

$$
\begin{aligned}
&Max \quad 2x_1 + 4x_2 + x_3 \\
&S.c: \\
&\quad x_1 + x_2 \leq 100 \\
&\quad x_1 + 2x_2 + 3x_3 \leq 498 \\
&\quad x_1, x_2, x_3 \geq 0
\end{aligned}
$$

1.3 Le paquetage Java mascoptLib.lpSolver

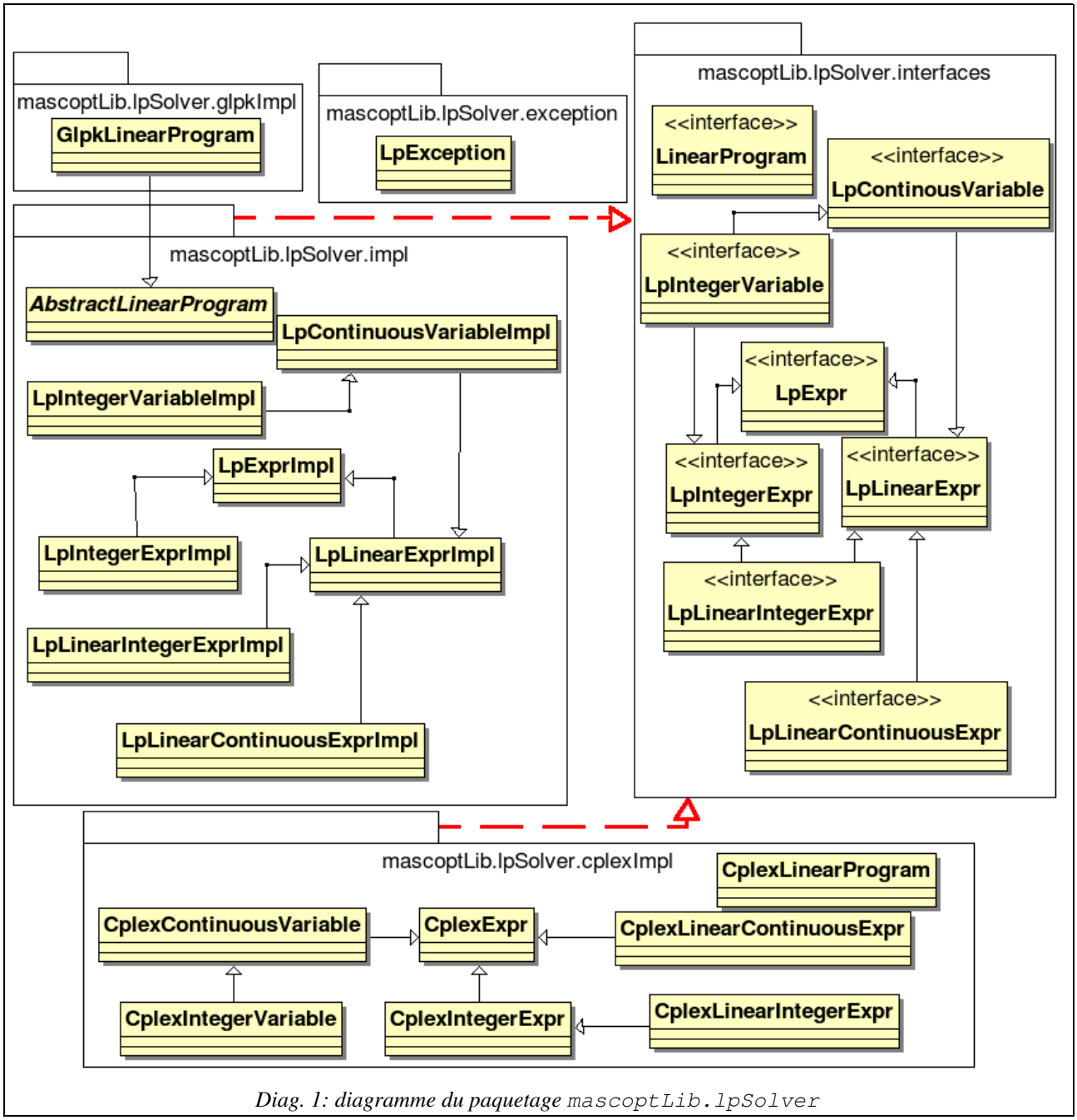
Le paquetage `mascoptLib.lpSolver` de la librairie MascOpt, sur lequel j'ai travaillé, est celui qui définit et implémente tout ce qui concerne les programmes linéaires et leur résolution. Les solvers CPLEX et GLPK sont utilisés via ce paquetage. Le diagramme UML exposé plus bas décrit les sous-paquetages de `mascoptLib.lpSolver`.

Le paquetage `interfaces` contient les interfaces qui représentent tous les éléments nécessaires à la modélisation d'un programme linéaire et à sa résolution :

- les variables : `LpContinuousVariable` (variables fractionnaires), `LpIntegerVariable` (variables entières).
- les expressions : avec les interfaces `LpExpr`, `LpLinearExpr` (linéaires), `LpIntegerExpr`, `LpLinearIntegerExpr`, `LpLinearContinuousExpr`.
- Enfin l'interface `LinearProgram` qui représente les opérations permettant de créer à proprement parler le programme linéaire ; d'y ajouter des contraintes linéaires, de minimiser ou maximiser une fonction objectif et de résoudre (éventuellement) le programme, récupérer la valeur objective après résolution...

18 *Simplex* : en partant du problème primal ou dual (le dual étant formé à partir du primal par transposition des matrices, et réciproquement).

Le paquetage `impl` contient les classes qui implémentent les interfaces du paquetage précédant, dont la plus importante `AbstractLinearProgram` qui est abstraite. Il contient aussi les paquetages `glpkImpl` et `cplexImpl` qui sont les implémentations contenant les solvers réels. À noter que `glpkImpl` implémente le paquetage `impl` alors que `cplexImpl` implémente directement le paquetage `interfaces`. Cela s'explique par le fait que `AbstractLinearProgram` est réservée à l'implémentation des solvers de programmes linéaires en représentation matricielle alors que `LinearProgram` est adaptée à la représentation par objets.



Diag. 1: diagramme du paquetage `mascoptLib.lpSolver`

2. Présentation des outils utilisés

Dans ce chapitre sont présentés les principaux outils utilisés pour la réalisation de ce projet. Les langages de programmation C, C++ et Java qui étaient imposés. L'environnement de développement intégré Eclipse, qui a été choisi. Et enfin, Java Native Interface (JNI), outil que j'ai dû apprendre au cours du stage, contrairement aux précédents qui m'ont tous été enseignés en LP SIL ou ailleurs. L'API JNI sera en conséquence plus détaillée dans sa présentation.

2.1 Langages C et C++

C et C++ ont été utilisés pour développer le « module » JNI pour l'intégration à MascOpt du solver CLP/CBC (librairies qui sont développées en C++). Ci-dessous, une présentation brève de ces deux langages.

Commençons par le plus ancien, le C, il a été conçu au début des années 70 en parallèle du développement du système UNIX[19]. C'est un des langages les plus connus et utilisés. Sa syntaxe est à la base de nombreux autres langages qui lui ont fait suite, dont avant tout le C++.
Le C est un langage impératif et procédural.
C'est un langage de bas niveau, du fait de ces types proches de la machine puisque directement basés sur la taille des mots mémoire, ainsi que par sa gestion et son accès direct à la mémoire, gérés par le programmeur avec l'utilisation de pointeurs. C'est notamment pour cette raison que ce langage est utilisé principalement pour la programmation système et dans l'industrie pour le développement embarqué. D'autant plus qu'il est plus simple à utiliser que l'assembleur (qui peut tout de même être inséré par fragments dans du C), et permet évidemment une portabilité des programmes plus simple à mettre en oeuvre.
Il existe plusieurs normalisations du C, la première ANSI 89, la deuxième ISO 90, et la dernière en 99 qui reprend des éléments du C++.

En effet, C et C++ sont très liés techniquement et historiquement. C++ ayant été conçu[20] comme une amélioration du C. Sorti en 85, le C++ conserve les paradigmes du C, l'approche bas niveau mais ouvre de nombreuses et nouvelles possibilités et de nouveaux concepts.
Avec en premier lieu la programmation orientée objet qui permet de meilleures (ou plus simples) lisibilité et réutilisabilité des composants logiciels développés. Le paradigme orienté objet pousse également plus loin l'abstraction des applications, il est par exemple plus intuitif de modéliser un programme linéaire avec l'approche objets (la fonction objectif, les contraintes, les variables... sont intuitivement des objets).

19 Travaux effectués par Dennis Ritchie et Ken Thompson, au sein des laboratoires Bell.
20 Bjarne Stroustrup est l'auteur du langage, sous l'égide des laboratoires Bell.

On retrouve ainsi la plupart des fonctionnalités des langages orientés objet et d'autres plus spécifiques au C++ : classes, encapsulation des données, héritage, polymorphisme, généricité et transtypage (statique et dynamique), espaces de nommage, surcharge de méthodes et d'opérateurs, exceptions...
Le langage C++ permet l'utilisation de nombreuses librairies : toutes celles développées en C d'une part, et les nombreuses autres développées en C++ d'autre part. Notamment la librairie Standard Template Library, fournissant des classes génériques de conteneurs, d'itérateurs, et d'algorithmes de gestion de données très efficaces.
Tout comme le langage C, le C++ est normalisé par l'ISO et est très répandu à travers le monde.

Pour finir, voici les quelques problèmes souvent reprochés aux langages C et C++ : avant tout les complications dues à la gestion mémoire, source de bogues voire de failles graves de sécurité (débordement de tampons dans la pile mémoire d'un programme notamment). En second plan, on reproche au C/C++ la mise en oeuvre parfois fastidieuse de la portabilité des programmes (il faut pour cela souvent encombrer le code source de directives préprocesseur). Enfin, C/C++ pâtit d'une syntaxe et d'une sémantique parfois relativement lourdes, compliquées, pouvant même paraître contradictoire (par exemple : le mot clé `friend` qui peut faire exception à l'encapsulation – avec `private` – d'une classe donnée).
Ces défauts de C/C++ sont probablement liés à une approche de conception par empilement de fonctionnalités sans cohérence générale et sont une des motivations de l'arrivée du langage Java.

2.2 Langage Java

Le langage Java a été utilisé pour l'ajout de deux classes représentant le solver CLP/CBC au sein de MascOpt. Une de ces classes faisant le lien entre MascOpt et le « module » JNI correspondant. L'autre implémente l'interface imposée par MascOpt et qui définit ce que doit faire un solver de programme linéaire.

Java est sorti en 1995, il a été conçu par James Gosling et Patrick Naughton employés de Sun Microsystems.

C'est un langage dit semi-compilé, c'est-à-dire qu'il se situe entre l'interprété des scripts et le compilé en langage machine des exécutables binaires. Ce code intermédiaire est le bytecode, il est exécuté par la machine virtuelle Java, qui elle va exécuter les instructions processeur correspondantes au bytecode.
Ce fonctionnement permet une portabilité pratiquement à toute épreuve, étant donné qu'il suffit d'avoir une machine virtuelle disponible sur son système d'exploitation pour pouvoir exécuter un programme compilé sur toute autre plate-forme, de nombreux OS ont leur implémentation Sun de la

machine virtuelle. C'est une des raisons qui ont fait le choix de Java pour le développement de MascOpt.

