Flexibilité et performance de codes de calcul en optimisation et simulation

Bruno Bachelet

UNIVERSITÉ BLAISE PASCAL - CLERMONT-FERRAND II

ECOLE DOCTORALE
SCIENCES POUR L'INGÉNIEUR DE CLERMONT-FERRAND

HABILITATION À DIRIGER DES RECHERCHES

présentée par

Bruno BACHELET

Spécialité: INFORMATIQUE

FLEXIBILITÉ ET PERFORMANCE DE CODES DE CALCUL EN OPTIMISATION ET SIMULATION

Soutenue publiquement le 7 décembre 2016 devant le jury

Messieurs	Farouk TOUMANI	Président
	Joel FALCOU	Rapporteur
	Jean-Pierre MULLER	Rapporteur
	Eric RAMAT	Rapporteur
	David R.C. HILL	Responsable tutélaire

REMERCIEMENTS

Pour commencer, je tiens à remercier David Hill pour son soutien dans la préparation de ce dossier, et pour la confiance qu'il m'a accordée dans les différents projets de recherche où nous avons collaboré. Tous mes remerciements à Joel Falcou, Jean-Pierre Müller et Eric Ramat pour le temps qu'ils ont consacré à rapporter mon travail. Je remercie également Farouk Toumani qui a accepté de présider ce jury.

Je souhaite joindre à mes remerciements tous les collègues du LIMOS, du département MMI de Vichy et de l'ISIMA, ainsi que les étudiants que j'ai encadré et co-encadré. Ce mémoire est le fruit d'un échange avec toutes ces personnes. Je remercie tout particulièrement Loïc Yon et Claude Mazel pour leur relecture attentive et leurs précieuses remarques qui m'ont aidé à améliorer ce manuscrit.

Merci à mes proches pour leur soutien et leur compréhension, en particulier à Matilte pour sa patience et son amour, et à notre fils Thomas dont la joie de vivre illumine chaque jour.

RÉSUMÉ

Ce manuscrit présente la synthèse de travaux de recherche en optimisation et simulation dont l'objectif est le développement d'outils logiciels flexibles et performants. Les contributions s'articulent autour de trois grands domaines:

(i) l'optimisation combinatoire, avec la résolution de problèmes de synchronisation de documents hypermédia et d'optimisation de ressources dans les nuages informatiques;

(ii) la simulation numérique, avec un schéma de couplage optimisation-simulation original pour la résolution d'un problème de conception de tournées de bus dans un réseau de transport urbain, et une méthodologie pour la minimisation de la vulnérabilité d'un écosystème prairial face aux changements climatiques;

(iii) le développement logiciel générique, avec des patrons de conception pour le développement d'algorithmes d'optimisation, et une bibliothèque pour étendre la programmation générique aux "concepts" et permettre ainsi d'améliorer la qualité des codes de calcul.

Cette démarche pluridisciplinaire s'explique par la complémentarité de ces domaines pour répondre aux problématiques étudiées. Notamment, l'optimisation combinatoire permet l'élaboration d'algorithmes très efficaces pour résoudre des problèmes de recherche opérationnelle, mais en s'appuyant sur une modélisation souvent trop simpliste du système étudié. La simulation numérique, qui permet généralement une représentation plus fine du système, peut être couplée au processus d'optimisation afin d'identifier de meilleures solutions applicables.

Les systèmes étudiés peuvent aussi être très complexes et nécessiter des algorithmes d'optimisation et des modèles de simulation très coûteux en temps de calcul. Le développement logiciel devient alors un élément crucial où les méthodes usuelles de génie logiciel ne sont pas toujours suffisantes pour répondre aux besoins de calcul intensif. Il semble alors pertinent d'étudier les possibilités offertes par la programmation générique et la métaprogrammation pour combiner, avec peu de compromis, flexibilité et performance dans les codes de calcul.

Mots-clé: optimisation combinatoire, simulation, couplage optimisation-simulation, programmation générique, métaprogrammation, concepts (C++).

ABSTRACT

This manuscript presents the synthesis of research works in optimization and simulation with the aim of developing flexible and performant software tools. The contributions are centered on three main domains:

(i) combinatorial optimization, with the resolution of problems of hypermedia documents synchronization and resources optimization in cloud computing;

(ii) numerical simulation, with an original outline of optimization-simulation coupling for solving a bus routing design problem in an urban transportation network, and a methodology to minimize the vulnerability of a grassland ecosystem facing climate changes;

(iii) generic software development, with design patterns for developing optimization algorithms, and a library to extend generic programming to "concepts" and thus to allow improving the quality of computer codes.

This multidisciplinary approach is explained by the complementarity of these domains when tackling the studied problems. Notably, combinatorial optimization allows designing very efficient algorithms to solve operational research problems, but is based on a usually too simplistic modeling of the studied system. Numerical simulation, which generally allows representing more precisely the system, can be coupled with the optimization process to identify better applicable solutions.

The studied systems can also be very complex and can need extremely time-consuming optimization algorithms and simulation models. Software development becomes thereby a crucial element where usual software engineering methods are not always enough to meet high performance computing needs. It seems thus relevant to study the possibilities offered by generic programming and metaprogramming to combine, with few compromises, flexibility and performance in computer codes.

Keywords: combinatorial optimization, simulation, optimization-simulation coupling, generic programming, metaprogramming, concepts (C++).

TABLE DES MATIÈRES

INTRODUCTION

Ce mémoire présente la synthèse de mes travaux de recherche post-doctorat qui s'articulent autour de trois grands thèmes: l'optimisation combinatoire, la simulation numérique et le développement logiciel générique. La pluridisciplinarité de ma démarche s'explique par la complémentarité de ces domaines pour répondre aux différentes problématiques que j'ai eu à traiter.

Notamment, l'optimisation combinatoire permet l'élaboration d'algorithmes très efficaces pour résoudre des problèmes de recherche opérationnelle, mais en s'appuyant sur une modélisation souvent trop simpliste du système étudié. La simulation numérique, qui permet généralement une représentation plus fine du système, peut être couplée au processus d'optimisation afin d'identifier de meilleures solutions applicables.

Les systèmes étudiés peuvent aussi être très complexes et nécessiter des algorithmes d'optimisation et des modèles de simulation très coûteux en temps de calcul. Le développement logiciel devient alors un élément crucial où l'application des méthodes usuelles de génie logiciel n'est pas toujours suffisante pour répondre aux besoins de calcul intensif. Nous étudions donc les possibilités offertes par la programmation générique et la métaprogrammation de combiner, avec peu de compromis, flexibilité et performance dans les codes de calcul.

Dans une première partie, nous présentons nos contributions à l'optimisation combinatoire pour résoudre des problèmes de synchronisation de documents hypermédia (cf. chapitre 1) et d'optimisation de ressources dans les nuages informatiques (cf. chapitre 2).

Dans une seconde partie, nous présentons nos contributions à la simulation numérique: un schéma de couplage optimisation-simulation original pour la résolution d'un problème de conception de tournées de bus dans un réseau de transport urbain (cf. chapitre 3), et une méthodologie pour la minimisation de la vulnérabilité d'un écosystème prairial face aux changements climatiques (cf. chapitre 4).

Dans une troisième partie, nous présentons nos contributions au développement logiciel générique: des patrons de conception pour le développement d'algorithmes d'optimisation (cf. chapitre 5), et une bibliothèque pour étendre la programmation générique aux "concepts" et permettre ainsi d'améliorer la qualité des codes de calcul (cf. chapitre 6). Enfin, nous décrivons notre principal projet de recherche qui vise à élaborer des schémas de conception logicielle utilisant les "concepts" pour répondre à des problématiques de développement en optimisation et simulation (cf. chapitre 7).

PARTIE I - OPTIMISATION COMBINATOIRE

CHAPITRE 1
SYNCHRONISATION DE DOCUMENTS HYPERMÉDIA

Ce chapitre introduit brièvement les problématiques liées à la synchronisation de documents hypermédia, et comment les aspects temporels de tels documents peuvent être représentés par des graphes temporels. Certains problèmes de synchronisation deviennent alors des problèmes d'optimisation dans les graphes appelés problèmes de tension de coût minimal. La structure de ces graphes temporels est particulière et très proche d'une classe de graphes dits série-parallèles. Nous avons proposé une méthode, appelée agrégation, qui permet une résolution rapide de problèmes de tension de coût minimal pour des graphes série-parallèles, et qui a été utilisée pour optimiser la qualité de rendu d'un document hypermédia.

Une adaptation de cette méthode est étudiée ici pour le problème de tension de coût binaire minimal dont l'objectif est d'optimiser la performance du rendu des documents hypermédia, cruciale dans un contexte temps réel (adaptation de la présentation à un événement) ou de mobilité (économie d'énergie). Les algorithmes, développés avec une approche générique en C++ (cf. chapitre 5), ont été intégrés au système multimédia HyperProp conçu en Java. Pour réaliser le couplage de ces codes, une bibliothèque facilitant le contrôle de composants Java directement en C++ est proposée.

1.1. Scénario temporel

Un document multimédia est un document électronique composé de différents médias comme du texte, du son, de la vidéo, des images, des animations, des applets... Lorsqu'il est possible de naviguer entre différents documents multimédia, grâce au lien hypertexte, on parle de documents hypermédia. Par exemple, les pages HTML que l'on trouve sur Internet et la plupart des CD-ROMs éducatifs constituent des documents hypermédia.

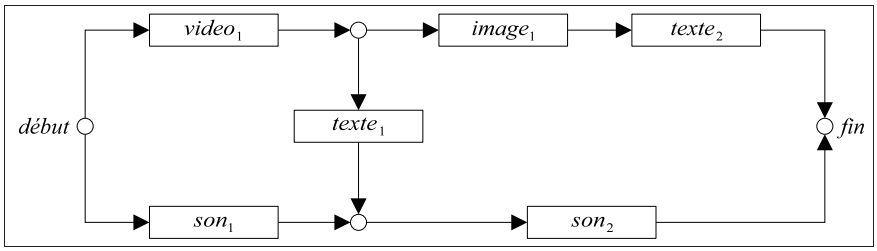

Figure 1.1: Exemple de scénario avec relations temporelles.

Les objets multimédia variés qui constituent un document hypermédia sont structurés au niveau logique, i.e. le document est décomposé en parties, chapitres, sections...; et au niveau spatial, en permettant le contrôle du placement des différents objets à l'écran ou dans une page. Dans certains types de docu-

ments comme les supports pédagogiques, il est également nécessaire de structurer les différents objets au niveau temporel. L'auteur du document décrit alors le déroulement de l'animation qu'il envisage sous la forme de relations temporelles entre les objets multimédia, notamment sous la forme de relations de précédence où les temps morts ne sont pas admis (cf. figure 1.1 où les flèches signifient que la fin de l'objet source coïncide avec le début de l'objet cible). Les objets multimédia ont généralement été produits sans envisager leur usage final, il y a donc peu de chance de pouvoir satisfaire les contraintes exprimées précédemment. Cela signifierait par exemple que la durée de $video_1$ ajoutée à celle de $texte_1$ est égale à celle de son_1. Nous appelons durée idéale la durée intrinsèque d'un objet s'il en a une (e.g. une vidéo ou un son), ou la durée que l'auteur a choisie s'il n'en a pas (e.g. un texte ou une image).

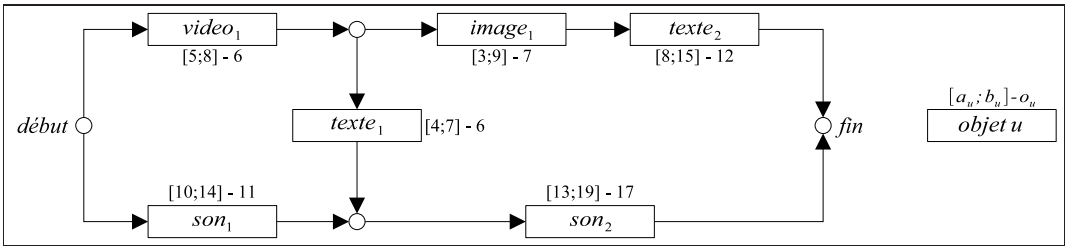

Figure 1.2: Exemple de scénario avec relations temporelles et intervalles de tolérance.

Pour que les contraintes induites par les relations temporelles puissent être satisfaites, l'auteur doit accepter une certaine flexibilité sur la durée idéale des objets multimédia. Cette flexibilité ne pose aucun souci pour une image ou un texte; en revanche, pour un extrait sonore ou vidéo, il est nécessaire d'appliquer des techniques coûteuses en temps de calcul, détériorant plus ou moins la qualité du média, pour ajuster de manière continue sa durée [Stei90]. Chaque objet est donc décrit par une durée idéale o_u et un intervalle $[a_u; b_u]$ dans lequel sa durée effective (i.e. planifiée) peut varier (cf. figure 1.2). Le problème est finalement de planifier la durée θ_u de chaque objet multimédia u de manière à satisfaire à la fois les intervalles de tolérance et les relations temporelles.

1.2. Problèmes de tension dans les graphes

Généralement, il existe plusieurs planifications possibles satisfaisant les contraintes temporelles, ce sont donc des critères de qualité sur le rendu du document hypermédia qui vont permettre de sélectionner la meilleure planification. Ce problème d'optimisation peut être interprété comme un problème de tension de coût minimal (*Minimum Cost Tension* - MCT) dans un graphe [Berg62]. En effet, comme l'illustre la figure 1.3, l'ensemble des contraintes temporelles peuvent être modélisées sous la forme d'un graphe [Bach03a]. Soit $G = (X; U)$ un graphe, avec X l'ensemble des noeuds, U l'ensemble des arcs, $m = |U|$ et $n = |X|$. Ces noeuds représentent des événements dans la présentation hypermédia (i.e. le début ou la fin de présentation d'un objet multimédia), et les arcs expriment des contraintes temporelles entre deux événements (i.e. la précédence et la durée entre deux événements).

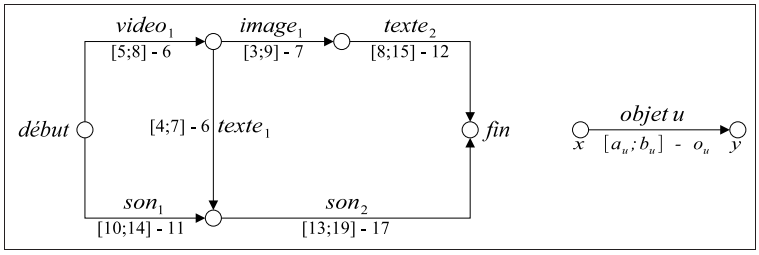

Figure 1.3: Exemple de graphe temporel.