Java est un langage de la même famille que C++ ; ce qui est un avantage pour passer de l'un à l'autre. Mais Java est plus rigoureusement orienté objet et surtout de manière plus limpide, avec une syntaxe plus simple.

Concernant la mémoire : la machine virtuelle Java est équipée d'un collecteur de déchets qui s'occupe de libérer les ressources mémoires allouées aux objets qui sont hors d'usage dans le programme − i.e. un objet dont plus aucune référence n'est accessible dans le ou les fils d'exécution de l'application. Cette gestion simple (pour le programmeur) de la mémoire peut être avancée comme une amélioration vis-à-vis des problèmes encourus avec C++. En Java les destructeurs qu'on avait en C++ n'existent pas[21], le collecteur de déchets doit s'occuper automatiquement de la désallocation mémoire.

Enfin, à remarquer que l'API standard de Java est très riche, ce qui est un atout considérable car c'est avant tout une facilité pour développer des applications sans faire des agrégats de librairies diverses comme on peut le faire en C++.

2.3 EDI Eclipse

J'ai travaillé avec l'environnement de développement intégré Eclipse tout au long du stage. Ce qui d'entrée témoigne de la polyvalence d'Eclipse.

L'environnement Eclipse est distribué depuis 2001. Il est à l'origine un projet IBM, qui est passé par la suite dans la « communauté du libre ». Depuis, plusieurs versions se sont succédées, la dernière version 3.4 Ganymede est sortie en Juin dernier, pour ma part j'ai utilisé la version 3.3 Europa.

Eclipse est très lié au développement Java, mais comme je le disais il est très polyvalent. En fait, il permet de développer dans de nombreux langages de programmation. De nombreux plugins sont développés, pour faire toute sorte de choses dans un même environnement. Eclipse est un environnement extensible. Facile et efficace dans sa prise en main, avec la notion de perspective,

21 Il n'y a pas de desctructeurs en Java, cependant la méthode Object.`finalize()` peut être définie pour chaque classe, et ainsi spécifier certaines opérations à effectuer en fin de vie des objets.

chaque plugin est associé à une perspective, on bascule de l'une à l'autre selon qu'on veut spécifiquement programmer en Java, déboguer un programme, écrire un programme en C++... Chacune des perspectives étant composées de vues, qui sont des panneaux très ergonomiques, que l'on peut déplacer à sa convenance. L'interface graphique de Eclipse a été conçue avec l'API Standard Widget Toolkit (SWT) développée par IBM, c'est une librairie Java (portable) qui utilise des composants génériques mais qui est basée sur les différentes librairies graphiques natives.

Au cours du stage, j'ai utilisé principalement les plugins Java Development Toolkit (JDT), C/C++ Development Toolkit (CDT) avec le compilateur GNU , Subclipse faisant office de client graphique SVN[22] intégré à Eclipse, et enfin m2eclipse pour l'utilisation de Maven (cf. partie III).

Dernier point remarquable à propos d'Eclipse : en plus de sa polyvalence il n'altère pas la structure d'un projet, on peut décider à tout moment de faire migrer un projet Eclipse vers un autre environnement sans que Eclipse ne complique ou ne rende impossible cette opération.

2.4 Java Native Interface

2.4.1 Présentation

Java Native Interface (JNI) est une spécification standard de Sun qui permet d'exécuter du code natif via des appels Java. Concrètement, la machine virtuelle Java peut charger des librairies dynamiquement, lesquelles contiennent les définitions (en code natif) de méthodes Java. Au cours de l'exécution d'un programme Java, lorsque qu'une méthode d'implémentation native est rencontrée, la machine virtuelle va exécuter la méthode native de symbole correspondant, méthode qui est fournie par la librairie (module JNI) chargée au préalable.

Il y a plusieurs intérêts à utiliser JNI :

- Améliorer les performances : en effet, la machine virtuelle reste plus lente que du code natif malgré le mécanisme de compilation Just In Time (JIT). JNI est donc très utile pour le développement de composants bien précis d'une application, composants nécessitant des performances accrues.
- Développer une API Java basée sur des librairies natives existantes : on parle de *wrappers* (enveloppes), car cela consiste grossièrement à créer des liens entre des déclarations de méthodes Java et des méthodes natives qui sont leurs implémentations définies dans une librairie. Un exemple de cette utilisation est l'API Java JOGL, qui est un *wrapper* de la

22 SVN : SubVersioN est un système de gestion concurrente de versions d'un programme, spécialement adapté au développement collaboratif. Il maintient de manière incrémentielle les versions d'un programme en cours de développement.

librairie native (bien que portable) OpenGL.

- Interagir avec le matériel : Java étant portable il produit du code géré par la machine virtuelle, donc sans accès direct au matériel. On a besoin de JNI pour développer des applications et des API Java qui vont pouvoir opérer au niveau matériel. Par exemple, pour utiliser la communication sur liaison série RS232 d'un PC.
- Une dernière utilisation qui est plus rare - mais qui dévoile un aspect clé de JNI : on peut utiliser JNI pour utiliser l'API standard Java, dans une application développée en C/C++. En effet, JNI permet à des appels Java d'être liés à des appels natifs, mais cette spécification permet également de créer des objets Java, faire des appels aux méthodes Java, déclencher des exceptions Java... Cette dernière utilisation peut être utile pour un programmeur peu familier des librairies C/C++ qui peuvent être plus compliquées d'utilisation que l'API Java ; exemple : développer des applications réseaux (création de *sockets*) est beaucoup plus aisé en Java qu'en C (programmation système).

Quelques mots sur la portabilité : il est évident que l'utilisation de bibliothèques natives, en l'occurence des « modules JNI », exige de reconsidérer la portabilité de son application. Elle n'est plus automatique dès lors qu'on utilise du code natif. Pour conserver cette portabilité avec JNI, il faut que le code C/C++ (JNI) soit rendu portable et qu'il soit compilé pour chacune des architectures cibles.

2.4.2 Mise en oeuvre

Je décris ci-dessous les étapes générales à réaliser pour mettre en oeuvre un composant logiciel Java d'implémentation JNI :

- Déclarer la classe et surtout les méthodes Java que l'on veut implémenter en C/C++ via le module JNI. Ces méthodes doivent contenir le mot-clé Java `native` dans leurs déclarations ; on les appelle les méthodes natives.
- Générer le fichier en-tête C, qui contient les déclarations en C des méthodes natives Java que l'on a déclarées. Pour cette génération, on utilise l'utilitaire javah fourni dans le JDK[23]. Voici la commande pour utiliser cet utilitaire :
  `javah –classpath` *`Classpath_de_la_classe_Java`* `–o` *`Nom_de_fichier_en-tête.h Nom_complètement_qualifié_de_la_classe_Java`*
- Implémenter les méthodes natives : sur la base du fichier d'en-tête généré – que l'on inclut dans le fichier C ou C++ de définition, on définit les méthodes natives.
- Compiler le code JNI d'implémentation : à ce stade il suffit de compiler le composant JNI. Il faut spécifier au compilateur que l'on souhaite obtenir une librairie dynamique partagée

23 JDK : Java Development Toolkit, tout JDK fournit un compilateur, une machine virtuelle Java, bref un environnement de développement rudimentaire, ainsi que des utilitaires dont javah.

(dont l'extension du fichier est `.so` sous Linux et `.dll` sous Windows). Avec le compilateur GNU GCC, que j'ai utilisé durant le stage, la commande type à cet effet est : `gcc` *`fichier_JNI.c`* `-shared -I`*`/chemin/en-tetes/divers`* `-L` *`/chemin/des_eventuelles/librairies`* `-o` *`libJNI.so`*

- Enfin, il faut rajouter le code de chargement de la librairie partagée que l'on vient de compiler dans la classe Java contenant les méthodes natives. Il n'est pas obligatoire de le rajouter dans cette classe, mais il est plus efficace (au niveau de la gestion des ressources) de charger la librairie seulement lorsque la machine virtuelle chargera la classe Java qui en a réellement besoin.
  Voici le code à placer dans le constructeur ou (de préférence) dans un bloc `static` de la classe Java : `System.loadLibrary("`*`libJNI.so`*`");`

La démarche générale de développement d'un composant JNI est donc assez aisée. Cette démarche décrit les étapes à réaliser pour une seule classe. Mais de manière analogue on peut le faire pour plusieurs classes dont le code natif pourrait être contenu dans une même et seule librairie partagée. Cependant l'implémentation JNI – l'écriture du code à proprement parler – est assez lourde de part les mécanismes à mettre en oeuvre pour interagir avec la machine virtuelle Java.

2.4.3 Pratique de JNI en détails

Il ne s'agit pas de détailler tout ce que l'on peut faire avec JNI, cela serait bien trop vaste. En effet, l'API JNI contient de nombreuses fonctions (des fonctions en langage C) pour remplir diverses opérations. Je n'expose donc ici que les principales dont j'ai eu besoin.

a) Prérequis pour utiliser JNI

Avant d'aborder les fonctions JNI, il y a plusieurs généralités de fonctionnement de JNI à connaître. Tout d'abord concernant javah : lorsqu'il effectue la génération des en-têtes C en correspondance avec les méthodes natives Java à implémenter, il associe une fonction C à chaque méthode Java. Les fonctions C générées sont différentes selon que la méthode native Java associée est une méthode `static` ou non.

Pour une méthode Java non `static` (qui est donc appelable via son instance), javah génère le code type suivant :

```
JNIEXPORT type_retour JNICALL Java_Paquetage_Classe_methode(JNIEnv *,
                                jobject,...);
```

Quelques remarques :

- Les mots-clés `JNIEXPORT` et `JNICALL` sont des extensions JNI. `JNIEXPORT` indique que la fonction concernée est exportée (c'est une fonction de librairie partagée). `JNICALL`

indique assez intuitivement que cette fonction C est une implémentation d'une méthode native Java.

- Sur le nom de la fonction C générée : javah nomme chaque fonction qu'il génère selon une nomenclature bien précise. Cela permet à la machine virtuelle Java de déterminer spécifiquement quelle fonction aller chercher dans la librairie (implémentation JNI) projetée en mémoire, lorsqu'elle doit exécuter une méthode native Java. Cette nomenclature impose que le nom de la fonction commence par Java, suivi des noms des différents paquetages contenant la classe de la méthode et se termine par le nom de la méthode native Java en question (les différents éléments étant séparés par le caractère tiret bas).
- Les arguments de la fonction C générée : le premier de type `JNIENV*` est un pointeur vers une structure C qui contient tous les pointeurs de fonctions qui constituent l'API JNI, elle permet d'interagir avec la machine virtuelle. Le second de type `jobject` fait référence à l'objet Java auquel appartient la méthode native Java correspondante à la fonction C. On peut utiliser cet objet pour différentes opérations, que je détaille plus bas. Les autres arguments qui peuvent suivre (représentés ici par "...") sont les arguments de la méthode Java native. On a donc accès plus ou moins directement à ces arguments.
- Le type de retour de la fonction : il correspond au type de retour de la méthode Java native correspondante.