Soit $\pi : X \longmapsto \mathbb{R}$ une fonction qui affecte un potentiel à chaque noeud du graphe. Elle représente la date planifiée pour chaque événement. Ainsi, la tension θ_u d'un arc $u = (x; y)$, qui est la différence de potentiels $\theta_u = \pi_y - \pi_x$, est la durée entre les événements x et y, et est contrainte par $\theta_u \in [a_u; b_u] \subset \mathbb{R}$. Le problème de tension de coût minimal peut finalement être modélisé de la manière suivante.

$$P_{MCT} \begin{cases} \text{minimiser } \sum_{u \in U} c_u(\theta_u) \\ \text{avec } \pi_y - \pi_x = \theta_{(x;y)},\ \forall (x; y) \in U \\ \qquad a_u \leq \theta_u \leq b_u,\ \forall u \in U \end{cases}$$

La qualité d'une planification temporelle doit être mesurée. L'approche retenue consiste à attribuer à chaque arc u une fonction de coût c_u dépendante de la valeur de la tension θ_u, telle que $c_u(o_u) = 0$, i.e. le coût de la tension θ_u est nul si celle-ci est à sa valeur idéale o_u (i.e. l'objet multimédia est planifié à sa durée idéale). Ce coût augmente à mesure que la tension θ_u s'éloigne de o_u. Plusieurs études considèrent des coûts convexes linéaires par morceaux et expriment la fonction de coût d'un arc u par des coûts unitaires c_u^1 de diminution et c_u^2 d'augmentation de la tension d'un arc u [Buch92, Kim95, Bach03a].

$$c_u(\theta_u) = \begin{cases} c_u^1(o_u - \theta_u),\ \text{si } a_u \leq \theta_u \leq o_u \\ c_u^2(\theta_u - o_u),\ \text{si } o_u < \theta_u \leq b_u \end{cases}$$

Ce problème de tension de coût convexe linéaire par morceaux minimal (*Convex Piecewise Linear Cost Tension* - CPLCT) s'exprime alors comme un programme linéaire, et plusieurs algorithmes polynomiaux ont été développés pour ce problème [Hadj97, Ahuj03]. La nature spécifique des contraintes temporelles utilisées pour la synchronisation hypermédia produit des graphes temporels très proches d'une classe de graphes appelés série-parallèles (ou SP-graphes). Nous avons proposé dans [Bach03b] une méthode dite d'agrégation pour résoudre le problème CPLCT sur des SP-graphes. Elle a montré des performances supérieures aux meilleurs algorithmes de résolution du problème CPLCT (mise à l'échelle - *dual-cost scaling* [Ahuj03]).

En pratique, les graphes sont "quasi" série-parallèles (on parle alors de QSP-graphes), ce qui signifie que leur structure générale est série-parallèle avec des perturbations plus ou moins importantes. La méthode d'agrégation a donc été combinée avec une technique dite de reconstruction pour permettre une résolution efficace du problème CPLCT sur des QSP-graphes [Bach04b]. Là aussi, la méthode de reconstruction a présenté des performances supérieures aux autres algorithmes pour des QSP-graphes où la perturbation de la structure série-parallèle ne dépasse pas 10 % du nombre total d'arcs du graphe.

1.3. Méthode d'agrégation

1.3.1. Graphes série-parallèles

Une définition commune des SP-graphes repose sur une construction récursive de ces graphes (e.g. [Epps92] ou [Vald82]) intuitive et proche de la manière de construire les relations temporelles dans un document hypermédia. Un graphe orienté G est série-parallèle s'il peut être obtenu à partir d'un graphe avec seulement deux noeuds reliés par un arc, en appliquant récursivement les deux opérations suivantes.

- La composition série, appliquée sur un arc $u = (x; y)$, crée un nouveau noeud z et remplace u par deux arcs $u_1 = (x; z)$ et $u_2 = (z; y)$ (cf. figure 1.4a). La relation série qui lie u_1 et u_2 est notée $u_1 \bigoplus u_2$.
- La composition parallèle, appliquée sur un arc $u = (x; y)$, duplique u en créant un nouvel arc $v = (x; y)$ (cf. figure 1.4b). La relation parallèle qui lie u et v est notée $u \textcircled{//} v$.

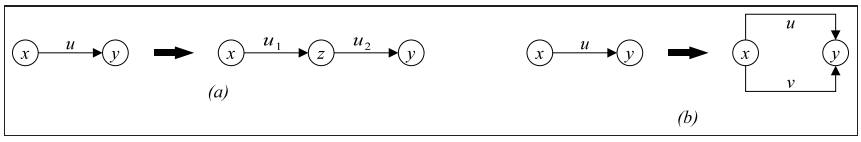

Figure 1.4: Compositions série et parallèle.

Les relations série et parallèle sont rassemblées sous le terme SP-relations. Pendant le processus de construction, une SP-relation qui lie deux arcs peut devenir une relation entre deux sous-graphes série-parallèles. Les SP-relations sont binaires, il est donc possible de représenter un SP-graphe par un arbre binaire (cf. figure 1.5) dit de décomposition [Datt99] (ou SP-arbre). [Vald82, Scho95, Epps92, Bach04b] proposent différentes manières de construire un tel arbre à partir d'un SP-graphe en un temps linéaire.

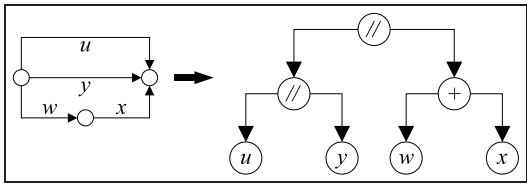

Figure 1.5: Exemple de SP-graphe et d'un SP-arbre associé.

1.3.2. Principe de la méthode

La méthode d'agrégation, qui permet de résoudre des problèmes de tension de coût minimal sur les SP-graphes, a été introduite dans [Bach03b]. L'algorithme s'appuie sur un SP-arbre T associé au SP-graphe G à optimiser et est récursif: pour une SP-relation dans T, on suppose que les tensions optimales des deux sous-graphes impliqués dans la relation sont connues; à partir de ces tensions, il est possible de construire rapidement la tension optimale de la SP-relation. Ainsi, en partant des feuilles de T, la tension optimale de chaque SP-relation est construite pour finalement atteindre la racine de l'arbre et obtenir la tension optimale du graphe entier.

Etant donnée sa définition, il est évident qu'un SP-graphe a seulement un noeud source (i.e. sans arc entrant) et un seul noeud puits (i.e. sans arc sortant). On définit la tension principale $\overline{\theta}$ d'un SP-graphe comme étant la tension entre sa source s et son puits t, i.e. $\overline{\theta} = \pi_t - \pi_s$. Pour obtenir un algorithme efficace, connaître la fonction dite de coût minimal (ou min-fonction) C_G d'un SP-graphe G est nécessaire. Cette fonction indique le coût de la tension optimale d'un graphe pour une tension principale donnée.

$$C_G(x) = \min\{\sum_{u \in U} c_u(\theta_u) \mid \theta \text{ tension}, \bar{\theta} = x\}$$

Considérons deux sous-graphes série-parallèles G_1 et G_2, et supposons que leur min-fonction C_{G_1} et C_{G_2} est connue. La min-fonction $C_{G_1 \oplus G_2}$ de la SP-relation $G_1 \oplus G_2$ est l'inf-convolution $C_{G_1} \square C_{G_2}$:

$$C_{G_1 \oplus G_2}(x) = \min_{x=x_1+x_2} C_{G_1}(x_1) + C_{G_2}(x_2) = (C_{G_1} \square C_{G_2})(x)$$

La min-fonction $C_{G_1 \oslash G_2}$ de la SP-relation $G_1 \oslash G_2$ est la somme $C_{G_1} + C_{G_2}$:

$$C_{G_1 \oslash G_2}(x) = C_{G_1}(x) + C_{G_2}(x) = (C_{G_1} + C_{G_2})(x)$$

Si la fonction c_u est convexe quel que soit l'arc u, comme dans le problème CPLCT, alors la min-fonction du graphe entier est convexe, car l'inf-convolution et la somme conservent la convexité. De ce constat, un algorithme récursif [Bach03b] pour le problème CPLCT a été proposé pour construire la min-fonction C_G d'un graphe G et déterminer sa tension optimale.

1.4. Tension de coût binaire minimal

Le nombre d'objets multimédia qui doivent être modifiés (i.e. qui n'ont pas été planifiés à leur durée idéale) est également un critère de qualité très pertinent pour la synchronisation hypermédia [Medi04]. L'altération de la durée d'un objet multimédia peut nécessiter d'importantes ressources de calcul. Dans un contexte temps réel ou de mobilité, minimiser cette opération est crucial. Nous proposons ici de développer une méthode d'agrégation pour le problème de tension de coût binaire minimal (*Binary Cost Tension* - BCT) sur des SP-graphes, où l'on cherche à minimiser uniquement le nombre d'objets qui ne sont pas planifiés à leur durée idéale. Les fonctions de coût s'expriment alors de la manière suivante.

$$c_u(\theta_u) = \begin{cases} 0, \text{ si } \theta_u = o_u \\ 1, \text{ si } \theta_u \neq o_u \end{cases}$$

Le problème se modélise alors sous la forme d'un programme linéaire en nombres entiers (PLNE), où les variables binaires y_u représentent la décision pour chaque arc u du graphe de fixer sa tension à sa valeur idéale ($y_u = 0$ signifie la tension $\theta_u = o_u$).

$$P_{BCT} \begin{cases} \text{minimiser } \sum_{u \in U} y_u \\ \text{avec } \pi_y - \pi_x = \theta_{(x;y)}, \ \forall (x;y) \in U \\ \quad -\theta_u - (o_u - a_u) y_u \leq -o_u, \ \forall u \in U \\ \quad \theta_u - (b_u - o_u) y_u \leq o_u, \ \forall u \in U \\ \quad a_u \leq \theta_u \leq b_u, \ \forall u \in U \\ \quad y_u \in \{0,1\}, \ \forall u \in U \end{cases}$$

En raison de sa nature discrète, le problème BCT est NP-difficile pour des graphes quelconques [Datt99], mais également pour la classe des SP-graphes [Ribe02]. Dans le problème BCT, les min-fonctions n'ont pas de propriétés spécifiques contrairement au problème CPLCT [Bach03b]. Ainsi, pour chaque agrégation, les min-fonctions des sous-graphes doivent être entièrement construites, car chaque partie peut être intéressante dans le processus d'agrégation.

Construire une min-fonction consiste alors à énumérer dans le pire des cas toutes les solutions possibles: toutes les combinaisons 0 ou 1 des variables y_u, ce qui représente 2^m valeurs pour un graphe avec m arcs (à noter que beaucoup de ces solutions sont irréalisables). Durant l'agrégation de deux sous-graphes, la min-fonction de chaque sous-graphe doit être traitée intégralement, donc une bonne représentation est nécessaire. En outre, les solutions irréalisables doivent être détectées pour pouvoir être éliminées. Ces considérations nous ont conduits à choisir la modélisation suivante pour représenter une min-fonction.

Nous considérons une min-fonction C_G comme un ensemble de cas, un cas e étant un ensemble d'arcs S_e du graphe $G = (X;U)$ planifiés à leur tension idéale. Soit $I_G = [MIN_G; MAX_G]$ l'intervalle dans lequel C_G est défini (i.e. l'intervalle de réalisabilité de la tension principale $\overline{\theta}$), et W_G le pire coût possible de C_G (i.e. le coût lorsqu'aucun arc n'est planifié à sa durée idéale, i.e. $W_G = |U|$). A chaque cas e est associé un intervalle de réalisabilité $i_e = [min_e; max_e]$ (de la tension principale $\overline{\theta}$) dans lequel e est possible, et son coût c_e (constant sur tout l'intervalle i_e), qui est égal à $|U| - |S_e|$. La min-fonction d'un simple arc u se définit donc de la manière suivante, avec un seul cas.

$$C_u \begin{cases} W_u = 1 \ ; \ I_u = [a_u; b_u] \\ \text{Cas 1: } S_1 = \{u\} \ ; \ c_1 = 0 \ ; \ i_1 = [o_u; o_u] \end{cases}$$

1.4.1. Agrégation parallèle

Nous expliquons ici comment construire la min-fonction C_G d'un graphe $G = G_1 \mathbin{⫽} G_2$, connaissant les min-fonctions C_1 et C_2 des SP-graphes G_1 et G_2. Mais dans un premier temps, on considère G_1 et G_2 comme de simples arcs u et v avec les min-fonctions C_u et C_v.

$$C_v \begin{cases} W_v = 1 \ ; \ I_v = [a_v; b_v] \\ \text{Cas 2: } S_2 = \{v\} \ ; \ c_2 = 0 \ ; \ i_2 = [o_v; o_v] \end{cases}$$

Construisons la min-fonction $C_{u \mathbin{⫽} v}$ de l'agrégation parallèle $u \mathbin{⫽} v$. Dans le pire cas (où aucun arc n'est planifié à sa durée idéale), le coût de la fonction est $W_{u \mathbin{⫽} v} = W_u + W_v = 2$, et l'intervalle de réalisabilité est $I_{u \mathbin{⫽} v} = I_u \cap I_v$. Ensuite, les cas particuliers suivants peuvent être identifiés.

- Cas 1': cas 1 pour u et pire cas pour v. Le coût est $c_1 + W_v$ et le cas est réalisable uniquement quand les deux cas se produisent, i.e. quand $\overline{\theta}_{u \mathbin{⫽} v} \in i_1 \cap I_v$.
- Cas 2': pire cas pour u et cas 2 pour v. Le coût est $W_u + c_2$ et le cas est réalisable quand $\overline{\theta}_{u \mathbin{⫽} v} \in I_u \cap i_2$.
- Cas 3: cas 1 pour u et cas 2 pour v. Le coût est $c_1 + c_2$ et le cas est réalisable quand $\overline{\theta}_{u \mathbin{⫽} v} \in i_1 \cap i_2$.

En résumé, la min-fonction $C_{u \mathbin{⫽} v}$ est définie comme suit.

$$C_{u \mathbin{⫽} v} \begin{cases} W_{u \mathbin{⫽} v} = W_u + W_v \ ; \ I_{u \mathbin{⫽} v} = I_u \cap I_v \\ \\ \text{Cas 1': } S_{1'} = \{u\} \quad ; \ c_{1'} = c_1 + W_v \ ; \ i_{1'} = i_1 \cap I_v \\ \text{Cas 2': } S_{2'} = \{v\} \quad ; \ c_{2'} = c_2 + W_u \ ; \ i_{2'} = i_2 \cap I_u \\ \text{Cas 3 : } S_3 = \{u; v\} \ ; \ c_3 = c_1 + c_2 \quad ; \ i_3 = i_1 \cap i_2 \end{cases}$$

L'idée est donc de tester tous les cas possibles pour déterminer leur réalisabilité et leur coût. L'algorithme 1.1 détaille la procédure pour construire la min-fonction $C_{1 \mathbin{⫽} 2}$ de la relation $G_1 \mathbin{⫽} G_2$ à partir des min-fonctions C_1 et C_2 des graphes G_1 et G_2.

Proposition 1.1: *Soient n_1 et n_2 les nombres de cas dans C_1 et C_2 respectivement. Le nombre de cas dans $C_{1 \mathbin{⫽} 2}$ ne peut pas excéder $n_1 + n_2 + n_1 n_2$, et l'agrégation parallèle $G_1 \mathbin{⫽} G_2$ nécessite $O(n_1 n_2)$ opérations.*

Preuve: L'algorithme 1.1 considère $n_1 + n_2 + n_1 n_2$ cas (les cas simples, puis les combinaisons de cas). □

Algorithme 1.1: **agrégationParallèle(min-fonction C_1,min-fonction C_2)**.

si $I_1 \cap I_2 \neq \varnothing$ alors
 $W_{1 \mathbin{@} 2} \leftarrow W_1 + W_2$; $I_{1 \mathbin{@} 2} \leftarrow I_1 \cap I_2$;

 pour tous les cas e dans C_1 tels que $i_e \cap I_2 \neq \varnothing$ faire
  créer nouveau cas e' pour $C_{1 \mathbin{@} 2}$;
  $S_{e'} \leftarrow S_e$; $c_{e'} \leftarrow c_e + W_2$; $i_{e'} \leftarrow i_e \cap I_2$;
 fin pour;

 pour tous les cas e dans C_2 tels que $i_e \cap I_1 \neq \varnothing$ faire
  créer nouveau cas e' pour $C_{1 \mathbin{@} 2}$;
  $S_{e'} \leftarrow S_e$; $c_{e'} \leftarrow c_e + W_1$; $i_{e'} \leftarrow i_e \cap I_1$;
 fin pour;

 pour toutes les paires de cas e dans C_1 et f dans C_2 telles que $i_e \cap i_f \neq \varnothing$ faire
  créer nouveau cas e' pour $C_{1 \mathbin{@} 2}$;
  $S_{e'} \leftarrow S_e \cup S_f$; $c_{e'} \leftarrow c_e + c_f$; $i_{e'} \leftarrow i_e \cap i_f$;
 fin pour;
fin si;

Il est à noter qu'en pratique, de nombreuses combinaisons de cas ne seront pas possibles (i.e. leur intervalle de réalisabilité sera vide). Notamment, il semble raisonnable de penser que le cas 3 n'arrivera que rarement.

1.4.2. Redondance et recouvrement

Le nombre de cas pourrait être surestimé: il est possible, dans une même min-fonction, d'avoir plusieurs cas avec le même intervalle. Afin d'éviter cette redondance (et ensuite des combinaisons de cas inutiles), l'un des cas doit être retiré: celui avec le pire coût (s'ils sont égaux, peu importe celui qui est éliminé). Si l'on suppose que $o_u = o_v$ dans l'exemple précédent, les cas 1' et 2' deviennent redondants avec le cas 3. Nous proposons de détecter la redondance à chaque fois qu'un cas e est ajouté à la min-fonction d'une SP-relation. Le cas e est alors confronté à chaque cas f déjà présent dans la min-fonction. Il peut y avoir trois situations possibles.

- Si $c_e \geq c_f$ et $i_e \subseteq i_f$, les cas e et f sont redondants, e n'est donc pas ajouté.
- Si $c_e \leq c_f$ et $i_e \supseteq i_f$, les cas e et f sont redondants, f est alors remplacé par e.
- Sinon, il peut y avoir un recouvrement des cas, mais e est ajouté.

A noter qu'on ne tente pas de construire un ensemble minimal de cas pour une SP-relation, i.e. un ensemble sans recouvrement des cas. Tout d'abord, cet ensemble n'est pas le plus petit pouvant représenter une min-fonction. Ensuite, maintenir cet ensemble semble compliqué et potentiellement coûteux en temps de calcul. Il faudra donc considérer le recouvrement de cas dans les algorithmes (e.g. des cas e et f où $i_e \cap i_f \neq \varnothing$, $i_e \nsubseteq i_f$ et $i_f \nsubseteq i_e$).

1.4.3. Agrégation série

Nous expliquons ici comment construire la min-fonction C_G d'un graphe $G = G_1 \bigoplus G_2$, connaissant les min-fonctions C_1 et C_2 des SP-graphes G_1 et G_2. Considérons dans un premier temps la min-fonction de l'agrégation $u \bigoplus v$ de deux arcs.

$$C_{u \bigoplus v} \begin{cases} W_{u \bigoplus v} = W_u + W_v \ ; \ I_{u \bigoplus v} = [a_u + a_v; b_u + b_v] \\ \text{Cas 1': } \{u\} \quad ; \ c_{1'} = c_1 + W_v \ ; \ i_{1'} = [o_u + a_v; o_u + b_v] \\ \text{Cas 2': } \{v\} \quad ; \ c_{2'} = c_2 + W_u \ ; \ i_{2'} = [a_u + o_v; b_u + o_v] \\ \text{Cas 3 : } \{u; v\} \ ; \ c_3 = c_1 + c_2 \quad ; \ i_3 = [o_u + o_v; o_u + o_v] \end{cases}$$

L'idée est, là aussi, d'énumérer tous les cas (ils sont toujours réalisables) et de déterminer leur coût. L'algorithme 1.2 détaille la procédure pour construire la min-fonction $C_{1 \bigoplus 2}$ de la relation $G = G_2 \bigoplus G_2$ à partir des min-fonctions C_1 et C_2 des graphes G_1 et G_2.

Algorithme 1.2: **agrégationSérie(min-fonction** C_1**,min-fonction** C_2**).**

$W_{1 \bigoplus 2} \leftarrow W_1 + W_2$; $I_{1 \bigoplus 2} \leftarrow [MIN_1 + MIN_2; MAX_1 + MAX_2]$;

pour tous les cas e dans C_1 faire
 créer un nouveau cas e' pour $C_{1 \bigoplus 2}$;
 $S_{e'} \leftarrow S_e$; $c_{e'} \leftarrow c_e + W_2$; $i_{e'} \leftarrow [min_e + MIN_2; max_e + MAX_2]$;
fin pour;

pour tous les cas e dans C_2 faire
 créer un nouveau cas e' pour $C_{1 \bigoplus 2}$;
 $S_{e'} \leftarrow S_e$; $c_{e'} \leftarrow c_e + W_1$; $i_{e'} \leftarrow [min_e + MIN_1; max_e + MAX_1]$;
fin pour;

pour toutes les paires de cas e dans C_1 et f dans C_2 faire
 créer un nouveau cas e' pour $C_{1 \bigoplus 2}$;
 $S_{e'} \leftarrow S_e \cup S_f$; $c_{e'} \leftarrow c_e + c_f$; $i_{e'} \leftarrow [min_e + min_f; max_e + max_f]$;
fin pour;

Proposition 1.2: *Soient n_1 et n_2 les nombres de cas dans C_1 et C_2 respectivement. Le nombre de cas dans $C_{1 \bigoplus 2}$ ne peut pas excéder $n_1 + n_2 + n_1 n_2$, et l'agrégation série $G_1 \bigoplus G_2$ nécessite $O(n_1 n_2)$ opérations.*

Preuve: L'algorithme 1.2 considère $n_1 + n_2 + n_1 n_2$ cas (les cas simples, puis les combinaisons de cas). □

Contrairement à l'agrégation parallèle, tous les cas sont réalisables. Cependant, il est toujours nécessaire de supprimer les cas redondants. A titre indicatif, pour une instance de SP-graphe avec 20 noeuds et 40 arcs, nous avons observé plus de 2 millions de cas dans la min-fonction finale, si les cas redondants sont conservés. Avec une constante élimination de ces cas au cours du processus d'agrégation, on observe moins de 100 cas dans la min-fonction finale (cf. tableau 1.1 dans [Bach09]).

1.4.4. Complexité

Proposition 1.3: *Pour résoudre le problème BCT, l'agrégation génère une min-fonction avec au pire* $2^m - 1$ *cas, et nécessite* $O(2^m)$ *opérations.*

Preuve: La min-fonction C_k d'un SP-graphe G_k avec k arcs possède au pire $2^k - 1$ cas. Pour $k = 1$, c'est évident. Supposons maintenant que l'affirmation est vraie pour tout SP-graphe avec au pire k arcs, et considérons un SP-graphe G_{k+1} avec $k+1$ arcs. Ce graphe est la composition série ou parallèle de deux SP-graphes G_p et G_q, avec p et q arcs respectivement, tels que $p + q = k + 1$. D'après les propositions 1.1 et 1.2, la min-fonction de G_{k+1} possède $(2^p - 1) + (2^q - 1) + (2^p - 1)(2^q - 1) = 2^{k+1} - 1$ cas.

D'après les propositions 1.1 et 1.2, chaque composition i nécessite $O(2^{k_i})$ opérations, où k_i est le nombre d'arcs dans i. Comme il y a $m - 1$ SP-relations dans le SP-graphe [Bach03b], l'agrégation complète nécessite $O(\sum_{i=1}^{m-1} 2^{k_i})$ opérations (en supposant que les compositions sont numérotées dans l'ordre avec lequel elles ont été effectuées pendant l'agrégation). Il est connu que $\sum_{i=1}^{m} 2^i < 2^{m+1}$ et il peut être vérifié que $\sum_{i=1}^{m-1} 2^{k_i} \leq \sum_{i=1}^{m} 2^i$, donc l'agrégation complète nécessite $O(2^m)$ opérations. □

1.4.5. Résultats numériques

La complexité établie précédemment considère les pires situations, nous avons donc étudié le comportement de l'algorithme en pratique. A notre connaissance, la seule autre méthode de résolution exacte du problème BCT consiste à résoudre le programme linéaire en nombres entiers P_{BCT}. Le tableau 1.1 présente une comparaison des temps de calcul (en secondes) de la technique d'agrégation et de la résolution du PLNE (utilisant le solveur CPLEX [1]), ainsi que le nombre de cas générés par la méthode d'agrégation. Chaque résultat est la moyenne de 30 résolutions de problèmes générés aléatoirement [2]. La valeur indiquée entre parenthèses est l'écart-type. La dernière colonne précise le nombre de fois où l'agrégation a été plus rapide que la résolution du PLNE. Ces résultats sont extraits de [Bach09].

Noeuds	Arcs	CPLEX		Agrégation				Agrégation
n	m	Temps		Temps		Cas		*vs.* CPLEX
10	20	0.01	(0.01)	0.01	(0.01)	32.3	(35.9)	(0/30)
20	40	0.03	(0.03)	0.01	(0.01)	86.4	(68.2)	(24/30)
30	60	0.09	(0.14)	0.01	(0.02)	143	(157)	(26/30)
40	80	0.43	(0.92)	0.03	(0.03)	231	(264)	(25/30)
50	100	4.2	(14.9)	0.08	(0.13)	342	(365)	(27/30)
60	120	9.1	(18.5)	0.19	(0.35)	498	(577)	(27/30)
70	140	276	(1049)	0.29	(0.58)	445	(574)	(27/30)
80	160	982	(5027)	0.22	(0.3)	464	(467)	(28/30)
90	180	3172	(6153)	0.9	(1.4)	844	(730)	(28/30)
100	200	8490	(19298)	1.1	(3.2)	783	(904)	(28/30)

Tableau 1.1: Résultats numériques de la méthode d'agrégation pour le problème BCT.

[1] `http://www.ibm.com/software/integration/optimization/cplex-optimizer`
[2] Générateur d'instances: `http://www.nawouak.net/?doc=bpp_library+ch=build_graph`

Il apparaît que l'agrégation est plus rapide que la résolution du PLNE sur la plupart des instances (cf. dernière colonne). La programmation linéaire est parfois plus rapide car l'agrégation n'est pas très efficace avec une chaîne de compositions série: dans une relation parallèle, il y a des cas qui peuvent être détectés irréalisables instantanément, alors que dans une relation série, tous les cas sont potentiellement réalisables, et la plupart sont éliminés plus tard dans le processus d'agrégation.

1.5. Intégration dans un lecteur hypermédia

Nos travaux [Bach03a, Bach03b, Bach04b, Bach09] sur la synchronisation hypermédia nous ont amenés à concevoir une bibliothèque logicielle C++ [3] reposant sur la programmation objet et la programmation générique pour résoudre des problèmes d'optimisation dans les graphes. L'approche utilisée (cf. chapitre 5) permet d'obtenir des algorithmes génériques (i.e. interchangeables et extensibles) efficaces, mais aussi une plateforme homogène pour des tests numériques comparatifs.

Nous avons collaboré avec le laboratoire Telemídia de l'Université Catholique de Rio de Janeiro sur la modélisation de problèmes de synchronisation hypermédia et sur l'intégration des algorithmes d'optimisation développés dans le système de diffusion multimédia HyperProp [Bach07a]. Ce dernier, écrit en Java, est un outil avant tout expérimental pour la diffusion de documents hypermédia synchronisés, où il est important de pouvoir tester différentes approches de synchronisation. Il est donc inenvisageable de réécrire tous les algorithmes de manière dédiée en Java avec pour conséquence de perdre les aspects génériques et donc la flexibilité de nos algorithmes.

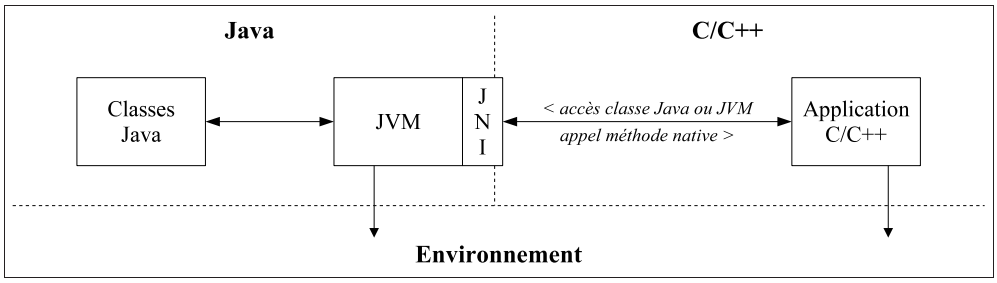

Figure 1.6: Interaction entre C++ et Java avec JNI.

La solution retenue consiste à faire coopérer des codes Java et C++ grâce à l'interface JNI (*Java Native Interface* [Lian99]) intégrée au langage Java. Cette interface permet à la machine virtuelle (*Java Virtual Machine* - JVM) qui interprète un code Java de lancer du code natif (i.e. compilé dans le langage de la machine physique). Elle permet aussi à un code natif (e.g. écrit originellement en C++) d'interagir avec la machine virtuelle et de manipuler ainsi des composants Java (cf. figure 1.6). Néanmoins, le mécanisme d'échange est fastidieux et de nombreuses instructions sont nécessaires pour des actions

[3] B++ Library: `http://www.nawouak.net/?doc=bpp_library`

simples. Nous avons donc développé la bibliothèque Jirk++ [4] qui encapsule le mécanisme d'échange et permet de manipuler des objets Java directement sous la forme d'objets C++.