Pour une méthode Java `native` et `static` le prototype de fonction C généré est :

```
JNIEXPORT type_retour JNICALL Java_Paquetage_Classe_methode(JNIENV*,
                                jclass,...);
```

La seule différence avec le prototype précédent est que pour ce type de fonction on obtient, en deuxième argument, une référence sur la classe de la méthode Java native. Cela est évident car cette fonction est associée à une méthode Java statique.

Enfin, il faut préciser que les types Java sont associés à des types C qui sont définis par la spécification JNI. Tous ces types JNI sont utilisés en C/C++ pour utiliser (via la machine virtuelle) des variables et des objets qui proviennent du code Java ou y sont créés.
Ce tableau résume cette correspondance de types :

Type Java	***Type JNI (ou C)***
`void`	`void`
`boolean`	`jboolean`
`char`	`jchar`
`byte`	`jbyte`
`short`	`jshort`
`int`	`jint`
`long`	`jlong`

Type Java	*Type JNI (ou C)*
`float`	`jfloat`
`double`	`jdouble`
`Object`	`jobject`
`Class`	`jclass`
`String`	`jstring`
`Object[]`	`jobjectArray`
`bool[]`	`jbooleanArray`
`byte[]`	`jbyteArray`
`char[]`	`jcharArray`
`short[]`	`jshortArray`
`int[]`	`jintArray`
`long[]`	`jlongArray`
`float[]`	`jfloatArray`
`double[]`	`jdoubleArray`
`Throwable`	`jthrowable`

Malheureusement, il n'est pas évident d'accèder à tous les éléments du code Java en JNI. En effet, ce mécanisme nécessite une spécification fine qui permet d'indiquer rigoureusement et efficacement à la machine virtuelle ce à quoi on veut accèder et comment.
JNI utilise donc des signatures, qui permettent par exemple de désigner le type d'un attribut Java auquel on veut accèder depuis C/C++ (côté JNI). Ainsi JNI attribue une signature à chaque type d'élément du code Java (que ce soit une donnée, une classe ou une méthode).
Le programmeur doit utiliser ces signatures lorsqu'il veut interagir avec ces éléments Java.
Voici le tableau qui présente l'association entre type Java et signature JNI :

Type/Méthode	**Signature**
boolean	Z
byte	B
char	C
short	S
int	I
long	J
float	F
double	D
classe	L/paquetage1/paquetage2/.../Classe
tableau	[type
méthode	(type1,type2,...,typeN)type_retour

Table 1: Signatures JNI

b) Quelques utilisations de fonctions de l'API JNI

Ici je montre comment on peut accéder à un attribut d'objet, appeler une méthode Java, créer un objet Java et déclencher une exception Java depuis le code C/C++ du composant JNI. Ces explications sont exposées étape par étape. À chaque étape correspond l'appel d'une fonction de l'API JNI qui y est accessible via la structure `JNIENV`. J'indique donc plus bas les méthodes correspondantes à chaque étape, mais sans redondance lorsque les étapes sont les mêmes. Des exemples sont joints aux explications. Par souci de concision les exemples ne contiendront pas les vérifications nécessaires à un code réellement rigoureux.

- Accéder (en lecture ou en écriture) à un attribut d'objet Java : pour poser le contexte, je rappelle que nous avons accès – via la fonction C JNI générée par javah – à la structure `JNIENV` et à une référence de l'objet Java auquel appartient l'attribut que l'on veut lire ou écrire.
  - On récupère la classe de l'objet Java auquel appartient l'attribut :
    ```
    jclass GetObjectClass(jobject obj);
    ```
  - On récupère l'identifiant JNI de l'attribut à partir de sa classe :
    ```
    jfieldID GetFieldID (jclass clazz, const char * name, const char *sig);
    ```
    On doit également indiquer le nom, et la signature JNI (cf. tab 1) de l'attribut.
  - À partir de l'identifiant de l'attribut on récupère sa valeur, en précisant l'objet :
    ```
    type_natif Get<type>Field(jobject obj, jfieldID fieldID);
    ```
    Où on remplace `<type>` par le type de l'attribut (exemple `Int` pour le type `int`). `type_natif` est un type natif du langage C.
  - Ou on peut, de la même manière, lui affecter une nouvelle valeur :
    ```
    void Set<type>Field(jobject obj, jfieldId fieldId, type_natif value);
    ```

```
JNIEXPORT void JNICALL Java_examplePackage_methode(JNIENV * env,  jobject obj)
{
      jclass clazz;
      jfieldID fid;
      long attribut;

      clazz = env->GetObjectClass(obj); //on récupère la classe
      fid  = env->GetFieldID(clazz, "attribut", "J"); //l'id. de l'attribut
      attribut = (long) env->GetLongField(obj, fid); //sa valeur
      attribut += 10; //on modifie la valeur récupérée
      env->SetLongField(obj, fid, attribut); //on affecte la valeur à l'attribut
}
```

Exemple 1: lecture puis affectation d'un attribut d'objet Java via JNI

- Appeler une méthode d'objet Java :
  - On récupère la classe de l'objet Java auquel appartient la méthode.
  - On récupère l'identifiant JNI de la méthode à partir de sa classe :

    ```
    jmethodID GetMethodID(jclass clazz, const char * name, const char *
                                        sig);
    ```

    On doit également indiquer le nom, et la signature JNI (cf. tab 1) de la méthode.
  - On peut ensuite appeler la méthode :

    ```
    type_natif Call<type>Method(jobject obj, jmethodID methodID, ...);
    ```

    Les arguments suivants le second argument sont les paramètres à passer à la méthode Java lors de son appel.

```
JNIEXPORT void JNICALL Java_examplePackage_methode(JNIENV * env,  jobject obj)
{
      jclass clazz;
      jmethodID mid;
      jint valeur = 2222222;
      clazz = env->GetObjectClass(obj); //on récupère la classe
      mid = env->GetMethodID(clazz, "methodeJava", "(I)V"); //l'id. de méthode
      //La signature (I)V correspond à une méthode de type void methode(int);
      env->CallVoidMethod(obj, mid, valeur); //on appelle de la méthode Java
}
```

Exemple 2: appel d'une méthode d'un objet Java via JNI

- Créer un objet Java :
  - Il faut récupérer la classe de l'objet Java à instancier :

    ```
    jclass FindClass(const * char sig);
    ```

    Il faut indiquer la signature de la classe Java (avec le nom complètement qualifié, les paquetages étant séparés par le caractère /).
  - Il faut récupérer l'identifiant du constructeur comme on le fait pour une méthode classique, le nom du constructeur est "<init>".
  - On peut ensuite instancier l'objet, en spécifiant la classe Java et l'identifiant de son constructeur :

    ```
    jobject NewObject (jclass clazz, jmethodID methodID, ...);
    ```

    À partir du troisième argument, ce sont les paramètres à passer au constructeur.

```
JNIEXPORT void JNICALL Java_examplePackage_methode(JNIENV * env,  jobject obj)
{
      jclass clazz;
      jmethodID mid;
      jobject instance;

      clazz = env->FindClass("paquetageClasse/NomClasse"); //récupérer la classe
      mid = env->GetMethodID(clazz, "<init>", "()V"); //l'id. du constructeur
      instance = NewObject(clazz, mid); //créer l'objet Java
      /* Ensuite on peut utiliser l'objet au besoin
       * ex. : modifier ses attributs, appeler ses méthodes */
}
```

Exemple 3: instancier un objet Java depuis un composant JNI

- Déclencher une exception Java : c'est un mécanisme à la fois puissant et simple d'utilisation que permet JNI.
  - Premièrement, on récupère la classe de l'exception à déclencher.
  - Deuxièmement, on déclenche l'exception :
    `void ThrowNew(jclass clazz, const char * message);`
    `message` est le message associé à l'exception déclenchée (on peut le récupérer en Java).

```
JNIEXPORT void JNICALL Java_examplePackage_methode(JNIENV * env,  jobject obj)
{
      jclass clazz;

      /* traitements divers menant au déclenchement éventuel d'une exception */

      clazz = env->FindClass("java/lang/Exception");
      env->ThrowNew(clazz, "Une exception s'est produite.");

}
```

Exemple 4: Déclencher une exception Java via JNI

3. Présentation des librairies/solvers de COIN-OR

COIN-OR signifie COmputational INfrastructure for Operations Research. COIN-OR regroupe de nombreux projets de logiciels libres. IBM est une compagnie membre des projets COIN-OR (des ingénieurs de chez IBM participent activement au développement).
Le leitmotiv de COIN-OR s'exprime ainsi :

> *Our goal is to create for mathematical software what the open literature is for mathematical theory.*[24]

24 Notre but est de créer pour les logiciels de mathématiques ce que la littérature libre est pour la théorie mathématique.

Les deux librairies CLP et CBC que j'ai eues à utiliser font partie de ces projets, je les présente ci-dessous avec des exemples de résolution d'un programme linéaire.

3.1 Librairie CLP

Basiquement la librairie CLP (pour COIN-OR Linear Programming) permet de résoudre des programmes linéaires en nombres fractionnaires (algorithme simplexe ou point interieur).

Cela est fait dans une approche orientée objet (langage C++). Ainsi une des classes centrales de l'utilisation de CLP est `ClpSimplex` qui dérive de `ClpModel`, permet de modéliser un programme linéaire et de le résoudre assez simplement.

Voici un programme qui illustre comment utiliser CLP pour résoudre le programme linéaire (très simple) présenté en exemple en 1.2 :

Exemple de modélisation/résolution de PL avec CLP

```
#include<stdlib.h>
#include<coin/ClpSimplex.hpp>
#include<iostream>
/* compilation avec gcc : g++ -lClp -lCoinUtils -o TestClp TestClp.cpp */
int main ()
{
    ClpSimplex * model;

    //coefficients de la fonction objectif
    const double objective[3] = {2,4,1};
    //index des variables (en partant de 0)
    //dont on veut affecter les coefficients
    const int indexes[5] = {0,1, 0,1,2};
    //valeurs des coefficients correspondants
    const double coeffs[5] = {1,1, 1,2,3};
    //bornes supérieures respectives des deux contraintes
    const double upperBounds[2] = {100,498};
    //indique où commencer à lire les données dans indexes et coeffs
    //resp. pour la première et la deuxième contrainte
    const CoinBigIndex rowStarts[2] = {0,2};
    //nb de variables resp. des contraintes
    const int rowLengths[2] = {2, 3};

    //instancier le modèle du PL
    model = new ClpSimplex();

    //annuler l'affichage automatique d'informations
    model->setLogLevel(0);

    //redimensionner la matrice du problème, on a 3 variables x1, x2, x3
```

Exemple de modélisation/résolution de PL avec CLP

```
    model->resize(0,3);

    //définir les variables comme positives (borne inférieure = 0)
    for(int i=0; i < 3; i++) {
        model->setColLower(i,0);
        model->setColUpper(i, DBL_MAX);
    }

    //définir la fonction objectif
    model->chgObjCoefficients(objective);

    //maximiser la fonction objectif
    model->setOptimizationDirection(-1);

    //ajouter les contraintes avec les bornes et les coefficients des variables
    model->addRows(2,NULL, upperBounds, rowStarts, rowLengths, indexes, coeffs);

    //résoudre le PL par le simplexe sur le primal
    model->primal();

    //afficher les solutions
    std::cout << "La valeur objective est : " << model->objectiveValue() <<
    std::endl << "Les solutions sont : ";

    for (int i=0;i < 3; i++)
        std::cout << "x" <<  i+1 << "=" << model->getColSolution()[i] << ", ";

    std::cout << std::endl;

    return EXIT_SUCCESS;
}
```

3.2 Librairie CBC

CBC (pour COIN-OR Branch and Cut) est une librairie permettant principalement de résoudre des programmes linéaires en nombres entiers (ou mixtes). Un algorithme nommé *Branch and Bound* est utilisé après une première phase de relaxation linéaire (résolution du programme linéaire en tant que programme fractionnaire).