1.5.1. *Java Native Interface*

Prenons un exemple simple d'échange où l'on considère une classe `Produit` avec une méthode `facturer` native (e.g. dont le code est écrit en C++) qui reçoit en argument un objet de la classe `Promotion`. Côté Java, cette méthode sera déclarée à l'aide du mot-clé `native` et pourra ensuite être utilisée comme n'importe quelle autre méthode, comme le montre le code suivant.

```
class Produit {
 [...]

 public double getPrix() { [...] }

 public native double facturer(int quantite,Promotion promo);

 public static void main(String[] args) {
  Produit produit = new Produit(100);
  double  montant = produit.facturer(250,new Promotion(100,0.1));
 }

 static { System.loadLibrary("exemple"); }
}

class Promotion {
 [...]

 public double calculer(int quantite,double prix) { [...] }
}
```

Côté C++, le code natif est défini dans une fonction avec un nom précisément formaté pour identifier la méthode Java associée (cf. code ci-dessous). Cette fonction reçoit entre autres arguments l'identifiant de l'objet Java sur lequel la méthode est exécutée (i.e. l'équivalent de `this`) dans le cas d'une méthode d'instance (i.e. non statique), ou l'identifiant de la classe Java dans le cas d'une méthode de classe (i.e. statique). Une fois le code C++ compilé, il est embarqué dans une bibliothèque dynamique (e.g. `exemple.dll`) qui doit être importée au premier chargement de la classe (cf. section `static` dans le code Java).

```
JNIEXPORT jdouble JNICALL Java_Produit_facturer(JNIEnv * env,jobject thiz,
                                                jint quantite,jobject promo) {
 jclass jc_facture = env->GetObjectClass(thiz);
 jclass jc_promo   = env->GetObjectClass(promo);

 jmethodID jm_getPrix  = env->GetMethodID(jc_facture,"getPrix","()D");
 jmethodID jm_calculer = env->GetMethodID(jc_promo,"calculer","(ID)D");

 jdouble ristourne = 0, prix = 0;

 if (jm_getPrix) prix=env->CallDoubleMethod(thiz,jm_getPrix);
 if (jm_calculer) ristourne=env->CallDoubleMethod(promo,jm_calculer,quantite,prix);

 return quantite*prix-ristourne;
}
```

[4] `http://www.nawouak.net/?doc=bpp_library+ch=jirk`

Une méthode native peut recevoir des arguments. Ceux-ci peuvent être de types primitifs (i.e. `int`, `double`, `boolean`...) côté Java et seront automatiquement convertis en arguments de types primitifs côté C++. Les arguments peuvent aussi être des références sur des objets côté Java et deviendront des identifiants côté C++ (cf. type `jobject`). Il faut alors passer par l'interface JNI pour accéder aux objets référencés et à leurs membres. Dans notre exemple, la méthode native a besoin d'accéder à la méthode `getPrix` de l'objet courant, et à la méthode `calculer` de l'objet `Promotion` reçu en argument. Pour effectuer cette simple opération, il faut d'abord obtenir l'identifiant de la classe concernée (e.g. `jc_promo`), ensuite celui de la méthode (e.g. `jm_calculer`, en indiquant sous forme de chaîne de caractères sa signature codifiée: `"(ID)D"`), pour enfin pouvoir exécuter la méthode.

1.5.2. Modélisation des composants Java en C++

Dans le cas d'une interaction évoluée entre Java et C++, comme c'est le cas avec HyperProp, le mécanisme élémentaire d'échange n'est pas satisfaisant. Au lieu de manipuler l'identifiant d'un objet Java en C++, nous proposons de manipuler un objet "miroir" qui encapsule cet identifiant, et surtout le mécanisme qui permet d'accéder aux membres de l'objet Java.

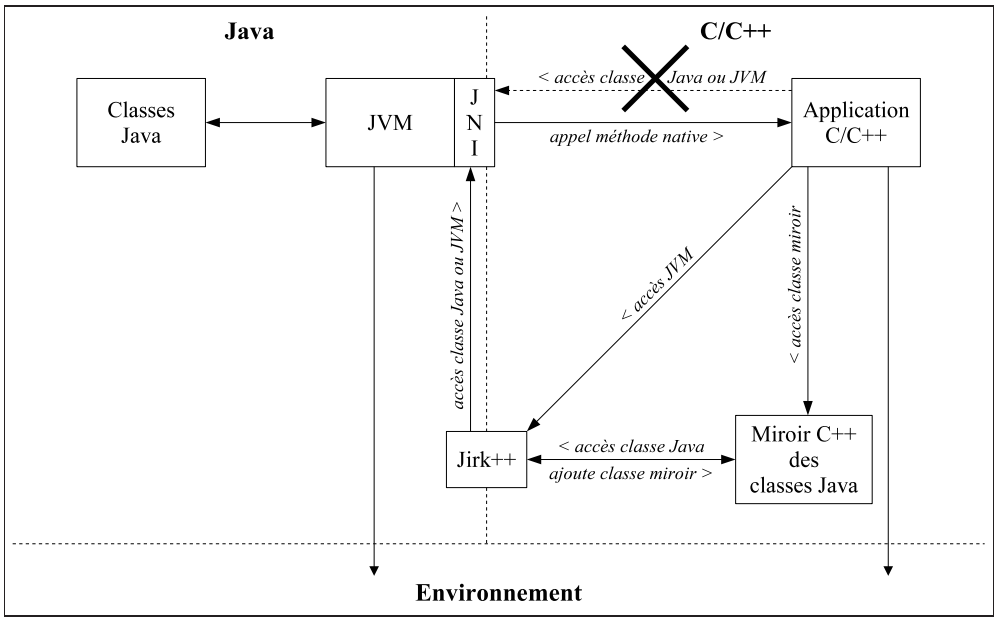

Figure 1.7: Interaction entre C++ et Java avec la bibliothèque Jirk++.

L'idée est que chaque classe Java soit représentée par une classe équivalente côté C++ (cf. figure 1.7): chaque membre public de la classe Java sera modélisé par une méthode dans sa classe miroir côté C++, cette méthode encapsulant le mécanisme d'échange avec JNI pour accéder à ce membre. L'exemple précédent peut alors s'écrire de la manière suivante [5].

[5] Les classes et les méthodes miroirs sont préfixées respectivement par `ja` et `j_` pour éviter tout conflit côté C++.

```
java_method(jdouble,Produit,facturer)(JNIEnv * env,jobject thiz,
                                      jint quantite,jobject promo) {
 getVirtualMachine(env);

 jaProduit   j_thiz  = thiz;
 jaPromotion j_promo = promo;

 jdouble prix      = j_thiz.j_getPrix();
 jdouble ristourne = j_promo.j_calculer(quantite,prix);

 return quantite*prix-ristourne;
}
```

Comme Java permet l'introspection (i.e. la capacité à examiner notamment la structure d'une classe), il est possible de générer automatiquement la classe C++ équivalente à toute classe Java (cf. figure 1.8): tous les membres de la classe Java (e.g. `Produit`) sont parcourus pour générer les méthodes d'accès associées dans la classe miroir (e.g. `jaProduit`). La hiérarchie des classes est également respectée: les relations d'héritage sont transposées sur les classes miroirs. Quant aux interfaces, elles sont modélisées sous la forme de classes abstraites côté C++.

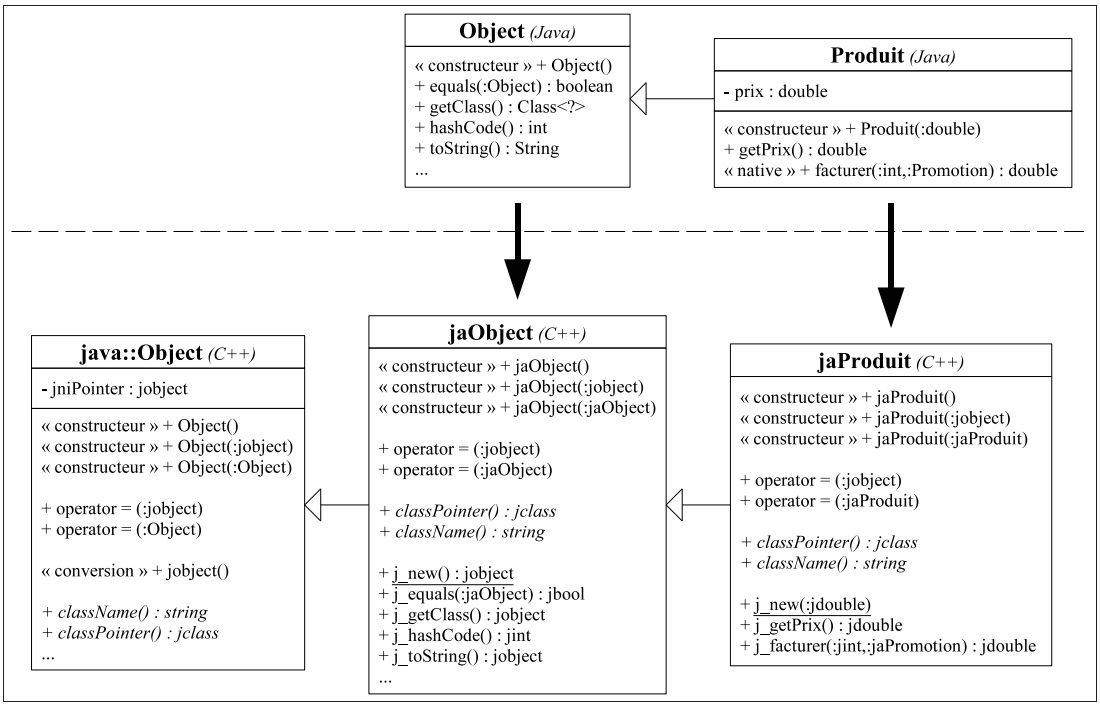

Figure 1.8: Exemple de classes miroirs.

Chaque classe miroir hérite de la super-classe `java::Object` qui implémente la base de l'encapsulation des échanges avec JNI. La sémantique des références propre à Java est transposée côté C++, principalement via la surcharge d'opérateurs. Par exemple, l'affectation entre deux objets miroir doit signifier une copie de référence, alors que la sémantique originale du C++ suppose une copie d'état (i.e. des attributs). D'autres éléments du langage Java nécessitent une modélisation particulière en C++ comme les références de tableau et les chaînes de caractères [4].

1.6. Conclusion

La méthode d'agrégation présentée ici est une première alternative à la programmation linéaire en nombres entiers (PLNE) pour résoudre le problème BCT sur des SP-graphes. L'efficacité pratique de la méthode est largement au dessus de celle de la PLNE (cf. tableau 1.1). Il serait utile de résoudre le problème BCT sur des graphes non-spécifiques: comme nous l'avons montré dans [Bach09], l'agrégation permet de résoudre efficacement le problème sur un cycle élémentaire, ce qui est utile pour générer des coupes facilitant la résolution du programme linéaire du problème BCT [Bach04a]. L'agrégation peut aussi servir à résoudre de manière exacte et relativement rapidement le problème sur les sous-graphes de n'importe quel graphe, ce qui permettrait d'obtenir des bornes pour de nouvelles coupes.

La méthode d'agrégation produit plus qu'une tension optimale. Elle fournit des informations agrégées sur le graphe (sa min-fonction) qui permettent d'adapter de manière optimale la tension d'un SP-graphe à n'importe quelle tension principale donnée en un temps polynômial [Bach09]. Pour le problème CPLCT, la technique de mise à conformité (ou *out-of-kilter* [Pla71], méthode courante pour résoudre des problèmes de tension optimale) peut exploiter efficacement des min-fonctions de sous-graphes fournies par la méthode d'agrégation. Cette idée d'exploiter les informations des min-fonctions de sous-graphes est à la base de la technique de reconstruction que nous avons présentée dans [Bach04b] et qui permet de résoudre le problème CPLCT sur des quasi SP-graphes (QSP-graphes). Il semble pertinent d'envisager cette approche pour le problème BCT.

CHAPITRE 2
OPTIMISATION DE RESSOURCES DANS LES NUAGES

Les nuages informatiques permettent d'accéder à des ressources informatiques sans avoir à investir dans l'installation et la maintenance de matériel. Le modèle de tarification des fournisseurs consiste à facturer en fonction de ce qui est consommé (transfert, stockage, calcul...). Plusieurs problèmes d'optimisation se posent alors à l'utilisateur d'un nuage, d'abord en termes de dimensionnement des ressources à demander, et ensuite en termes d'utilisation de ces ressources. Ces problèmes sont de nature bicritère, dans la mesure où l'on cherche à optimiser deux objectifs antagonistes: minimiser le coût d'exploitation dans le nuage et maximiser la qualité des services déployés. Dans ce chapitre, nous considérons d'abord une optimisation de ces critères séparément, avant d'envisager une optimisation bicritère dont nous présentons une étude préliminaire.

Sur la base d'un modèle de tarification suffisamment général pour représenter les pratiques actuelles des fournisseurs de nuage, deux problèmes d'optimisation sont étudiés. Le premier problème vise à améliorer les temps de réponse de requêtes sur une base de données stockée dans un nuage, à l'aide d'un mécanisme de cache reposant sur la matérialisation de vues (i.e. des extractions de la base), pour un ensemble de ressources de calcul fixé. Le second problème s'intéresse à la scalabilité, i.e. la capacité à adapter les ressources de calcul, afin de répondre notamment à un pic d'activité prévisible sur une application déployée dans un nuage, l'objectif à terme étant d'intégrer l'aspect scalabilité au problème de matérialisation de vues.

2.1. Modèle de tarification

Bien que les pratiques de tarification diffèrent d'un fournisseur de nuage à l'autre, nous avons défini dans [Perr13], en complément de la proposition de [Nguy12], un modèle général sur lequel s'appuient les problèmes d'optimisation présentés dans la suite du chapitre. Nous considérons ici une application déployée dans un nuage à laquelle des utilisateurs sollicitent des services. Comme illustration, nous choisissons la gestion de données dans un nuage, où des utilisateurs soumettent des requêtes $Q = \{Q_i\}_{i=1..n_Q}$ (que nous appelons la charge) à une base de données D sur une période de temps donnée. Notons f_i le nombre de fois où la requête Q_i est demandée.

Le coût total C pour déployer une application dans un nuage se décompose en trois parties, selon le type des ressources consommées: le coût C_t de transfert de données vers et depuis le nuage (les transferts internes ne sont pas facturés), le coût C_s de stockage de données dans le nuage, et le coût C_c de calcul lié aux traitements effectués dans le nuage par l'application.

$$C = C_t + C_s + C_c$$

Pour chaque type de ressources, le coût est généralement proportionnel à la consommation, mais les fournisseurs proposent parfois la gratuité en dessous d'un certain seuil, ou bien une réduction du coût unitaire au delà de certaines quantités. Cette forme de tarification peut être modélisée par une fonction linéaire par morceaux: c_t^- pour les transferts montants, c_t^+ pour les transferts descendants, et c_s pour le stockage. Une fonction linéaire par morceaux c se décompose en segments (cf. figure 2.1).

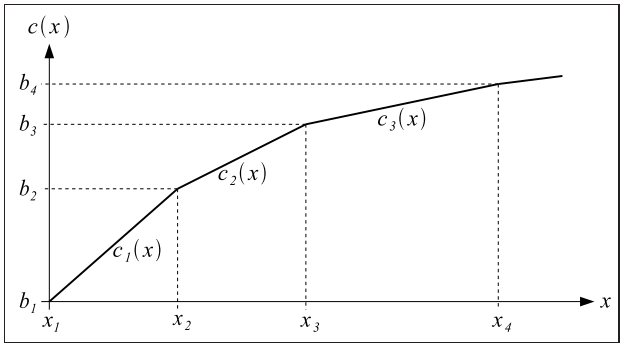

Figure 2.1: Fonction de coût linéaire par morceaux.

Chaque segment e représente c pour un intervalle des valeurs d'entrée $[x_e; x_{e+1}[$ et est caractérisé par une pente a_e et un coût initial $b_e = c(x_e)$. La fonction linéaire par morceaux c s'exprime donc comme suit.

$$c(x) = a_e(x - x_e) + b_e, \ e \text{ tel que } x_e \leq x < x_{e+1} \tag{2.1}$$

Le coût de transfert de données C_t dépend de la taille des données envoyées au nuage, i.e. des requêtes Q et de la base de données D (incluant les données initiales et les données éventuellement insérées ensuite), et de la taille des données émises par le nuage, i.e. les réponses $A = \{A_i\}_{i=1..n_Q}$ aux requêtes. Notons $s(X)$ la taille d'un ensemble de données X. Le fournisseur applique généralement un tarif différent aux transferts montants (fonction de coût c_t^-) et descendants (fonction de coût c_t^+).

$$C_t(D, Q, A) = c_t^-(s(D) + \sum_{i=1}^{n_Q} f_i \, s(Q_i)) + c_t^+(\sum_{i=1}^{n_Q} s(A_i)) \tag{2.2}$$

Plusieurs types de noeuds de calcul (appelés instances de calcul) sont proposés par le fournisseur de nuage. Chaque type d'instances correspond à une configuration matérielle et logicielle offrant des performances différentes. Le coût de calcul C_c dépend de la charge Q et des instances de calcul demandées $I = \{I_j\}_{j=1..n_I}$. Le coût de calcul lié à une instance est proportionnel à sa durée d'utilisation et dépend de ses performances. Notons $c_c(I_j)$ le coût d'utilisation par unité de temps de l'instance I_j, et $t(Q_i, I_j)$ le temps de calcul nécessaire à la requête Q_i sur l'instance I_j. Le coût de calcul de l'ensemble des requêtes Q s'exprime comme suit.

$$C_c(Q, I) = \sum_{i=1}^{n_Q} \sum_{j=1}^{n_I} f_i \, t(Q_i, I_j) \, c_c(I_j) \tag{2.3}$$

Le coût de stockage C_s dépend de la taille et du temps de stockage de la base de données D, ainsi que de la configuration des instances I. Certains fournisseurs proposent un stockage global, i.e. où toutes les instances accèdent au même espace, alors que d'autres proposent un stockage dédié à chaque instance, ce qui implique de dupliquer les données. Posons $n_s(I) = 1$ si le stockage est global, et $n_s(I) = n_I$ sinon, et notons $t(X)$ le temps de stockage d'un ensemble de données X. Comme la taille de la base peut évoluer, il est nécessaire de diviser le temps de stockage $t(D)$ en périodes telles que $t(D) = \sum_{p=1}^{n_D} t(D_p)$, où D_p représente la base de données, de taille fixe, pour la période p. Le coût total de stockage est donc la somme des coûts de chaque période.

$$C_s(D, I) = n_s(I) \sum_{p=1}^{n_D} c_s(s(D_p))\, t(D_p) \tag{2.4}$$

2.2. Problème de matérialisation de vues

La matérialisation de vues est un genre de cache permettant de réduire les temps de traitement de requêtes sur une base de données. Une vue est le résultat d'une requête dans la base, et matérialiser une vue consiste à exécuter la requête associée et à en conserver le résultat qui pourra ensuite servir au traitement d'autres requêtes, réduisant ainsi les calculs sur la base de données. En contrepartie de gains sur les temps de réponse des requêtes, la matérialisation induit des coûts de calcul et de stockage liés à la création des vues (traitement des requêtes et stockage des résultats) et à leur maintenance (une mise à jour de la base peut induire une mise à jour de certaines vues).

Plusieurs approches ont été étudiées pour optimiser la matérialisation de vues [Mami12], reposant sur diverses techniques d'optimisation comme une recherche exhaustive, une heuristique gloutonne, des métaheuristiques (recuit simulé, algorithme génétique). La recherche d'une matérialisation se déroule généralement en deux étapes: une présélection des vues potentiellement améliorantes (à partir de l'analyse de la structure des requêtes, e.g. [Bari03]), suivie d'une sélection parmi ces candidates pour déterminer une matérialisation satisfaisante en termes de performance et de coût.

Dans un premier temps, nous choisissons d'étudier la phase de sélection uniquement, avec pour objectif futur d'intégrer la phase de présélection, afin d'obtenir une seule étape d'optimisation. Nous considérons le problème pour une base de données stockée dans un nuage, en nous appuyant sur le modèle de tarification présenté précédemment pour définir les coûts liés à la matérialisation de vues. Le problème est d'abord décrit sous la forme d'un programme linéaire en nombres entiers dont la résolution exacte permet d'obtenir une matérialisation optimale. Cependant, le temps nécessaire à cette résolution n'est pas toujours adapté à un contexte dynamique (i.e. où la charge évolue), il est donc essentiel de disposer de méthodes de résolution rapides, comme des métaheuristiques, qui peuvent fournir des solutions satisfaisantes sans que leur optimalité ne soit garantie.

Nous avons retenu la métaheuristique GRASP (*Greedy Randomized Adaptive Search Procedure*) pour la résolution du problème de matérialisation, car elle s'appuie sur des mécanismes de voisinages et de mouvements qui offrent une certaine souplesse (on peut envisager d'adapter les mouvements si le problème d'optimisation évolue), et qui peuvent être exploités dans le cadre d'une optimisation bicritère. Pour l'instant, nous limitons la résolution du problème à un seul critère: la minimisation du temps cumulé de traitement des requêtes (qui exprime une qualité de service pour les utilisateurs de l'application), avec la contrainte d'un budget limité. L'aspect bicritère est discuté à la section 2.4 dans l'optique de futurs travaux.

2.2.1. Impact sur les coûts

Nous supposons que l'application est déployée sur un nombre fixé n_I d'instances de calcul du même type I_0: $I_j = I_0,\ \forall j = 1..n_I$. Dans la section 2.3, nous abordons l'aspect scalabilité, qui consiste à adapter les quantités de chaque type d'instances de calcul utilisées en fonction de la charge. A l'avenir, nous envisageons d'intégrer la scalabilité au problème de matérialisation. Soit $V_{cand} = \{V_k\}_{k=1..n_V}$ l'ensemble des vues candidates à la matérialisation présélectionnées lors de la première phase (e.g. [Bari03]). L'algorithme de la seconde phase devra sélectionner un sous-ensemble de vues $V \subset V_{cand}$ à matérialiser dans le nuage.

Les deux phases sont supposées s'exécuter hors du nuage, elles n'impliquent donc aucun coût de calcul supplémentaire. En revanche, la matérialisation se déroule dans le nuage, ce qui nécessite des ressources de calcul et de stockage, mais aucun transfert de données avec l'extérieur. La formule 2.2 du coût de transfert C_t exprimée dans le modèle général de tarification reste donc inchangée.

2.2.1.1. Coût de calcul

L'utilisation de vues matérialisées modifie le modèle du coût de calcul: le temps total de calcul $T(Q,V)$ nécessaire à l'application est la somme du temps cumulé $T_{proc}(Q,V)$ de traitement des requêtes de la charge Q en utilisant l'ensemble de vues matérialisées V, du temps $T_{mat}(V)$ pour matérialiser les vues, et du temps $T_{main}(V)$ pour les maintenir.

$$T(Q,V) = T_{proc}(Q,V) + T_{mat}(V) + T_{main}(V)$$

Avec l'hypothèse d'une configuration d'instances fixe et homogène, la formule 2.3 du coût de calcul C_c du modèle général de tarification devient:

$$C_c(Q,I,V) = T(Q,V)\, n_I\, c_c(I_0)$$

Soit $t_{mat}(V_k)$ et $t_{main}(V_k)$ les temps nécessaires, respectivement, à l'exécution de la requête associée à la vue k, et aux mises à jour de la vue k induites par les actualisations de la base de données D. Les temps de matérialisation et de maintenance pour l'ensemble des vues s'expriment alors:

$$T_{mat}(V) = \sum_{V_k \in V} t_{mat}(V_k)$$

$$T_{main}(V) = \sum_{V_k \in V} t_{main}(V_k)$$

Certaines vues matérialisées de l'ensemble V peuvent être exploitées pour réduire le temps de traitement d'une requête Q_i sur la base de données. Notons $t(Q_i, V)$ le temps de traitement d'une requête Q_i en fonction de l'ensemble de vues matérialisées V. Le temps total de traitement des requêtes de la charge Q s'exprime donc:

$$T_{proc}(Q, V) = \sum_{i=1}^{n_Q} f_i \, t(Q_i, V)$$

2.2.1.2. Coût de stockage

L'utilisation de vues matérialisées ne change pas fondamentalement le modèle du coût de stockage: le volume des données stockées, initialement la base de données D, augmente simplement de la taille des vues matérialisées de l'ensemble V. L'ensemble de données D est alors remplacé dans la formule 2.4 par l'ensemble $D \cup V$, où $s(D \cup V) = s(D) + \sum_{V_k \in V} s(V_k)$.

$$C_s(D, I, V) = C_s(D \cup V, I)$$

2.2.2. Formulation linéaire en nombres entiers

Le but est de sélectionner un sous-ensemble de vues V à matérialiser dans l'ensemble de vues candidates V_{cand}. Dans un premier temps, nous posons des hypothèses sur V_{cand}, en raison de la difficulté à déterminer le gain apporté par l'utilisation d'une ou plusieurs vues pour une requête donnée. Deux approches sont possibles: soit modéliser le fonctionnement du nuage pour exprimer analytiquement le gain, soit réaliser des expériences pour obtenir une estimation du gain.

Nous avons retenu la seconde approche, en limitant chaque requête Q_i à n'exploiter qu'une seule vue parmi l'ensemble des vues candidates. Autrement dit, pour chaque requête Q_i, il y a un ensemble $V^i \subset V$ de vues candidates, et pas plus d'une vue dans cet ensemble ne doit être sélectionnée pour la requête Q_i. Dans de futurs travaux, nous pourrons considérer les vues candidates d'une requête Q_i comme étant un ensemble $V^i = \left\{V^{ij}\right\}_{j=1..n_i}$ qui contient n_i ensembles de vues candidats $V^{ij} \subset V$; il s'agira alors de choisir un ensemble de vues V^{ij} pour chaque requête Q_i au lieu d'une seule vue.

Le problème d'optimisation doit sélectionner au mieux une vue V_k pour chaque requête Q_i. Dans ce but, des variables de décision x_{ik} sont introduites: $x_{ik} = 1$ si la requête Q_i utilise la vue V_k, et $x_{ik} = 0$ sinon. Soit t_i le temps de réponse de la requête Q_i sans utiliser aucune vue, et g_{ik} le gain sur t_i lorsque Q_i exploite la vue V_k. Le temps de réponse de la requête Q_i en utilisant les vues s'exprime comme suit.

$$t(Q_i, V) = t_i - \sum_{k=1}^{n_V} g_{ik}\, x_{ik}$$

Les temps de réponse et les gains sont des constantes du problème qui ont été estimées en amont à partir d'expériences [Perr13], mais il est envisagé de les calculer à partir d'un modèle des structures des requêtes et du fonctionnement d'un nuage. Il ne doit pas y avoir plus d'une vue sélectionnée pour la requête Q_i, ce qui est assuré par les contraintes suivantes.

$$\sum_{k=1}^{n_V} x_{ik} \leq 1,\ \forall i = 1..n_Q$$

Des variables de décision x_k sont également introduites pour déterminer si une vue V_k est matérialisée: $x_k = 1$ si la vue V_k est matérialisée, et $x_k = 0$ sinon. Une vue V_k est matérialisée si elle est utilisée par au moins une requête (i.e. s'il existe au moins une requête Q_i telle que $x_{ik} = 1$), ce qui s'exprime par les contraintes suivantes.

$$x_{ik} \leq x_k,\ \forall i = 1..n_Q,\ \forall k = 1..n_V$$

A l'inverse, il n'est pas nécessaire de matérialiser V_k si elle n'est pas utilisée du tout, ce qui s'exprime par les contraintes suivantes.

$$x_k \leq \sum_{i=1}^{n_Q} x_{ik},\ \forall k = 1..n_V$$

Les temps de matérialisation $t_{mat}(V_k)$ et de maintenance $t_{main}(V_k)$ sont également estimés en amont à partir d'expériences. Ils doivent être considérés pour une vue V_k uniquement si celle-ci est matérialisée.

$$T_{mat}(V) = \sum_{k=1}^{n_V} t_{mat}(V_k)\, x_k$$

$$T_{main}(V) = \sum_{k=1}^{n_V} t_{main}(V_k)\, x_k$$

Nous considérons une seule période de stockage d'une durée T au cours de laquelle la taille des données stockées dans le nuage n'évolue pas. La taille de l'ensemble des données $D \cup V$ s'exprime comme suit.

$$S(D \cup V) = s(D) + \sum_{k=1}^{n_V} s(V_k)\, x_k$$

Dans le cas où la tarification c_s du stockage est linéaire par morceaux, elle peut être formulée à l'aide de contraintes linéaires utilisant des variables continues et entières. Considérons que c_s est composée de n segments et s'exprime comme dans l'expression 2.1, chaque segment e étant défini dans l'intervalle des valeurs d'entrée $[S_e; S_{e+1}[$.