Pour cette première phase CBC doit utiliser un solver continu, que ce soit CLP, CPLEX ou encore Dylp... Des interfaces sont fournies pour l'utilisation de ces différents solvers. Elles dérivent toutes de `OsiSolverInterface` qui fait en fait partie de la librairie OSI[25] qu'il faut donc brièvement

25 OSI : Open Solver Interface, librairie COIN-OR, qui est une API uniforme pour l'utilisation de solvers fractionnaires ou mixtes.

utiliser. Ici bien sûr on utilise CLP et c'est l'interface `OsiClpSolverInterface` qui correspond.

Pour ce qui est du code pour modéliser le programme linéaire, il est assez semblable à celui de CLP.
Voici un exemple de résolution du même programme linéaire que précédemment, avec les commentaires nécessaires (i.e. pas sur ce qui est déjà décrit dans l'exemple CLP) :

Exemple de modélisation/résolution de PL en nombres entiers avec CBC

```
#include<cstdlib>
#include<coin/OsiClpSolverInterface.hpp>
#include<coin/CbcModel.hpp>
#include<iostream>
/*compiler avec gcc : g++ -lCbc -lOsiClp -lClp -lCgl -lOsi -lCoinUtils -o
TestCbc TestCbc.cpp */
int main ()
{

    //interface de solver clp
    OsiClpSolverInterface * osi_clp;
    //"solver" clp
    ClpSimplex * clp;
    //"solver" cbc
    CbcModel * cbc;

    /** données nécessaires à la modélisaiton du PL
      * ce sont les mêmes dans l'exemple Clp
      */
    const double objective[3] = {2,4,1};
    const int indexes[5] = {0,1, 0,1,2};
    const double coeffs[5] = {1,1, 1,2,3};
    const double upperBounds[2] = {100,498};
    const CoinBigIndex rowStarts[2] = {0,2};
    const int rowLengths[2] = {2, 3};

    //donnée supplémentaire : index des variables entières (toutes)
    const int integerIndexes[3]={0,1,2};

    osi_clp = new OsiClpSolverInterface();

    //récupérer le pointeur sur le modèle CLP de l'interface
    clp = osi_clp->getModelPtr();

    clp->resize(0,3);

    //on définit les variables comme entières
    osi_clp->setInteger(integerIndexes, 3);

    for(int i=0; i < 3; i++) {
        clp->setColLower(i,0);
```

Exemple de modélisation/résolution de PL en nombres entiers avec CBC

```
        clp->setColUpper(i, DBL_MAX);
    }
    clp->chgObjCoefficients(objective);
    clp->setOptimizationDirection(-1);
    clp->addRows(2,NULL, upperBounds, rowStarts, rowLengths, indexes, coeffs);

    //on instancie le solver en nombres entiers
    cbc = new CbcModel(*osi_clp);

    cbc->setLogLevel(0);

    //relaxation linéaire
    cbc->initialSolve();
    //résoudre le PL
    cbc->branchAndBound();

    //afficher les solutions
    std::cout << "La valeur objective est : " << cbc->getObjValue() <<
    std::endl << "Les solutions sont : ";

    for (int i=0;i < 3; i++)
        std::cout << "x" <<  i+1 << "=" << cbc->getCbcColSolution()[i] << ", ";

    std::cout << std::endl;

    return EXIT_SUCCESS;
}
```

4. Analyse et réalisation

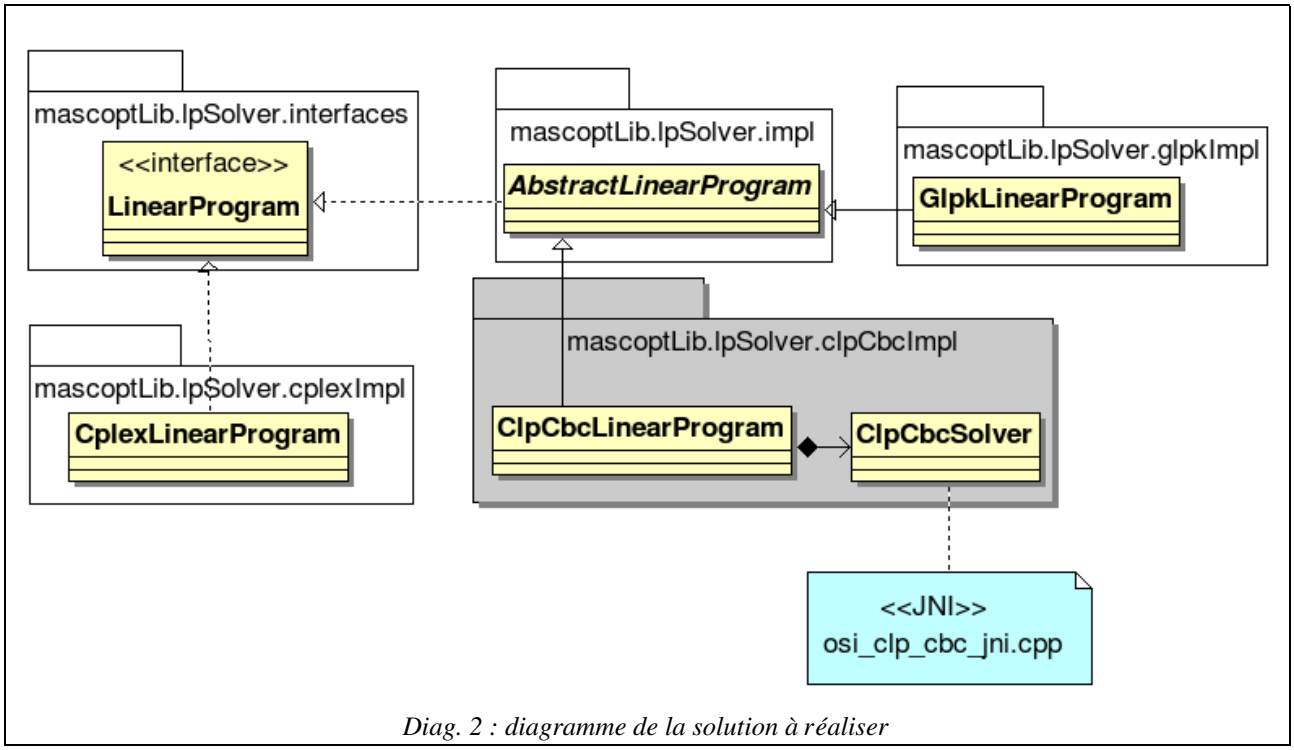
Après avoir vu ou revu les outils et librairies présentés en 2. j'ai pu travailler à résoudre le problème qui m'était posé.

Le but de ce chapitre est de faire un rappel des objectifs complété par l'analyse du problème et la réalisation de la solution, ainsi que les étapes concrètes de la réalisation de cette solution.

4.1 Solution à mettre en oeuvre

Suite à la présentation en 1.3 du paquetage de `mascoptLib.lpSolver` par son diagramme de classes, il devient évident qu'il faut fonctionner par analogie en se basant sur les implémentations CPLEX et GLPK pour réaliser l'intégration d'un solver CLP/CBC à MascOpt.
En effet, l'architecture logicielle du paquetage ne laisse pas de choix plus logique : il y a une interface générique définissant un solver de programme linéaire et il y a différentes implémentations dites réelles ou concrètes.

D'où le diagramme suivant qui résume la solution à mettre en oeuvre :



Diag. 2 : diagramme de la solution à réaliser

Ainsi, il faut rajouter un paquetage dédié au solver CLP/CBC : `mascoptLib.lpSolver.clpCbcImpl`. Ce paquetage doit contenir la classe `ClpCbcLinearProgram`, implémentation de la classe `AbstractLinearProgram` et par son intermédiaire de l'interface `LinearProgram`. Cette classe d'implémentation a une relation de composition avec la classe `ClpCbcSolver` qui représente le solver réel CLP/CBC. En somme, `ClpCbcLinearProgram` assure le respect de l'interface générique pour solver de MascOpt, et `ClpCbcSolver` contient tout ce qui est nécessaire (i.e. des méthodes) dans le solver réel (librairies CLP et CBC) pour pouvoir mener à bien cette implémentation (de `ClpCbcLinearProgram`).

Il va donc de soi que `ClpCbcSolver` est la classe qui contient les méthodes natives qui sont implémentées via JNI dans le composant `osi_clp_cbc_jni`, qui doit être compilé sous forme de librairie dynamique partagée et que la classe `ClpCbcSolver` devra charger à l'exécution.

4.2 Interface LinearProgram et classe AbstractLinearProgram

Pour réaliser la solution, la première nécessité est de comprendre les comportements définis ou simplement déclarés par l'interface `LinearProgram` et la classe `AbstractLinearProgram`. C'est donc la première étape que j'ai entamée durant le stage. Cela consiste simplement à lire le

code, comprendre les méthodes avec l'aide des implémentations existantes (CPLEX et GLPK) qui parfois m'ont été indispensables pour comprendre l'abstraction mise en place. Cela consiste également à comprendre comment `AbstractLinearProgram` construit génériquement un programme linéaire et éventuellement lance sa résolution.

Ci-dessous une partie de ce travail de compréhension ; tableau qui regroupe les méthodes à implémenter dans la classe `ClpCbcLinearProgram`.