$$c_s(S) = a_e(S - S_e) + b_e,\ e \text{ tel que } S_e \leq S < S_{e+1}$$

Des variables de décision y_e sont introduites, telles que $y_e = 1$ si le segment e de la fonction c_s est utilisé (i.e. si $S \in [S_e; S_{e+1}[$) et $y_e = 0$ sinon; ainsi que des variables continues x_e, telles que $x_e \in [S_e; S_{e+1}[$ si le segment e est utilisé et $x_e = 0$ sinon. La fonction c_s s'exprime alors de la manière suivante.

$$c_s(S) = \sum_{e=1}^{n} \left(a_e \left(x_e - S_e\right) + b_e\right)\, y_e = \sum_{e=1}^{n} \left(a_e\, x_e + \left(b_e - a_e\, S_e\right) y_e\right)$$

$$\text{avec } S_e\, y_e \leq x_e \leq S_{e+1}\, y_e,\ \forall e = 1..n \qquad \sum_{e=1}^{n} y_e = 1$$

$$y_e \in \{0,1\},\ \forall e = 1..n \qquad \sum_{e=1}^{n} x_e = S$$

$$x_e \in \mathbb{R},\ \forall e = 1..n$$

Le coût de transfert C_t n'est pas impacté par la matérialisation. Il s'agit donc d'une constante du problème calculée par la formule 2.2. Finalement, le problème d'optimisation est un programme linéaire en nombres entiers (PLNE), noté P_{VM}, dont l'objectif est de minimiser la somme T_{proc} des temps de réponse des requêtes sous la contrainte d'un coût total $C \leq C_{max}$, et se résume comme suit [6].

$$P_{VM} \begin{cases} \text{minimiser } T_{proc} = \displaystyle\sum_{i=1}^{n_Q} f_i \left(t_i - \sum_{k=1}^{n_V} g_{ik}\, x_{ik}\right) & \\ \text{avec } C = C_c + C_t + C_s \leq C_{max} & C_s = n_s\, c_s(S)\, T \\ \quad C_c = \left(T_{proc} + T_{mat} + T_{main}\right)\, n_I\, c_c(I_0) & S = s(D) + \displaystyle\sum_{k=1}^{n_V} s(V_k)\, x_k \\ \quad \displaystyle\sum_{k=1}^{n_V} x_{ik} \leq 1,\ \forall i = 1..n_Q & T_{mat} = \displaystyle\sum_{k=1}^{n_V} t_{mat}(V_k)\, x_k \\ \quad x_{ik} \leq x_k,\ \forall i = 1..n_Q,\ \forall k = 1..n_V & T_{main} = \displaystyle\sum_{k=1}^{n_V} t_{main}(V_k)\, x_k \\ \quad x_k \leq \displaystyle\sum_{i=1}^{n_Q} x_{ik},\ \forall k = 1..n_V & \\ \quad x_{ik} \in \{0,1\},\ \forall i = 1..n_Q,\ \forall k = 1..n_V & \\ \quad x_k \in \{0,1\},\ \forall k = 1..n_V & \end{cases}$$

Ce problème s'apparente au problème du sac à dos (e.g. [Mart00]): les vues à matérialiser peuvent être assimilées à des objets à ranger dans le sac, chaque vue ayant un poids (le coût induit par sa matérialisation) et un profit (le gain sur le temps de traitement induit par sa matérialisation). Le problème de matérialisation est NP-difficile, car plus difficile à résoudre que le problème du sac à dos par le fait que le profit de chaque objet dépend du reste du contenu du sac (i.e. le profit induit par une matérialisation dépend des autres matérialisations).

[6] Pour des raisons de clarté, la notation est simplifiée: les paramètres Q, D, V et I ont été masqués.

2.2.3. Métaheuristique GRASP

GRASP (*Greedy Randomized Adaptive Search Procedure*) est une métaheuristique en deux étapes: construction non-déterministe d'une solution réalisable, suivie d'une amélioration de cette solution par recherche locale [Feo95]. L'approche est à démarrage multiple: les deux étapes sont répétées it_{GR} fois, afin d'effectuer une recherche à partir de différents points de l'espace des solutions, produisant chaque fois une solution x (cf. algorithme 2.1). La meilleure solution x^* est retenue, dans notre cas, il s'agit de celle qui induit un temps cumulé de traitement des requêtes T_{proc} minimal.

Algorithme 2.1: **GRASP(problème P,solution x^*).**

$x^* \leftarrow 0$; $T^*_{proc} \leftarrow +\infty$;

pour $i = 1..it_{GR}$ faire
 heuristiqueConstructive(P,x);
 rechercheLocale(P,x);
 calculer le temps T_{proc} de la solution x;

 si $T_{proc} < T^*_{proc}$ alors $x^* \leftarrow x$; $T^*_{proc} \leftarrow T_{proc}$;
fin pour;

Dans le problème de matérialisation P_{VM}, deux types de variables de décision ont été définies: x_k qui indique si la vue V_k est matérialisée, et x_{ik} qui indique si la vue V_k est exploitée par la requête Q_i. Nous remarquons que si tous les x_k sont fixés, trouver les valeurs optimales de tous les x_{ik} est trivial: il suffit de sélectionner la vue matérialisée V_k qui maximise le gain g_{ik} pour chaque requête Q_i. Dans l'heuristique, une solution sera donc représentée par le vecteur $x = (x_k)_{k=1..n_V}$ uniquement.

La phase de construction est une heuristique gloutonne, i.e. une approche itérative où les choix effectués aux itérations précédentes ne sont pas remis en question, ce qui assure une faible complexité de l'algorithme. A chaque itération, les vues candidates à la matérialisation dans la solution courante sont classées, à l'aide d'une fonction dite gloutonne qui estime le bénéfice d'insérer une vue dans la solution. L'une des meilleures vues de ce classement est sélectionnée aléatoirement pour être insérée dans la solution, ce non-déterminisme permettant d'obtenir une solution différente à chaque itération de GRASP. L'heuristique est dite adaptative car le classement des vues candidates est actualisé à chaque itération de la construction, afin de tenir compte des changements engendrés par la dernière insertion.

La recherche locale consiste à se déplacer dans l'espace des solutions, à partir de la solution fournie par l'heuristique constructive, dans le but de l'améliorer. Ce déplacement repose sur des mouvements, i.e. des opérations de modification de la solution, qui transforment la solution en une solution supposée proche. L'application de tous les mouvements considérés sur la solution courante définit un ensemble de solutions proches appelé voisinage. Le déplacement s'effectue en sélectionnant une solution dans le voisinage. Plusieurs approches sont possibles (e.g. [Burk05]) et nous avons choisi ici de prendre aléatoirement une solution parmi les meilleures du voisinage.

2.2.3.1. Heuristique constructive

La phase de construction, décrite par l'algorithme 2.2, démarre avec une solution sans aucune vue matérialisée. Itérativement, une vue est sélectionnée et matérialisée dans la solution. Le but ici est de générer une solution réalisable, i.e. où le coût C est inférieur à C_{max}. Seules les vues qui réduisent le coût C peuvent donc être insérées dans la solution, celles-ci sont classées en fonction de la réduction du coût induit par leur matérialisation.

Algorithme 2.2: **heuristiqueConstructive(problème P,solution x).**

```
i ← 0;

répéter
 i ← i+1; x ← 0;

 répéter
  L ← ∅;

  pour tout k = 1..n_V tel que x_k = 0 faire
   calculer le bénéfice w_k;
   si w_k > 0 alors L ← L ∪ {V_k};
  fin pour;

  si L ≠ ∅ alors
   trier L par ordre décroissant de w_k;
   sélectionner aléatoirement V_k parmi les premiers sel_HC pourcents de L;
   x_k ← 1;
  fin si;
 jusqu'à L = ∅;

 calculer le coût C de la solution x;
jusqu'à C ≤ C_max ou i = it_HC;
```

Plusieurs indicateurs sont définis pour évaluer l'impact de la matérialisation d'une vue. Soit c_k le coût de matérialisation de la vue V_k, incluant le coût de traitement $\Delta C_c(V_k)$ de sa matérialisation et de sa maintenance, et son coût de stockage $\Delta C_s(V_k)$.

$$c_k = \Delta C_c(V_k) + \Delta C_s(V_k)$$

$$\Delta C_c(V_k) = (t_{mat}(V_k) + t_{main}(V_k))\, n_I\, c_c(I_0)$$

$$\Delta C_s(V_k) = n_s\ (c_s(S + s(V_k)) - c_s(S))\ T$$

Soit g_k le gain sur le temps de traitement des requêtes T_{proc} suite à la matérialisation de la vue V_k, qui est la somme des gains induits sur chaque requête Q_i. La vue V_k induit un gain pour la requête Q_i seulement si $g_{ik} > g_{ih}$, où V_h est la vue actuellement exploitée par la requête Q_i.

$$g_k = \sum_{i=1}^{n_Q} f_i \times \max\left(0,\ g_{ik} - \sum_{h=1}^{n_V} g_{ih}\, x_h\right)$$

Soit w_k le bénéfice de la matérialisation de la vue V_k, qui est la différence entre l'économie liée au gain g_k sur le temps de traitement des requêtes et le coût c_k de matérialisation de la vue.

$$w_k = g_k \, n_I \, c_c(I_0) - c_k$$

A chaque itération de l'heuristique gloutonne, le bénéfice w_k de toute vue V_k qui n'est pas encore matérialisée est calculé. Si $w_k > 0$, la vue est insérée dans une liste L où les vues sont classées par ordre décroissant de w_k. Ensuite, une vue est sélectionnée aléatoirement parmi les premiers sel_{HC} pourcents de la liste L, pour être matérialisée dans la solution x. La procédure se répète jusqu'à ce qu'il n'y ait plus aucune vue candidate à la matérialisation. Finalement, si le coût C de la solution x est supérieur à C_{max}, une nouvelle tentative de construction d'une solution réalisable est lancée. L'heuristique s'arrête après un nombre fixé it_{HC} de tentatives.

2.2.3.2. Recherche locale

La recherche locale, décrite par algorithme 2.3, vise à améliorer la solution réalisable obtenue par l'heuristique constructive, en réduisant le temps de traitement des requêtes T_{proc}. L'heuristique se déplace de solution en solution en matérialisant une vue à chaque itération. Les indicateurs g_k et w_k de chaque vue V_k qui n'est pas encore matérialisée sont calculés. Le voisinage N de x est constitué des solutions ayant une vue matérialisée V_k supplémentaire telle que $g_k > 0$, tout en restant réalisables, i.e. telles que $C - w_k \leq C_{max}$. L'heuristique se déplace sur la solution sélectionnée aléatoirement parmi les sel_{LS} pourcents des meilleures solutions (i.e. ayant les g_k les plus élevés) du voisinage.

Algorithme 2.3: **rechercheLocale(problème P,solution x)**.

```
répéter
 retirer les vues inexploitées de la solution x;
 calculer le coût C de la solution x;
 N ← ∅;

 pour tout k = 1..n_V tel que x_k = 0 faire
  calculer g_k et w_k;
  si g_k > 0 et C − w_k ≤ C_max alors N ← N ∪ {V_k};
 fin pour;

 if N ≠ ∅ alors
  trier N par ordre décroissant de g_k;
  sélectionner aléatoirement V_k parmi les premiers sel_LS pourcents de N;
  x_k ← 1;
 fin si;
jusqu'à N = ∅;
```

Chaque nouvelle vue V_k matérialisée peut rendre certaines matérialisations inutiles: il est possible que des vues matérialisées ne soient plus exploitées par aucune requête, car leur gain est inférieur à celui de V_k. De telles vues peuvent être retirées de la solution, réduisant ainsi le coût total C sans augmenter le temps de traitement des requêtes T_{proc}. L'heuristique s'arrête lorsque le voisinage est vide.

2.2.4. Résultats numériques

Sans aucune garantie sur l'optimalité des solutions obtenues avec l'heuristique GRASP, nous avons donc étudié le comportement de l'algorithme en pratique. A notre connaissance, la seule méthode de résolution exacte du problème de matérialisation consiste à résoudre le programme linéaire en nombres entiers P_{VM}. Des problèmes ont été générés aléatoirement et testés avec trois valeurs différentes pour C_{max}, afin d'analyser le comportement des méthodes avec différentes restrictions de budget. Pour chaque problème, le coût minimum c^{min} (i.e. le coût obtenu en minimisant le budget, sans contrainte sur le temps de traitement T_{proc}) et le coût maximum c^{max} (i.e. le coût obtenu en minimisant T_{proc} sans contrainte sur C_{max}) ont été calculés. Ils servent à définir trois valeurs pour C_{max}: $c_1 = c^{min} + 0.05(c^{max} - c^{min})$ qui est 5 % au dessus de c^{min}, c_2 qui est 15 % au dessus de c^{min}, et c_3 qui est 25 % au dessus de c^{min}.

Le tableau 2.1, extrait de [Perr14], présente une comparaison des temps de calcul (en secondes) de l'heuristique GRASP et de la résolution du PLNE (utilisant le solveur CPLEX[7]), ainsi que le gap (i.e. la différence relative) entre les temps de traitement T_{proc} des solutions de chaque méthode (un gap de n % signifie que le temps T_{proc} de la solution de GRASP est n % au dessus de celui de la solution de CPLEX). Chaque résultat est la moyenne de 5 résolutions de problèmes générés aléatoirement. Si CPLEX ne trouve pas de solution optimale à l'un des problèmes en deux minutes, le gap est calculé avec la meilleure solution obtenue jusque-là (cas marqués avec * dans le tableau).

	$C_{max} = c_1$			$C_{max} = c_2$			$C_{max} = c_3$		
Vues	Temps		Gap	Temps		Gap	Temps		Gap
(n_V)	CPLEX	GRASP	(%)	CPLEX	GRASP	(%)	CPLEX	GRASP	(%)
20	3.4	0.0	0.6	2.2	0.0	0.6	0.9	0.1	0.5
30	9.7	0.1	0.8	6.5	0.1	0.4	3.3	0.1	0.2
40	17.4	0.1	0.4	9.8	0.1	0.3	6.5	0.2	0.3
50	26.4	0.1	0.5	13.5	0.2	0.3	5.8	0.2	0.1
60	47.7	0.2	0.4	13.8	0.2	0.3	3.5	0.3	0.1
70	43.3	0.2	0.4	14.0	0.3	0.3	2.6	0.3	0.0
80	69.2*	0.2	0.3	38.9	0.3	0.3	1.6	0.4	0.0
90	73.4*	0.3	0.8	53.0*	0.4	0.3	2.3	0.5	0.0
100	84.6*	0.3	0.4	45.1	0.5	0.2	1.0	0.6	0.0

Tableau 2.1: Résultats numériques de l'heuristique GRASP pour la matérialisation de vues.