Méthodes Java à implémenter dans ClpCbcLinearProgram		
Méthodes de AbstractLinearProgram		
Méthode	*Rôle*	*Arguments*
`setProblemClass()`	Définir le programme comme fractionnaire ou mixte.	`ProblemClass` : énumération dont les constantes définissent le type de problème.
`setObjectiveExpr()`	Définir la fonction objectif du programme.	`- int term` : nombre de termes de la fonction. `- double[] coefs` : les valeurs des coefficients de chaque terme. `- int[] variablesIndex` : les index des variables correspondants à `coefs`.
`setObjectiveSense()`	Définir si on veut maximiser ou minimiser la fonction objectif.	`ObjectiveSense` : énumération dont les constantes définissent maximisation et minimisation.
`setVariableNumber()`	Définir le nombre de variables du problème.	`int nbVariables` : nombre de variables.
`setVariableType()`	Définir le type d'une variable du programme : entier ou fractionnaire.	`- int variableIndex` : index de la variable. `- VariableType` : énumération définissant les 2 types.
`setVariableName()`	Définir le nom d'une variable.	`- int variableIndex` `- String name`
`setVariableBounds()`	Définir les bornes supérieures et inférieures du domaine de définition d'une variable.	`- int variableIndex` `- double lowerBound` : borne inférieure. `- double upperBound` : borne supérieure.
`setConstraintNumber()`	Définir le nombre de	`int nbConstraints`

	contraintes linéaires du programme.	
`addConstraint()`	Ajouter une contrainte au programme.	`- int constraintIndex` : index de la contrainte. `- int term` : nombre de termes de la contraintes. `- int[] varIndex` : les variables dont on veut affecter les coefficients. `- double[] varCoef` : valeurs des coefficients en correspondance à `varIndex`. `- double bound` : limite de la contrainte. `- ConstraintsType` : énumération qui définit les constantes indiquant si la contrainte est inférieure ou supérieure à `bound`. `- String name` : nom de la contrainte.
`callingSolve()`	Résoudre le problème avec utilisation de l'algorithme adapté à la nature du programme linéaire (entier ou mixte).	
`getVariableValue()`	Obtenir la valeur solution d'une variable .	`int variableIndex`
`writeClpexFile()`	Écrire le modèle du programme linéaire dans un fichier au format CPLEX (proche du format LP).	`String filename`
`writeMpsFile()`	Écrire le modèle dans un fichier au format MPS	`String filename`
Méthodes de LinearProgram		
`setDoubleOption()`	Positionner une option dont la valeur est réelle.	`- DoubleOption` : énumération qui définit les options à valeur réelle ; `TIME_LIMIT` (le temps maximum pour la résolution d'un programme), et `SOLUTION_GAP` (pour l'intervalle de

		tolérance de la solution vis-à-vis de l'optimale, dans le cas de PL mixtes). - `double value` : valeur de l'option.
`setIntegerOption()`	Positionner une option dont la valeur est entière.	- `IntegerOption` : énumération qui définit les options entières ; `SOLVER_ALGO` pour l'algorithme de résolution à utiliser. - `int value` : valeur de l'option.
`clearModel()`	Réinitialiser/effacer le modèle.	

4.3 Classes et méthodes CLP/CBC à utiliser

Une fois les méthodes à implémenter comprises. Il faut trouver comment satisfaire les exigences de chacune de ces méthodes par les API CLP et CBC (ainsi que brièvement dans OSI, l'interface générique de résolution de solver évoquée précédemment).

Au cours du stage, c'est à ce stade que j'ai commencé à étudier CLP/CBC (guides et documentations des API). Il a fallu déterminer quelles classes et quelles méthodes étaient nécessaires. Hormis les quelques problèmes précis la principale difficulté résidait dans la différence d'approche entre `AbstractLinearProgram` de MascOpt et CLP/CBC dans la représentation des programmes linéaires.

Ce sont principalement les classes C++ `ClpSimplex`, `CbcModel` et `OsiClpSolverInterface`, déjà présentées en 3., qui ont été utilisées.

Le tableau suivant résume la correspondance qu'il fallait établir entre les méthodes Java à implémenter et celles qui sont nécessaires au sein de l'API CLP/CBC pour cette implémentation.

Correspondance entre méthodes ClpCbcLinearProgram et méthodes CLP/CBC/OSI	
Méthode ClpCbcLinearProgram	**Méthode CLP/CBC/OSI**
`setObjectiveExpr()`	`ClpSimplex::setObjectiveCoefficient()`
`setObjectiveSense()`	`OsiClpSolverInterface::setObjSense()`
`setVariableNumber()`	`ClpSimplex::resize()`
`setVariableType()`	`OsiClpSolverInterface::setContinuous()` `OsiClpSolverInterface::setInteger()`
`setVariableName()`	`ClpSimplex::setColumnName()`

`setVariableBounds()`	`OsiClpSolverInterface::setColBounds()`

`addConstraint()`	`ClpSimplex::addRow()`
	`OsiClpSolverInterface::setRowBounds()`
	`ClpSimplex::setRowName()`

`callingSolve()`	`CbcModel::initialSolve()` : pour la relaxation linéaire. `CbcModel::branchAndBound()` `CbcModel::status()` : indique le statut de la résolution après tentative ; utile en cas d'erreur. `ClpSimplex::primalFeasible()` : indique si il y a réalisabilité du programme primal. `ClpSimplex::dualFeasible()` : idem pour le dual. `ClpSimplex::dual()` : simplexe sur dual. `ClpSimplex::primal()` : simplexe sur primal.
`getVariableValue()`	`CbcModel::bestSolution()` `OsiClpSolverInterface::getColSolution()`
`writeClpexFile()`	`CoinLpIO::writeLp()`
`writeMpsFile()`	`ClpSimplex::writeMps()`
`setDoubleOption()`	`ClpSimplex::setDblParam()` `CbcModel::setAllowablePercentageGap()`
`setIntegerOption()`	`ClpSimplex::setAlgorithm()`

Remarque : certaines méthodes de `ClpCbcLinearProgram` ne sont pas précisées dans ce tableau car elles sont implémentées sans utilisation explicite de méthodes CLP/CBC/OSI. Des explications sont données par la suite.

4.4 Précisions diverses sur le développement

Pour résumer, on a la classe `ClpCbcSolver` qui représente le solver réel. Cette classe contient uniquement des méthodes natives qui sont implémentées via JNI dans `osi_clp_cbc_jni`. Enfin, `ClpCbcLinearProgram` utilise les méthodes natives de `ClpCbcSolver` pour pouvoir implémenter les méthodes héritées de `AbstractLinearProgram` et `LinearProgram`.

Ayant réussi à établir la correspondance entre les "mondes" MascOpt et CLP/CBC une bonne partie etait alors accomplie.

Mais à ce stade tout n'était pas encore fait et pas si trivial à réaliser.
Premièrement, JNI qui est syntaxiquement lourd à mettre en oeuvre (la moindre opération sur la machine virtuelle demande de nombreuses instructions), est également difficile à déboguer

(nombreux ont été les crashs violents et peu informatifs de la machine virtuelle pendant les tests).

Deuxièmement, il fallait résoudre une série de problème dont certains sont évoqués ci-dessous à titre indicatif :

- Stockage des pointeurs : le module JNI utilise des instances de `OsiClpSolverInterface`, `ClpSimplex` et `CbcModel` dont la mémoire est allouée dynamiquement. Il faut donc pour cela garder des pointeurs sur ces instances. Le problème est que le module JNI est une librairie partagée, ce qui veut dire que si l'on stocke les pointeurs dans la librairie, ils seront partagés (les instances également donc) par toutes les instances de MascOpt/machine virtuelle utilisant le module JNI. Ce qui n'est évidemment pas acceptable...
  La solution à ce problème est de stocker la valeur de chaque pointeur du côté Java. À cet effet, des variables entières privées ont été ajoutées à `ClpCbcSolver`. Cela implique la mise à jour et la consultation côté JNI de ces valeurs qu'il faut convertir en pointeurs. Des fonctions C ont été écrites pour cela. Elles sont utilisées dans toutes les fonctions d'implémentation des méthodes natives de `ClpCbcSolver`. De cette manière chaque instance de MascOpt stocke ses propres pointeurs.

- Méthode `ClpCbcLinearProgram.setProblemClass()` : expliquer comment cette méthode a été implémentée côté JNI revient à expliquer comment fonctionne le module JNI. Comme dit plus haut, le module utilise des instances de `OsiClpSolverInterface`, `ClpSimplex` et `CbcModel`. Mais en fait elles sont allouées que si on en a besoin. C'est-à-dire qu'au départ (au chargement de la librairie-module JNI) seules des instances de `OsiClpSolverInterface` et `ClpSimplex` sont allouées puisque par défaut nous sommes dans le cas d'un programme en nombres fractionnaires. Or la méthode `setProblemClass()` permet de passer en mode programme en nombres mixtes. C'est seulement lorsque cette utilisation est faite que la classe `CbcModel` est instanciée côté JNI.

- Index de variables et de contraintes : un premier problème concernant les index de variables a été rencontré et réglé sans grande difficulté. En effet, côté MascOpt les index des variables ont pour base 1 et côté CLP/CBC ils ont pour base 0. La méthode `ClpCbcLinearProgram.getClpVariableIndex()` a été rajoutée à cet effet. Un deuxième problème, cette fois pour les index de contraintes a été provoqué par le fait que CLP ne propose pas de spécifier des index pour les contraintes. La solution a été de créer des index dits "virtuels" côté JNI en utilisant le conteneur map de la STL. Ce conteneur map fait la correspondance entre les index MascOpt et les index réels côté CLP/CBC.

- Méthodes `clearModel()` et `setConstraintNumber()` de classe `ClpCbcLinearProgram` : ces deux méthodes ne sont pas précisées dans le tableau (en 4.3), car il n'a pas été utilisé d'équivalents CLP/CBC pour les implémenter. En effet, pour `clearModel()` la méthode `ClpCbcSolver.finalize()`[26] (implémentée côté JNI, pour libérer les ressources mémoires des instances allouées) est simplement appelée directement par la méthode C/C++ qui implémente (toujours côté JNI) `clearModel()`. Cela permet de réinitialiser le modèle du programme linéaire.
  Pour la méthode `setConstraintNumber()`, elle n'est pas indiquée tout simplement parce qu'il n'est pas nécessaire de l'implémenter, puisque CLP/CBC prend automatiquement en charge d'agrandir la taille de la matrice stockant les contraintes lorsque l'on en rajoute une au problème.

26 `finalize()` : en Java cette méthode (héritée de la classe Object) est normalement appelée (si possible) par le collecteur de déchets de la machine virtuelle lorsque l'objet est en fin de vie.

III DEUXIÈME PARTIE DU STAGE : mise en place de projets MAVEN

Dans la deuxième partie du stage, j'ai essentiellement travaillé avec l'outil de génie logiciel Maven. C'est un outil développé par la fondation Apache, organisation à but non lucratif – Maven est un logiciel libre. C'est un outil de gestion de projet, qui permet de gérer dans un seul environnement la compilation, la documentation, la génération de rapports et de site web. Il rend simples et rigoureux la distribution de projets et leur partage au moyen d'un système de gestion des dépendances et de plugins très efficace.
En somme, la philosophie de Maven consiste à améliorer le cycle de vie des projets en insistant sur les notions de visibilité, de réutilisabilité, de maintenabilité dans une approche simple et avec un fort niveau d'abstraction.

Pour poser la problématique de cette partie du stage, il faut revenir sur la partie précédente. J'ai développé un composant de la librairie MascOpt et il me semblait intéressant d'extraire ce composant pour pouvoir l'utiliser comme un logiciel ou une librairie à part entière. Ensuite, je voulais également m'intéresser à la portabilité du module JNI.
Mes objectifs avec Maven étaient donc de pouvoir distribuer mon projet simplement et efficacement et de pouvoir le construire de manière adéquate à la plate-forme de l'utilisateur – ce qui concerne en premier lieu le module JNI (Maven étant orienté Java essentiellement mais pas seulement, il permet de compiler du code C/C++).
Enfin, je voulais également envisager une approche pour convertir le projet MascOpt en projet Maven.

Dans cette partie je commencerai donc d'abord par décrire le fonctionnement de Maven. Ensuite, je présenterai concrètement le projet Maven que j'ai mis en place. Cela viendra par ailleurs compléter la présentation qui précède. Finalement, je présenterai brièvement l'approche qui me paraît la plus adaptée pour « maveniser » MascOpt.

1. Présentation de Maven

1.1 Créer rapidement un projet

Maven est un outil qui pour une part est orienté « ligne de commande ». On peut utiliser des plugins directement via les paramètres passés en ligne de commande.

Par exemple pour créer un projet applicatif de base on peut utiliser la commande suivante :

```
mvn archetype:create -DgroupId=org.myorganization.app -DartifactId=my-app
```

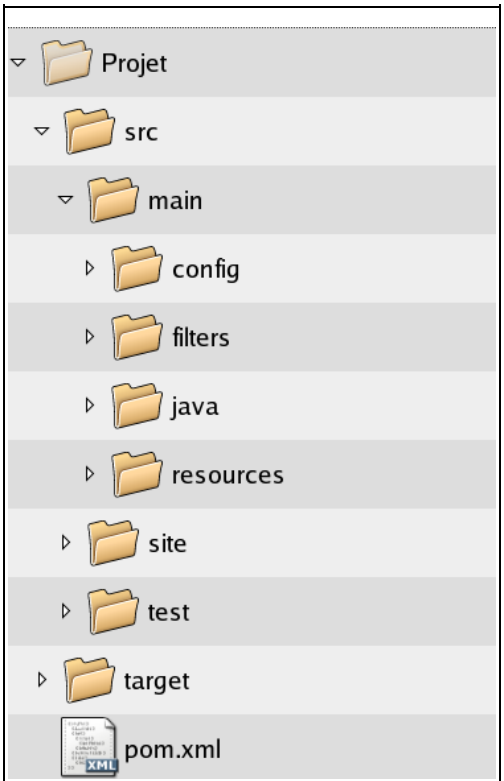
Cette commande expose divers éléments clés de Maven :

- les plugins : ils permettent d'effectuer différentes opérations dans un domaine précis, par exemple `archetype` permet de créer des projets Maven à partir de zéro, ou en partant d'un projet existant.
  Le téléchargement des plugins est géré automatiquement par Maven. Ces derniers sont téléchargés dans les dépôts de plugins distants et locaux. Il en va d'ailleurs de même pour les projets qui sont des dépendances du projet à construire (l'utilisateur doit le préciser dans sa configuration).
- Les goals : qui sont toutes les opérations « ponctuelles » que permet un plugin, ici on utilise le goal `create` de `archetype`.
- Les propriétés : en fait ce sont des propriétés Java, qui permettent de spécifier des paramètres pour les goals des plugins et qui sont aussi un moyen pour l'utilisateur de définir ses propres paramètres (voir plus bas). Ici la propriété `groupId` indique l'identifiant de l'organisation qui développe l'application, c'est en fait le paquetage Java contenant l'application. La propriété `artifactId` permet d'identifier l'application précisément.

La commande précédente permet au final de générer l'arborescence d'un projet Maven. Cette arborescence est codifiée.

1.2 Standard Directory Layout (SDL)

En effet, les projets Maven sont organisés selon une structure hiérarchique, appelée *Standard Directory Layout.* Elle permet de bien faire la différence entres les divers éléments du projet, pour une maintenance et une mise en place plus efficaces.



SDL Maven basique

À La racine du projet on trouve le fichier pom.xml, qui est central dans un projet Maven (voir plus bas). Mais à ce niveau on peut également placer les fichiers que l'on veut rendre visibles ou utilisables directement (fichiers README, licence...).

Le répertoire src contient tous les éléments qu'il faut traiter pour construire le projet. Les fichiers issus de la fabrication seront déplacés dans le répertoire target, qui conserve en grande partie l'arborescence de src. Ce dernier est aussi composé d'un répertoire par langage de programmation utilisé (ici java pour le code Java) et par le répertoire config (optionnel) pour les fichiers de configuration. Le répertoire resources contient des fichiers ressources, binaires ou textuels, que l'on peut simplement joindre aux fichiers du projet dans target ou que l'on peut filtrer (remplacement de variables par leurs valeurs à l'exécution). Le répertoire filters doit d'ailleurs contenir des fichiers de propriétés dont les valeurs sont éventuellement utilisées pour ce filtrage. Le filtrage permet d'adapter le projet à l'environnement, aux préférences de l'utilisateur...

1.3 Project Object Model (POM)

POM est le modèle central du projet Maven. C'est un fichier au format XML qui contient toutes les informations nécessaires pour que Maven puisse procéder à la construction du projet.

Il permet de préciser une série d'informations sur le projet : version, groupId, artifactId (triplet identifiant unique du projet), nom, description, développeurs, leur *mailing list*, URL du site web du projet...

Il permet de préciser les dépendances (autres projets Maven deployés dans des dépôts Maven) du projet. Les dépendances sont référencées par leur groupId et leur artifactId. Et surtout il contient les opérations à lancer pour construire le projet (lancement de goals appartenant aux plugins adéquats).

Voici la structure simplifiée d'un fichier pom.xml :

```
<project ...>
      <modelVersion>4.0.0</modelVersion> <!-- version du modèle POM utilisée -->

      <!-- infos projet -->
      <groupId>...</groupId>
      <artifactId>...</artifactId>
      <version>...</version>
      <name>...</name>
      <packaging>jar</packaging> <!-- le type d'archive que l'on veut
        généré à partir du projet, par exemple JAR. -->
<!-- C'est sous ce format que le projet est déployé dans un dépôt. -->

      <dependencies>
<!-- on précise chacune des dépendances (identifiées par artifactId...) -->
            <dependency>...</dependency>
            <dependency>...</dependency>
                  <!-- ... -->
      </dependencies>

      <build> <!-- opérations directes de construction du projet -->
            <plugins>
                  <plugin>
                        <configuration> ... </configuration>

                        <executions>
                  <!-- exécution(s) du plugin avec différente configuration -->
                              <execution>
                                    <configuration> ... </configuration>
                              </execution>
                              <!-- ... -->
                              <execution>
                                    <configuration> ... </configuration>
<!-- précise quel goal utiliser --> <goal> </goal>
```

```
                                </execution>
                            </executions>
                        </plugin>

                        <plugin>
                            <!-- un autre plugin... -->
                        </plugin>

                        <!-- ... -->
                    </plugins>

                    <resources>
  <!-- préciser les ressources du projet (répertoire, flitrage ou non...)-->
                        <resource> ... </resource>
                    </resources>

            </build>

            <!-- autre configurations... -->

</project>
```

Mais un projet Maven ne contient pas forcément qu'un seul fichier pom.xml. Précisément tous les modèles POM héritent par défaut du modèle appelé le super POM qui définit les éléments de base de tout POM.
Il y a en fait deux formes de relations entre POM : héritage et agrégation (qui sont compatibles).

- L'héritage : cela permet de créer des sous-projets au sein d'un projet Maven, augmentant ainsi la modularité de l'architecture du projet. Cela permet aux projets enfants d'hériter des dépendances du projet parent sans avoir à les redéclarer. Cela permet également d'hériter des nombreuses informations sur le projet (mais artifactId doit rester propre au sous-projet), ainsi que de la configuration de ses plugins. L'exécution de goals sur le parent est éventuellement propagée aux enfants. On utilise pour cela les balises `<parent></parent>` pour désigner le projet parent dans le POM enfant.

- Agrégat des modules par le parent : via les balises `<modules></modules>` spécifiant les sous-projets Maven (plus précisément les répertoires contenant les pom.xml correspondants) dont la construction est à poursuivre en partant du projet composant. Cela apporte à peu près les mêmes possibilités que l'héritage mais cela permet d'avoir une approche par modularité plus claire, on décharge ainsi souvent le projet parent de responsabilités attribuées logiquement aux modules composites. Il faut préciser « pom » comme valeur aux balises `<packaging></packaging>` du POM composant. Cela montre que le projet composant est une abstraction de haut niveau (du projet Maven) ayant pour but de déléguer et d'encadrer les sous-projets qui eux doivent avoir un rôle concret.

1.4 Phases de construction

Un autre élément central de Maven est son cycle de vie de construction, qui est organisé en phases. Les phases sont les étapes de ce cycle, qui lui est une séquence de différentes phases.

Concrètement on peut passer le nom d'une phase en paramètre de Maven (commande `mvn`). Maven exécute ensuite toutes les phases qui précèdent avant d'exécuter cette phase.

Voici les phases principales qui résument le cycle de vie de construction par défaut :

- `validate` : valide que le projet soit correct et que toute information nécessaire soit disponible.
- `compile` : compile le code source du projet.
- `test-compile` : compile les éventuels tests unitaires associés au code du projet.
- `test` : exécute les tests unitaires en utilisant un framework spécifique (ex : junit).
- `package` : place le code compilé dans un paquetage (par défaut il est de format JAR).
- `integration-test` : si besoin, déploie le paquetage dans un environnement pour y faire les tests d'intégration.
- `verify` : lance des tests sur la qualité du paquetage, avec critères de qualité.
- `install` : installe le paquetage dans un dépôt local pour son utilisation dans un autre projet.
- `deploy` : copie le paquetage final du projet vers un dépôt distant (rien n'empêche qu'il soit local) pour son partage (en tant que dépendance) avec d'autres projets.

Ces phases font appel à des phases intermédiaires, qui sont leurs sous-phases. Les phases sont basées sur l'utilisation de plugins et de leurs goals. Les plugins (référencés dans le POM) sont téléchargés par Maven s'ils ne sont pas déjà disponibles dans le dépôt local.

D'autres phases à part existe : notamment la phase `clean` pour nettoyer le projet après sa construction (grossièrement : suppression du répertoire target).

Voici un exemple d'exécution séquentielle de phases et de goals :

```
mvn clean package install:install-file propriétés...
```

Cette commande nettoie le projet, construit le paquetage (après avoir exécuté toutes les phases qui précédent), et ensuite exécute le goal `install-file` pour déplacer le paquetage vers le dépôt local (on peut faire ça directement via la phase `install`).

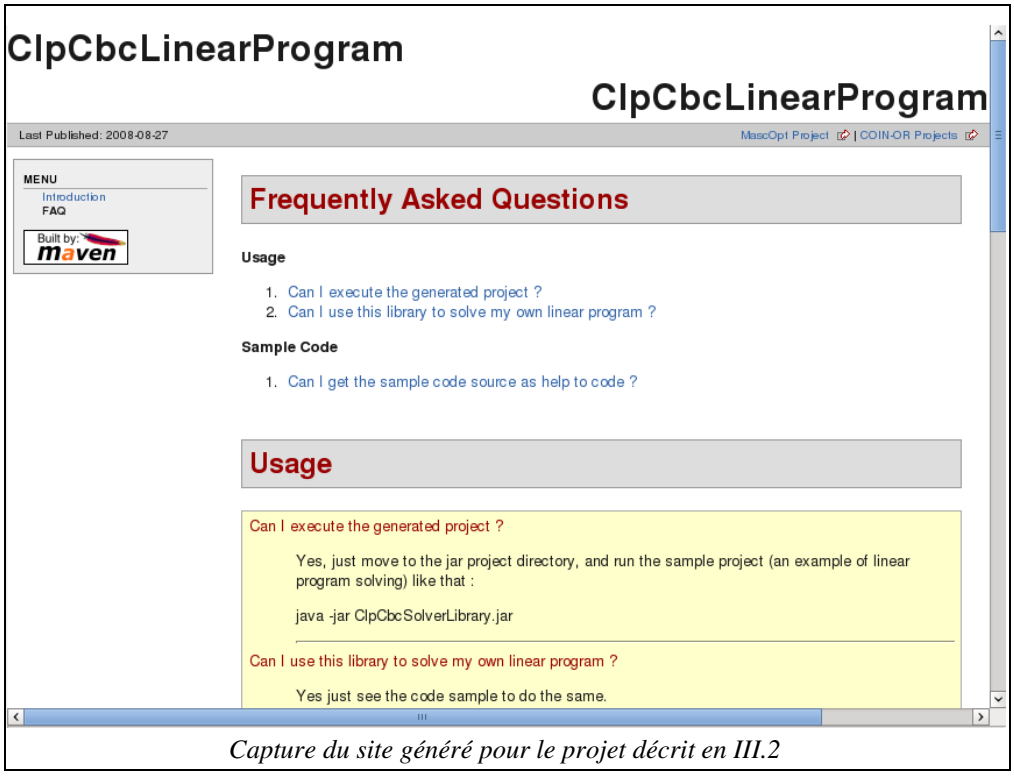
1.5 Générer un site de documentation

Maven offre la possibilité de générer un site de documentation du projet, site qui peut contenir des rapports basés sur le fichier pom.xml mais aussi tout contenu statique arbitraire.