Avec un budget restreint ($C_{max} = c_1$), le temps nécessaire à CPLEX augmente significativement avec le nombre de vues. Dans certains cas, CPLEX n'obtient pas de solution optimale dans les deux minutes. GRASP est plus rapide et fournit des solutions avec un gap inférieur à 1 %. Avec un budget moins contraint ($C_{max} = c_2$), CPLEX résout deux fois plus rapidement les problèmes, et GRASP trouve de meilleures solutions (i.e. le gap est sensiblement plus faible). Avec un budget important ($C_{max} = c_3$), CPLEX résout les problèmes très rapidement, alors que GRASP ne trouve pas toujours une solution optimale (des solutions optimales ont été trouvées uniquement sur les grandes instances du problème).

[7] `http://www.ibm.com/software/integration/optimization/cplex-optimizer`

2.3. Problème de scalabilité

Dans le problème de matérialisation de vues, nous avons considéré que toutes les instances de calcul étaient identiques et que leur nombre était fixé, alors que l'une des particularités d'un nuage informatique est la scalabilité, c'est-à-dire la capacité à pouvoir augmenter le nombre (scalabilité horizontale) et la puissance (scalabilité verticale) des instances de calcul à volonté. Cela peut permettre de répondre quasi instantanément à un pic d'activité de l'application, en demandant des ressources supplémentaires au nuage. Cependant, les fournisseurs d'accès proposent deux manières de louer des ressources de calcul: soit en demandant une instance lors du constat d'un pic d'activité, le coût est alors fonction du temps d'utilisation effectif de la machine (cf. section précédente), sans garantie qu'une instance sera disponible immédiatement; soit en réservant à l'avance une instance pour une longue période (e.g. une année), en prévision d'un pic d'activité, le coût est alors un forfait.

Considérons la problématique de déterminer les types et les quantités d'instances de calcul nécessaires au déploiement d'une application fournissant plusieurs services $i = 1..n$, connaissant la demande $D = \{d_i\}_{i=1..n}$, où d_i est le nombre d'utilisateurs attendus pour le service i sur la période de temps considérée. Pour rester suffisamment général, nous supposons un parc informatique virtuel M pouvant être constitué d'instances de calcul provenant d'un parc existant, de locations dans un nuage, ou d'achats pour étendre le parc existant. Nous considérons donc $M = \{(n_k, r_k, c_k)\}_{k=1..m}$, où n_k est le nombre d'instances de type k disponibles, r_k le coût d'acquisition d'une instance de type k, et c_k le coût d'utilisation par unité de temps d'une instance de type k (cf. section 2.1). Suivant la nature de l'instance, les coûts d'acquisition et d'utilisation peuvent être très différents.

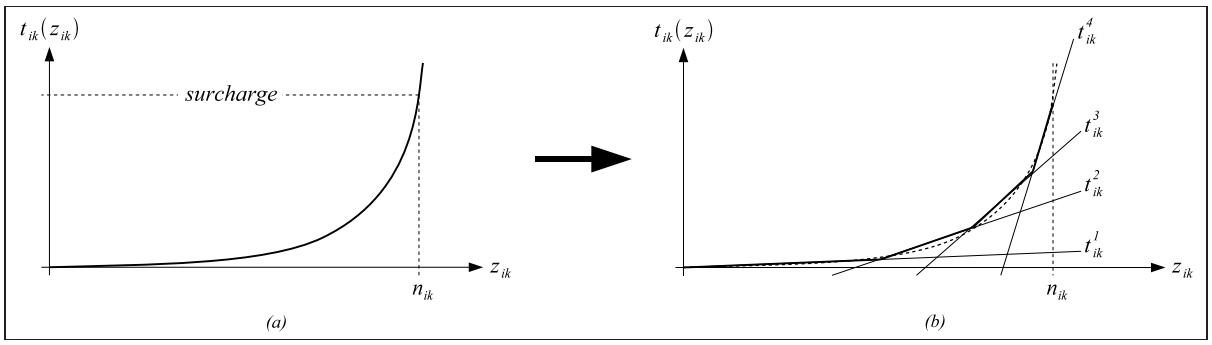

Figure 2.2: Approximation du temps de traitement d'une instance de calcul.

En plus de la composition du parc informatique, il faut déterminer quel travail attribuer à chaque instance. Nous proposons que chaque instance soit dédiée à un seul service, en supposant alors connu le temps de traitement $t_{ik}(z_{ik})$ de z_{ik} demandes de service i sur une instance de type k. Cette fonction convexe dépend de la puissance de l'instance et sera définie à partir de mesures expérimentales ou d'un modèle. Soit n_{ik} le nombre maximal de demandes de service i pouvant être exécutées sur une instance de type k, i.e. le seuil à partir duquel l'instance est considérée en surcharge (cf. figure 2.2a).

2.3.1. Formulation non linéaire

Le problème d'optimisation consiste à déterminer le nombre d'instances x_{ik} de chaque type k qui seront affectées à chaque service i, et le nombre de demandes y_{ik} de service i qui seront traitées par les instances de type k. On considère ici comme objectif la minimisation du coût total C d'exploitation de l'application, sous la contrainte d'un temps de réponse inférieur à T_{max} pour toutes les demandes de service. Le problème d'optimisation s'exprime sous la forme d'un programme non linéaire.

$$P_S \begin{cases} \text{minimiser } C = \displaystyle\sum_{k=1}^{m} r_k \sum_{i=1}^{n} x_{ik} + \sum_{k=1}^{m} c_k \sum_{i=1}^{n} x_{ik}\, t_{ik}(z_{ik}) \\ \text{avec } z_{ik} = \dfrac{y_{ik}}{x_{ik}},\ \forall i = 1..n,\ \forall k = 1..m & (1) \\ z_{ik} \leq n_{ik},\ \forall i = 1..n,\ \forall k = 1..m & (2) \\ \dfrac{t_{ik}(z_{ik})}{z_{ik}} \leq T_{max},\ \forall i = 1..n,\ \forall k = 1..m & (3) \\ \displaystyle\sum_{k=1}^{m} y_{ik} = d_i,\ \forall i = 1..n & (4) \\ \displaystyle\sum_{i=1}^{n} x_{ik} \leq n_k,\ \forall k = 1..m & (5) \\ x_{ik} \in \mathbb{N},\ y_{ik} \in \mathbb{N},\ z_{ik} \in \mathbb{R}^+,\ \forall i = 1..n,\ \forall k = 1..m \end{cases}$$

Le coût total C inclut uniquement le coût d'acquisition et le coût d'utilisation des instances, aucun coût lié au transfert et au stockage n'étant considéré ici. Le nombre moyen $z_{ik} = \dfrac{y_{ik}}{x_{ik}}$ de demandes de service i traitées par les instances de type k permet d'obtenir le temps moyen de traitement $t_{ik}(z_{ik})$ de ces demandes. Les contraintes (4) et (5) imposent respectivement que la demande soit intégralement satisfaite, et que les acquisitions d'instances ne dépassent pas les limites fixées. Les multiplications de variables de décision, que l'on retrouve dans l'objectif et dans les contraintes (1) et (3), ainsi que la nature des fonctions t_{ik}, empêchent le programme d'être linéaire.

2.3.2. Linéarisation

La minimisation du coût C entraîne la minimisation des valeurs $t_{ik}(z_{ik})$, et contrairement aux coûts du modèle de tarification (cf. figure 2.1), les fonctions t_{ik} sont convexes. Dans ces conditions, et en supposant qu'elles soient approximées par des fonctions linéaires par morceaux (e.g. [Magn09]), les fonctions t_{ik} peuvent être représentées sous la forme de contraintes linéaires avec des variables continues uniquement. Considérons ces fonctions avec p morceaux, où chaque segment e est porté par une droite d'équation $t^e_{ik}(z_{ik}) = \alpha^e_{ik}\, z_{ik} + \beta^e_{ik}$ (cf. figure 2.2b).

Les fonctions t_{ik} peuvent être modélisées sous forme linéaire dans le programme P_S en remplaçant $t_{ik}(z_{ik})$ par des variables de décision $u_{ik} \in \mathbb{R}^+$ et en ajoutant les contraintes suivantes.

$$\alpha^e_{ik}\, z_{ik} + \beta^e_{ik} \leq u_{ik},\ \forall e = 1..p,\ \forall i = 1..n,\ \forall k = 1..m$$

La minimisation de u_{ik} dans le programme P_S garantit que l'une des contraintes au moins devient une égalité pour toute solution optimale (car aucune autre contrainte du programme n'impose de borne inférieure à u_{ik}), et donc que la solution se trouve bien sur l'une des droites. Les contraintes imposent d'être au dessus des droites, ce qui assure qu'une solution optimale se trouve toujours sur un segment de la fonction linéaire par morceaux. En remplaçant z_{ik} par $\frac{y_{ik}}{x_{ik}}$, on obtient les contraintes suivantes.

$$\alpha^e_{ik}\, y_{ik} + \beta^e_{ik}\, x_{ik} \leq u_{ik}\, x_{ik},\ \forall e = 1..p,\ \forall i = 1..n,\ \forall k = 1..m$$

Finalement, le remplacement de $u_{ik}\, x_{ik}$, qui représente le temps total de traitement des demandes de service i affectées aux instances de type k, par la variable de décision s_{ik} rend le programme P_S linéaire.

$$P_S \begin{cases} \text{minimiser } C = \displaystyle\sum_{k=1}^{m}\sum_{i=1}^{n} (r_k\, x_{ik} + c_k\, s_{ik}) \\ \\ \text{avec } \alpha^e_{ik}\, y_{ik} + \beta^e_{ik}\, x_{ik} \leq s_{ik},\ \forall e = 1..p,\ \forall i = 1..n,\ \forall k = 1..m \\ \qquad y_{ik} \leq n_{ik}\, x_{ik},\ \forall i = 1..n,\ \forall k = 1..m \\ \qquad s_{ik} \leq T_{max}\, y_{ik},\ \forall i = 1..n,\ \forall k = 1..m \\ \qquad \displaystyle\sum_{k=1}^{m} y_{ik} = d_i,\ \forall i = 1..n \\ \qquad \displaystyle\sum_{i=1}^{n} x_{ik} \leq n_k,\ \forall k = 1..m \\ \qquad x_{ik} \in \mathbb{N},\ y_{ik} \in \mathbb{N},\ s_{ik} \in \mathbb{R}^+,\ \forall i = 1..n,\ \forall k = 1..m \end{cases}$$

2.3.3. Complexité et résolution

Le problème de scalabilité P_S s'apparente à un problème d'affectation généralisé (e.g. [Catt92]), où l'on cherche à minimiser le coût d'affectation de tâches à des processeurs, sachant le coût et le poids (i.e. la quantité de ressources utilisées sur le processeur) de l'affectation d'une tâche j sur un processeur k, et la capacité en ressources de chaque processeur. Supposons que les demandes de chaque service du problème P_S aient été séparées en sous-ensembles pouvant être vus comme des tâches (i.e. chaque tâche j représente d_j demandes d'un service p_j), et que les instances de calcul de chaque type k sont vues comme un processeur k dont la capacité est le nombre d'instances disponibles n_k.

Pour chaque affectation possible d'une tâche j à un processeur k, le poids sera la quantité d'instances nécessaires au traitement de d_j demandes de service p_j, i.e. $w_{jk} = \lceil d_j / n_{p_j k} \rceil$, et le coût sera le cumul du coût d'acquisition des instances $r_k \, w_{jk}$ et d'utilisation $c_k \, t_{p_j k} \left(\frac{d_j}{w_{jk}} \right) w_{jk}$. Le problème de scalabilité est NP-difficile, car plus difficile à résoudre que le problème d'affectation généralisé par le fait qu'un choix de partitionnement des demandes de service s'ajoute à la décision d'affectation.

Dans sa version linéaire, le problème P_S peut être résolu de manière exacte à l'aide d'un solveur comme CPLEX. Des résultats préliminaires [8] montrent des temps de résolution de quelques secondes pour de petites instances du problème (i.e. avec peu de services et de demandes), et pouvant dépasser l'heure de calcul pour de grandes instances. Une heuristique de recherche locale de type VND (*Variable Neighborhood Descent* [Mlad97]) reposant sur les quatre types de mouvements suivants a été étudiée [Reis15]: transférer des instances de type k d'un service i à un service j, transférer des demandes de service i d'un type d'instances k à un type h, supprimer une instance de calcul de type k affectée au service i, ajouter une instance de calcul de type k au service i. Des résultats préliminaires [9] présentent un gap pouvant atteindre 5 % entre les solutions de l'heuristique et celles obtenues avec CPLEX.

2.4. Optimisation bicritère

Afin d'aborder la nature bicritère des deux problèmes présentés dans ce chapitre, nous avons choisi de considérer d'abord les deux critères séparément. Pour le problème de matérialisation par exemple, deux problèmes d'optimisation ont été étudiés, P_{VM1} et P_{VM2}, où chacun des deux critères est optimisé alors que l'autre est seulement contraint par une limite. Le problème P_{VM1} recherche un ensemble de vues qui minimise la somme des temps de réponse des requêtes T_{proc} pour un budget maximal C_{max}, et le problème P_{VM2} recherche un ensemble de vues qui minimise le coût total d'exploitation C pour une somme des temps de réponse maximale T_{max}.

$$P_{VM1} \begin{cases} \text{minimiser } \ T_{proc} \\ \text{avec } \ C \leq C_{max} \\ \qquad \text{contraintes de } P_{VM} \end{cases} \qquad P_{VM2} \begin{cases} \text{minimiser } \ C \\ \text{avec } \ T_{proc} \leq T_{max} \\ \qquad \text{contraintes de } P_{VM} \end{cases}$$

Résoudre le problème d'optimisation bicritère, i.e. minimiser à la fois C et T_{proc}, requiert une approche différente de l'optimisation monocritère. Les deux critères étant contradictoires, il est impossible de déterminer une solution qui les optimise tous les deux. Les algorithmes d'optimisation bicritère cherchent donc plusieurs solutions de compromis, dites non-dominées ou Pareto-optimales, telles qu'elles ne peuvent pas être améliorées pour l'un des critères sans dégrader l'autre critère [Ehrg05]. Ces

[8] Lucas Ferreira Silva et Pedro Paulo Silva. *Modélisation d'un problème de scalabilité dans les nuages informatiques*. Rapport de stage recherche M1. LIMOS-ISIMA, Clermont-Ferrand, 2015.

[9] Gabriel Horikava et Marco Túlio Rodrigues. *Résolution d'un problème de scalabilité dans les nuages informatiques*. Rapport de stage recherche M1. LIMOS-ISIMA, Clermont-Ferrand, 2014.

solutions ne sont pas comparables entre elles sans élément discriminant supplémentaire, et forment un ensemble appelé front de Pareto. Les modèles et les méthodes d'optimisation présentées dans ce chapitre, bien qu'orientés monocritère, sont utiles pour une optimisation bicritère des problèmes étudiés.