Dans la SDL Maven d'un projet c'est le répertoire site qui est analysé pour la construction d'un site. On peut tout de même construire un site sans ajouté ce répertoire, mais tout sera généré selon la configuration par défaut. Or on peut via un fichier descripteur, faire une description du site que l'on veut construire. Ce fichier est site.xml et on doit le placer à la racine du répertoire site. En plus de cela on peut utiliser des formats spécifiques pour étendre son site, dont le format FML (FAQ Modeling Language) pour écrire une Foire Aux Questions concernant un projet Maven. On peut aussi définir le style CSS[27] du site généré comme on le ferait pour un site web classique, de plus on peut ajouter des images en ressources.

La construction du site se fait soit par la phase `site` soit par le plugin archetype. La phase `site-deploy` est ensuite utilisée pour déployer le site vers un serveur web ou autre.

Pour le fichier site.xml un format précis est défini : les balises `<bannerLeft>`, `<bannerRight>` permettent de définir des bannières à gauche et à droite (avec du texte, des images, des liens...), il y aussi les balises `<menu>`, `<links>`...



Capture du site généré pour le projet décrit en III.2

27 CSS : Cascading Style Sheet. Permet de définir le style d'un site web.

2. « Maveniser » le projet de solver CLP/CBC de MascOpt

2.1 Objectifs précis

Les objectifs de ce projet étaient de :

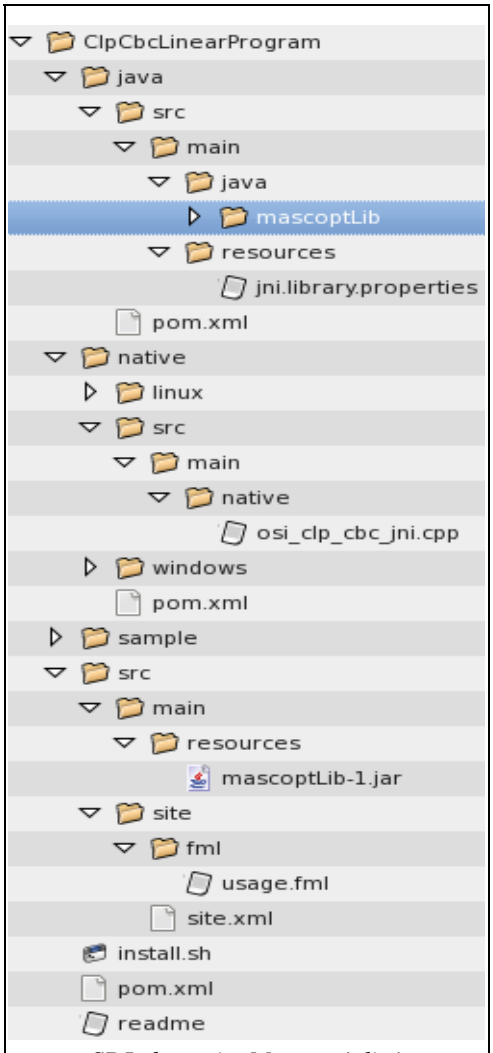
- distribuer le projet Java/JNI utilisant CLP/CBC de manière indépendante de MascOpt.
- Mettre en place une architecture de projet Maven, claire, modulaire et réutilisable.
- Générer dynamiquement le module JNI (librairie partagée) en accord avec la plate-forme exécutant le projet, de manière à favoriser la portabilité.
- Générer un site avec un minimum d'informations sur le projet Maven et son utilisation.

2.2 SDL du projet

De manière à bien comprendre comment le projet Maven a été conçu il faut partir de la SDL du projet.

Le projet contient 3 modules, qui chacun correspond à un sous-répertoire du répertoire du projet ClpCbcLinearProgram.

Le répertoire java contient le paquetage `mascoptLib.lpSolver.clpCbcImpl` qui est le code Java du projet que j'ai développé (il est situé dans le répertoire src/main).
Le sous-répertoire resources contient un fichier d'une propriété (jni.library.properties) qui est filtré par Maven à la construction du projet. Cette propriété permet de déterminer à l'exécution où est située la librairie d'implémentation du module JNI. Cela permet à la classe Java `ClpCbcSolver` de la charger. C'est l'utilisateur qui lorsqu'il lance la construction du projet donne une valeur à cette propriété, c'est-à-dire le chemin où il veut placer la librairie dynamique (module JNI).
Le fichier de propriété est ainsi ajouté, après filtrage, à l'archive JAR générée par le projet Maven, pour une consultation de sa valeur à l'exécution de l'application.



SDL du projet Maven réalisé

Le répertoire native contient le code C/C++ osi_clp_cbc_jni.cpp du module JNI. Ce module s'occupe de générer le fichier d'en-tête C à partir de la classe Java `ClpCbcSolver` qui est implémentée en C/C++ JNI. C'est un plugin Maven (nommé native) qui permet l'utilisation de l'outil javah pour générer le fichier d'en-tête. Cet en-tête est inclus à osi_clp_cbc_jni.cpp qui définit les fonctions (implémentations des méthodes natives Java) et qui est également compilé (en librairie dynamique) par le plugin native.
Le répertoire native contient deux sous-répertoires (sous-modules) qui sont windows et linux qui contiennent chacun un POM responsable de la compilation de la librairie JNI pour les systèmes Windows et Linux (c'est l'objectif de portabilité qui exige cela).

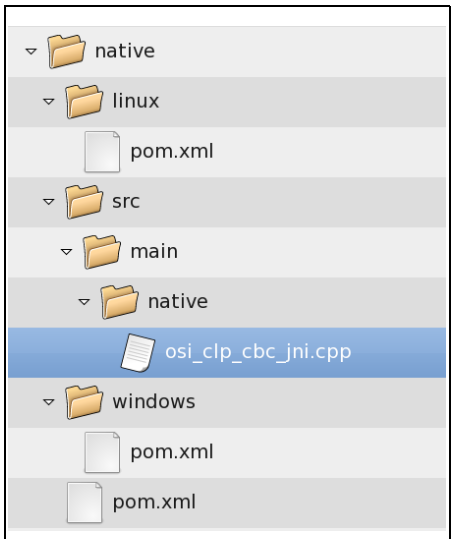
Le répertoire sample est un module contenant le code Java, exemple d'utilisation du composant-librairie de résolution de programmes linéaire (voir en annexe).

Le projet contient également en ressources un fragment de la librairie MascOpt, c'est le paquetage `MascoptLib.lpSolver` (sans les implémentations GLPK et CPLEX) qui définit l'interface générique des solvers de programmes linéaires de MascOpt. Cette ressource est utilisée comme une dépendance Maven (exportée à l'exécution vers le dépôt local Maven).

De plus, le projet contient le répertoire du site généré à la construction du projet (phase Maven `site`).

Enfin, à la racine du projet est situé le POM du projet parent-agrégat des modules précédents (java, native, sample). C'est le premier POM consulté à la construction du projet.

J'ai rajouté un script shell install.sh facilitant la construction du projet (l'utilisateur n'ayant par ce moyen pas la nécessité de taper lui-même les commandes Maven, ce qu'il peut tout de même faire sans grande difficulté).



Zoom sur le module native

2.3 Profils et compilation du module JNI

Il est intéressant de préciser le fonctionnement du module native qui génère un composant JNI adapté à l'architecture courante.
Les répertoires linux et windows contiennent leurs propres POM. Ces derniers effectuent la génération de l'en-tête C et la compilation du composant JNI.

Le POM à la racine du module native spécifie les profils de plate-forme. Les profils au sens Maven sont une manière d'effectuer des opérations sous certaines conditions. Ici on détermine quel est le système d'exploitation et on redirige Maven vers le module windows ou linux le cas échéant. Les POM de ces derniers utilisent des compilateurs C/C++ et des configurations propres à la plate-forme.
Ainsi, sous Windows on génère une librairie DLL et sous Linux une librairie d'extension so.

Voici, à titre d'illustration, des extraits du POM principal du module native et du POM windows.

Extrait de native/pom.xml

```
<!-- lance le module (POM) de l'OS exécutant la construction.. -->
<profiles>
    <profile>
        <id>win32</id>
        <activation>
            <os>
                <family>Windows</family>
            </os>
        </activation>
        <modules>
            <module>windows</module> <!-- si on est sous Windows -->
        </modules>
    </profile>

    <profile>
        <id>linux</id>
        <activation>
            <os>
                <name>Linux</name>
            </os>
        </activation>
        <modules>
            <module>linux</module> <!-- si on est sous Linux -->
        </modules>
    </profile>

</profiles>
```

Extrait de native/windows/pom.xml

```
 <plugin> <!-- utilise le plugin native de codehaus pour compiler/lier -->
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>native-maven-plugin</artifactId>
    <extensions>true</extensions>
    <configuration>
      <javaOS>windows</javaOS> <!-- on est sous Windows -->
      <compilerProvider>generic</compilerProvider>
```

```
          <compilerStartOptions> <!-- options de compilation gcc -->
            <compilerStartOption>-IC:\cygwin\include -I${clp.cbc.include.path}
-mno-cygwin -D_JNI_IMPLEMENTATION_ -O3  </compilerStartOption>
          </compilerStartOptions>

          <sources>
            <source> <!-- répertoires d'inclusion pour JNI -->
              <directory>${java.home}\..\include</directory>
            </source>
            <source>
              <directory>${java.home}\..\include\win32</directory>
            </source>
            <source> <!-- code du module JNI à compiler -->
              <directory>../src/main/native</directory>
              <fileNames>
                <fileName>osi_clp_cbc_jni.cpp</fileName>
              </fileNames>
            </source>
          </sources>

          <linkerEndOptions>
            <linkerEndOption> <!-- options d'édition de lien avec CLP/CBC -->
              -mno-cygwin -shared -Wl,--kill-at -L${clp.cbc.lib.path}
-lCoinUtils -lOsi -lCgl -lCbc -lOsiClp -lClp -lCoinUtils -lOsi -lCgl -lCbc
-lOsiClp -lClp -lstdc++
            </linkerEndOption>
          </linkerEndOptions>
        </configuration>
        <executions>
          <execution>
            <!-- génération de l'en-tête C -->
            <id>javah</id>
            <phase>generate-sources</phase> <!-- la génération se fait avant la
phase de compilation (phase compile), car on a besoin après -->
            <goals>
_             <goal>javah</goal> <!-- goal d'utilisation de javah -->
            </goals>


            <configuration>
                <!-- configuration du répertoire de sortie de l'en-tête C -->
              <outputDirectory>../src/main/native</outputDirectory>
              <outputFileName>ClpCbcSolver.h</outputFileName>
                <!-- code source de la classe Java pour générer l'en-tête -->
              <classNames>
              <className>mascoptLib.lpSolver.clpCbcImpl.ClpCbcSolver</className>
              </classNames>
            </configuration>
          </execution>
        </executions>
      </plugin>
```

Remarque : c'est le compilateur mingw (gcc pour Windows) qui a été utilisé pour la compilation. Ceci au sein de l'environnement Cygwin (qui permet d'utiliser des utilitaires Linux sous Windows). Mais on aurait très bien pu utiliser le compilateur de Visual C++ de Microsoft.

2.4 Données diverses sur l'utilisation

Le projet Maven ClpCbcLinearProgram.zip est téléchargeable à cette URL : http://dl.free.fr/jOAAhGI8U.

Son installation : elle est simple avec le script install.sh (utilisable également sous Windows via Cygwin, commande : bash install.sh).
Le script demande différents paramètres nécessaires à Maven : les chemins des répertoires où l'on veut installer l'archive JAR du projet (ClpCbCSolverLibrary.jar), la librairie partagée du module JNI, et le site web du projet. Il demande également où trouver les en-têtes C de CLP/CBC ainsi que les librairies. Il est donc nécessaire d'installer indépendamment le projet CBC (livré avec CLP) avant de pouvoir installer le projet Maven (sous Windows il faut également installer Cygwin).

Exemple de commande pour ne pas avoir à renseigner les paramètres directement pour l'installation sous Windows :

```
bash install.sh < fichier_paramètres.txt
```

Où fichier_paramètres.txt est par exemple :

```
C:\\Users\un_utilisateur\repertoire_installation_projet
C:\\Cbc-2.1.0\\lib
C:\\Cbc-2.1.0\\include\\coin
C:/Users/un_utilisateur/repertoire_installation_projet
C:\\Users\\un_utilisation\\repertoire_installation_projet
```

Son utilisation : il y a en fait deux utilisations ;

- Exécuter le sample du projet : cela permet de se faire une idée de ce qu'on peut faire avec la librairie du projet. Cela exécute le programme. On peut le faire avec la commande suivante :
  ```
  java –jar ClpCbCSolverLibrary.jar
  ```
- Développer son projet en utilisant la librairie : pour cela il faut ajouter au CLASSPATH de son projet l'archive contenant la librairie `ClpCbCSolverLibrary.jar`.
  Le programmeur pourra se baser sur l'exemple (sample) pour programmer des résolutions de programmes linéaires fractionnaires, entiers ou mixtes.
  Le code source de cet exemple est disponible en annexe.

3. Approche pour « maveniser » MascOpt

Ici je présente brièvement la stratégie envisagée, mais qui n'a pas été réalisée (faute de temps), pour convertir le projet MascOpt en projet Maven.

3.1 Ant pour construire MascOpt

Actuellement, le projet MascOpt est construit avec l'outil de génie logiciel Ant qui est développé par la fondation Apache tout comme Maven.
Ant est un outil qui permet de compiler un projet Java, mais pas seulement : il est utilisé pour générer de la documentation (javadoc), des tests unitaires (et les exécuter), créer des archives JAR et ZIP...

Ant fonctionne sur la base d'un fichier XML, build.xml, qui définit des cibles de fabrication (balise <target>) qui sont un ensemble d'opérations (*tasks*) à exécuter. Les cibles peuvent être dépendantes les unes des autres. Ant est portable puisqu'orienté Java. C'est un peu l'équivalent de l'utilitaire Make mais avec de nombreux plugins.

En somme, de nombreuses opérations que l'on fait avec Maven peuvent être effectuées par Ant. La différence réside dans l'organisation du projet qui est formalisée avec Maven mais peu ou pas avec Ant (cela ne dépend pas de lui du moins). De plus, l'abstraction de Maven est beaucoup plus forte, on aura besoin de quelques lignes de XML de haut niveau pour faire ce que l'on fait en beaucoup plus de lignes avec Ant de manière plus technique.

3.2 Passage de Ant à Maven

Puisque l'organisation hiérarchique de MascOpt est déjà de type Maven (ou presque), pour convertir le projet MascOpt en projet Maven, il suffit grossièrement de partir du fichier build.xml (Ant) pour obtenir un fichier pom.xml (Maven) ou plusieurs.

Dans un premier temps on peut simplement lancer l'exécution de build.xml via Maven. En effet, Maven permet l'utilisation de Ant !
Par la suite il faudrait convertir chaque opération ponctuelle effectuée avec Ant, en utilisation de goals (donc de plugins) Maven. Cela clarifierait probablement l'installation du projet. En augmentant ainsi sa maintenabilité.
Notamment, pour les modules JNI (celui de GLPK et CLP/CBC) on peut reprendre ce qui a été fait en partie III.2. Pour la génération d'archive JAR on peut utiliser le plugin jar de Maven, et pour la copie de fichiers on peut utiliser le plugin resources .

CONCLUSION

Ce stage a été l'occasion de mettre à profit les compétences et connaissances acquises en LP SIL. J'ai ainsi pu approfondir la pratique et la compréhension des outils Eclipse, Ant pour le génie logiciel, et pratiquer la programmation en Java en parallèle de la programmation C/C++.

Mais j'ai également acquis de nouvelles compétences. Avec l'utilisation de l'API JNI dans un cas réel. Jusque là je ne connaissais JNI que de nom et j'avais une idée vague de son utilisation. Il a également était intéressant de découvrir une partie (certes mineure mais) intéressante de MascOpt, à savoir la modélisation orientée objet de programmes linéaires, et d'avoir un exemple concret d'application des programmes linéaires dans la résolution de problèmes de routage.
De plus, cette initiation à Maven comme outil à la pointe du génie logiciel m'a semblée vraiment enrichissante.

J'ai tout de même rencontré quelques difficultés au cours du stage. De par la diversité des outils à utiliser et les nombreuses librairies à étudier en simultané. Mais aussi par le fait que je n'ai finalement pas eu le temps de mettre en place la conversion de MascOpt en projet Maven. Cela dit au départ le sujet du stage devait se borner à l'intégration d'un solver CLP/CBC à MascOpt (ce qui relativise cette difficulté).

Par ailleurs j'ai pu profiter de faire ce stage pour découvrir concrètement comment se déroule la recherche en informatique. En regardant les activités menées à MASCOTTE on est d'ailleurs assez impressionné, en tant qu'étudiant de licence informatique, par les compétences et les savoirs mis en jeu notamment au niveau des mathématiques qui sont très présentes à MASCOTTE. C'est un environnement qui favorise l'envie de progresser et de découvrir encore.

En définitive, ce stage a été une expérience déterminante dont je pourrais certainement profiter au niveau professionnel et de manière encore plus évidente au niveau personnel.

ANNEXES et BIBLIOGRAPHIE

Code du Sample du projet Maven (utilisant le composant Java/JNI développé pour MascOpt)

```
package mascoptLib.lpSolver.clpCbcSample;
import mascoptLib.lpSolver.exception.LpException;
import mascoptLib.lpSolver.interfaces.*;
import mascoptLib.lpSolver.Impl.*;
import mascoptLib.lpSolver.clpCbcImpl.ClpCbcLinearProgram;

public class Sample
{
    public static void main( String[] args ) throws LpException
    {
        System.out.println("Solve this linear program : ");
        System.out.println("\tMax 2 x1 + 4 x2 +   x3");
        System.out.println("\t2with constraints :");
        System.out.println("\t      x1 +   x2        <= 100");
        System.out.println("\t      x1 + 2 x2 + 3 x3 <= 498");
        System.out.println("\t      x1 ,   x2 ,   x3 >= 0");

        System.out.println("//------- As fractional linear program ----//");

        AbstractLinearProgram lp = new ClpCbcLinearProgram();
        LpLinearContinuousExprImpl objFunction = new
LpLinearContinuousExprImpl();
        LpLinearContinuousExprImpl contrainte1 = new
LpLinearContinuousExprImpl();
        LpLinearContinuousExprImpl contrainte2 = new
LpLinearContinuousExprImpl();
        int[] objTerms = { 2, 4, 1};
        int[] contrainte2Terms = { 1, 2, 3};
        int[] contrainte1Terms = {1,1};
        LpContinuousVariableImpl[] x = new LpContinuousVariableImpl[3];
        for (int i = 0; i < x.length; i++) {
                        x[i] = new LpContinuousVariableImpl(0,
Double.MAX_VALUE);
                        objFunction.addTerm(objTerms[i], x[i]);
                        contrainte2.addTerm(contrainte2Terms[i], x[i]);
                        if (i < 2)
                                contrainte1.addTerm(contrainte1Terms[i], x[i]);
                }
        lp.addLesserConstraint(contrainte1,100);
        lp.addLesserConstraint(contrainte2,498);
        lp.maximize(objFunction);

        lp.solve();
        System.out.println("==> SOLUTION : objective value :
"+lp.getObjectiveValue()+" variable values : x1="+lp.getVarValue(x[0])+"
x2="+lp.getVarValue(x[1])+" x3="+lp.getVarValue(x[2]));
```

```
        System.out.println("//------- As integer linear program ----//");
        lp = new ClpCbcLinearProgram();
        LpLinearIntegerExprImpl objFunction2 = new LpLinearIntegerExprImpl();
        LpLinearIntegerExprImpl cont1 = new LpLinearIntegerExprImpl();
        LpLinearIntegerExprImpl cont2 = new LpLinearIntegerExprImpl();

        LpIntegerVariableImpl[] x_int = new LpIntegerVariableImpl[3];
        for (int i = 0; i < x_int.length; i++) {
                        x_int[i] = new LpIntegerVariableImpl(0,
Integer.MAX_VALUE);
                        objFunction2.addTerm(objTerms[i], x_int[i]);
                        cont2.addTerm(contrainte2Terms[i], x_int[i]);
                        if (i < 2)
                                cont1.addTerm(contrainte1Terms[i], x_int[i]);
                }
        lp.addLesserConstraint(cont1,100);
        lp.addLesserConstraint(cont2,498);
        lp.maximize(objFunction2);
        lp.solve();
        System.out.println("==> SOLUTION : objective value :
"+lp.getObjectiveValue()+" variable values : x1="+lp.getVarValue(x_int[0])+"
x2="+lp.getVarValue(x_int[1])+" x3="+lp.getVarValue(x_int[2]));
    }
}
```

- **<u>Quelques liens de références :</u>**
  - CLP/CBC : avec guides et documentations des API.
    - http://www.coin-or.org
    - http://www.coin-or.org/projects/Clp.xml
    - http://www.coin-or.org/projects/Cbc.xml
  - Mascotte et MascOpt :
    - http://www-sop.inria.fr/mascotte/
    - http://www-sop.inria.fr/mascotte/mascopt/
  - JNI : http://java.sun.com/j2se/1.5.0/docs/guide/jni/
  - Maven :
    - http://maven.apache.org/
    - Mon projet Maven : http://dl.free.fr/jOAAhGI8U