Des méthodes exactes d'optimisation bicritère permettent d'obtenir l'intégralité du front de Pareto, notamment la méthode ϵ-contrainte [Béru09] et la méthode en deux phases (*Two-Phases Method* - TPM) [Visé98]. La première approche repose sur des résolutions successives de l'un des deux problèmes monocritère (e.g. P_{VM1} ou P_{VM2}), en adaptant à chaque fois la contrainte sur le second critère. La seconde approche repose sur des résolutions successives d'un problème monocritère ayant pour objectif une combinaison linéaire (évoluant à chaque fois) des deux critères. Pour le problème de matérialisation, cette méthode résout le problème P_{VM3} qui recherche un ensemble de vues offrant un compromis entre minimiser la somme des temps de réponse T_{proc} et minimiser le coût d'exploitation C, à l'aide d'un coefficient α exprimant l'importance relative du premier critère par rapport au second dans le compromis.

$$P_{VM3} \begin{cases} \text{minimiser } \alpha\, T_{proc} + (1-\alpha)\, C \\ \text{avec contraintes de } P_{VM} \end{cases}$$

Des métaheuristiques peuvent aussi être envisagées, comme l'algorithme génétique NSGA-II (*Non-dominated Sorting Genetic Algorithm - II*) [Deb02] ou des adaptations de l'algorithme GRASP [Mart15]. Pour le problème de matérialisation, nous avons proposé une adaptation de l'algorithme NSGA-II exploitant l'heuristique GRASP de la version monocritère, afin d'améliorer par recherche locale certaines solutions de la population de l'algorithme génétique à chaque itération. Des résultats préliminaires [10] montrent une amélioration des solutions non-dominées obtenues en comparaison de la version classique de NSGA-II. Pour le problème de scalabilité, nous avons proposé une heuristique basée sur les mouvements utilisés dans l'algorithme VND de la version monocritère. La méthode enrichit progressivement une population de solutions, en appliquant à chaque itération deux mouvements, chacun améliorant l'un des critères, sur l'une des solutions de la population. Plusieurs stratégies de déplacement dans l'espace des solutions ont été étudiées afin d'accélérer la découverte de solutions non-dominées [Reis15].

2.5. Conclusion

Deux problèmes d'optimisation de ressources, considérés du point de vue de l'utilisateur d'un nuage informatique, ont été modélisés sous la forme de programmes linéaires en nombres entiers (PLNE): un problème de matérialisation de vues pour la mise en oeuvre d'un mécanisme de cache sur une base de données, et un problème de scalabilité pour déterminer la composition du parc informatique et la charge attribuée à chaque machine en fonction des besoins d'une application. Ces problèmes peuvent être résolus de manière exacte, mais avec parfois des temps de calcul inadaptés à un contexte dynamique.

[10] Michael David de Souza Dutra et Vilmar Jefté Rodrigues de Sousa. *Optimisation des coûts des entrepôts de données dans les nuages*. Rapport de stage recherche M1. LIMOS-ISIMA, Clermont-Ferrand, 2013.

Pour résoudre le problème de matérialisation, nous avons proposé une heuristique de type GRASP qui repose sur la définition d'indicateurs mesurant les impacts de la matérialisation d'une vue sur le coût et les performances de la base de données. Cette approche permet d'obtenir rapidement de bonnes solutions (cf. tableau 2.1), avec un gap inférieur à 1 % par rapport aux solutions exactes obtenues par la résolution du PLNE. Cependant, nous n'avons abordé qu'une partie de la problématique de sélection des vues, en supposant donnée une liste de vues candidates. Il est donc envisagé d'intégrer la phase de présélection des vues au processus d'optimisation, à l'aide d'un modèle de la structure des requêtes sur la base de données (e.g. [Bari03]).

Notre formulation du problème de scalabilité n'est pas linéaire, notamment à cause des temps de traitement des instances de calcul qui dépendent de leur charge. L'approximation de ces temps par des fonctions linéaires par morceaux a permis d'obtenir un PLNE et donc de bénéficier de méthodes de résolution exacte. Une heuristique de type VND reposant sur quatre types de mouvements a également été étudiée, mais ses performances restent pour l'instant à améliorer, aussi bien en termes de rapidité que de qualité. Le problème de scalabilité est voué à être intégré au problème de matérialisation, afin obtenir un problème de décision plus global sur le déploiement d'une base de données.

La nature des problèmes étudiés est bicritère, avec des objectifs contradictoires de coût et de performance. Des techniques d'optimisation bicritère ont été étudiées, qu'elles soient exactes ou approchées. Les travaux réalisés pour le monocritère servent de support à ces approches: les modèles monocritère (critère simple ou combinaison linéaire de critères) sont utilisés par les méthodes exactes, et les stratégies des heuristiques monocritère (e.g. les mouvements de voisinage) sont reprises pour élaborer des heuristiques bicritère. Une étude comparative approfondie des résultats produits par les méthodes exactes et approchées est nécessaire pour juger de la pertinence des solutions de compromis obtenues.

PARTIE II - SIMULATION NUMÉRIQUE

CHAPITRE 3
ENRICHISSEMENT DE MODÈLE D'OPTIMISATION

La plupart des méthodes performantes d'optimisation s'appuient sur des modèles mathématiques avec d'importantes simplifications du système étudié. Les solutions produites, dont on a généralement la preuve qu'elles sont les meilleures (ou de bonne qualité), peuvent se révéler inadaptées au système réel et donc ne pas apporter les améliorations attendues. La simulation peut manipuler des modèles plus détaillés, ce qui permet une évaluation plus réaliste de la performance d'une solution. Nous proposons ici un couplage entre optimisation et simulation qui tente d'améliorer le réalisme de la solution produite par l'optimisation du modèle mathématique.

Contrairement aux approches traditionnelles de couplage optimisation-simulation qui tentent l'optimisation de la fonction objectif du problème évaluée par simulation, la technique proposée, que nous appelons enrichissement de modèle, cherche à affiner le modèle mathématique grâce à des mesures obtenues par simulation. Ces différentes approches d'optimisation sont étudiées pour un problème de conception de tournées de bus dans un réseau urbain, et sont comparées en termes de performance et de qualité.

3.1. Problème de conception de tournées de bus

Considérons le problème de conception de système de transport suivant: une compagnie de transport doit définir des tournées de bus, sous certaines contraintes, afin de satisfaire au mieux les demandes des usagers. Des modélisations mathématiques de ce problème ont été introduites dans [Yon05], mais nous ne présentons ici que les éléments essentiels à la compréhension de notre étude. Le réseau urbain est modélisé par un graphe orienté $G = (X, U)$, où X est l'ensemble des noeuds et U l'ensemble des arcs. Les noeuds représentent les arrêts de bus potentiels ou les croisements, et les arcs des portions de rue entre deux arrêts ou croisements (cf. figure 3.1).

Supposons que les demandes des usagers, i.e. les déplacements porte-à-porte souhaités par les usagers, sont connus. Une demande d'usagers $d \in D$ est définie par les données suivantes: o_d, s_d, Q_d, t_d^{min}, t_d^{max}. $o_d \in X$ est le noeud de départ du déplacement, $s_d \in X$ le noeud de destination, Q_d le nombre d'usagers pour cette demande. t_d^{min} et t_d^{max} sont les temps de référence de la demande: t_d^{min} est le temps nécessaire à un bus pour se déplacer de o_d à s_d; et t_d^{max} est le temps de trajet d'un piéton de o_d à s_d. Dans cette étude, nous limitons le problème à rechercher un système de transport Γ qui respecte les contraintes suivantes:

- Γ est un ensemble de tournées dans le graphe G (dans un souci de clarté, nous ne considérons qu'une seule tournée par la suite);

- la longueur des tournées de bus, i.e. le temps nécessaire à un bus pour parcourir son circuit, doit être inférieure à un seuil donné T;
- Γ doit maximiser la satisfaction des usagers.

Notons t_d le temps nécessaire à un usager avec la demande d pour voyager à travers le réseau en utilisant le système de transport Γ; t_d inclut les temps de marche à pied, d'attente aux arrêts et de voyage en bus. La satisfaction de l'usager peut être modélisée par une fonction $\phi_d(t_d)$ définie comme suit: si $t_d \leq 2t_d^{min}$ alors l'usager est pleinement satisfait, i.e. $\phi_d(t_d) = 1$; si $t_d \geq 2t_d^{max}$ alors l'usager n'est pas satisfait du tout, i.e. $\phi_d(t_d) = 0$; et entre ces deux extrêmes, plus t_d est faible, plus la satisfaction augmente (cf. figure 3.2).

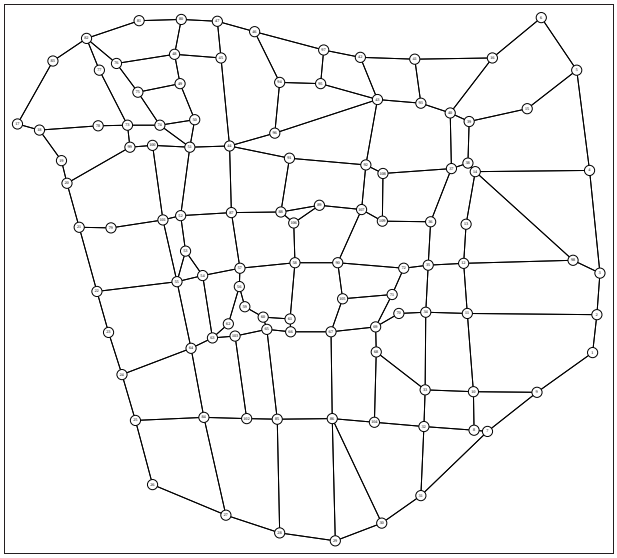

Figure 3.1: Graphe modélisant les rues du centre-ville de Clermont-Ferrand.

Les temps d'attente des usagers aux arrêts doivent être pris en compte dans le calcul des temps de trajet t_d. Notons w_i le temps d'attente à l'arrêt i. L'une des simplifications du modèle mathématique concerne ces temps d'attente qui sont considérés constants dans la formulation du problème. Soit $x = \{x_e\}_{e \in X}$ une solution, où $x_e = 1$ si le noeud e fait partie de la tournée de bus, et $x_e = 0$ sinon. L'objectif du problème est de maximiser la satisfaction des usagers, i.e. la fonction $f(x) = \sum_{d \in D} Q_d \phi_d(t_d(x))$.

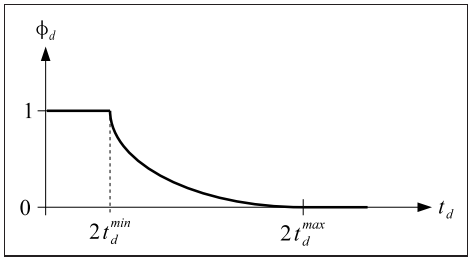

Figure 3.2: Fonction de satisfaction des usagers pour une demande d.

3.2. Amélioration d'une optimisation théorique

De nombreuses techniques avancées (programmation linéaire en nombres entiers: *branch-and-bound*, *branch-and-cut*... [Nemh99], méthodes de décomposition: Benders, Dantzig-Wolfe... [Lasd70]) permettent de résoudre efficacement des problèmes formulés avec des modèles mathématiques. Des résultats fondamentaux ont été établis pour démontrer l'optimalité ou la qualité des solutions obtenues (les algorithmes d'approximation notamment assurent un niveau de qualité déterminé par rapport à l'optimum [Hoch97]), et pour garantir l'efficacité des méthodes (leur complexité, leur vitesse de convergence vers l'optimum...).

Cependant, ces méthodes souffrent d'inconvénients majeurs dès qu'elles sont mises en oeuvre sur des cas pratiques. Tout d'abord, elles ne sont pas robustes face aux changements dans la structure du modèle: modifier le type de certaines contraintes peut rendre le problème insoluble avec la méthode d'optimisation précédemment employée (e.g. des contraintes linéaires, adaptées à la méthode du simplexe, remplacées par des contraintes non linéaires). Deuxièmement, des simplifications majeures dans le modèle doivent souvent être considérées, produisant ainsi des solutions optimales en théorie qui peuvent être mauvaises en pratique.

Nous proposons donc un couplage entre optimisation et simulation appelé enrichissement de modèle qui tente de renforcer le modèle mathématique pour rendre les solutions obtenues mieux adaptées au problème réel. Cette approche s'inspire des méthodes de décomposition utilisées pour la résolution exacte de problèmes d'optimisation, comme la décomposition de Benders et la génération de colonnes [Lasd70]. Le principe du couplage traditionnel optimisation-simulation est rappelé avant d'étudier les résultats d'une optimisation directe d'un modèle mathématique et les améliorations apportées par l'enrichissement de modèle.

3.2.1. Couplage optimisation-simulation

3.2.1.1. Principe et formulation

Un problème d'optimisation peut être exprimé comme la recherche d'une meilleure solution x à un problème réel P_r, i.e. minimiser ou maximiser une fonction $f_r(x)$. Une solution x est réalisable pour le problème P_r si elle satisfait un ensemble de contraintes, ce dernier définissant l'espace C_r des solutions réalisables. Dans notre exemple, x représente une tournée de bus dans la ville, C_r les contraintes sur cette tournée (sa longueur, les rues qu'elle peut utiliser...), et $f_r(x)$ représente la satisfaction des usagers utilisant cette tournée x.

$$P_r \begin{cases} \text{optimiser} \;\; f_r(x) \\ \text{avec} \;\; x \in C_r \end{cases}$$

La modélisation du problème réel P_r entraînera inévitablement des approximations. Considérons un couplage optimisation-simulation traditionnel formulé par le problème P_s. Le fait qu'une fonction (respectivement un espace de solutions) b approxime une autre fonction (respectivement un autre espace) a est noté $b \sim a$.

$$P_s \begin{cases} \text{optimiser } \ f_s(x) = g(x, \lambda), \ f_s \sim f_r \\ \text{avec} \quad x \in C_s \\ \qquad\quad \lambda \in \Lambda_s(x) \end{cases} \sim C_r$$

C_s représente des contraintes sur la solution x. Il définit généralement la structure fondamentale d'une solution réalisable (e.g. la tournée de bus doit être un circuit dans le graphe représentant les rues de la ville). λ est un vecteur de mesures (e.g. les temps de trajet des usagers) évaluées par la simulation de la solution x (cf. figure 3.3).

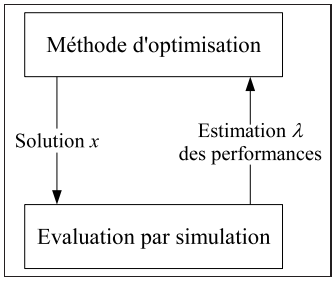

Figure 3.3: Principe du couplage optimisation-simulation classique.

$\Lambda_s(x)$ représente les contraintes qui déterminent λ pour une solution x donnée. Ce sont des contraintes implicites du modèle de simulation (pour une solution x donnée, la simulation retourne le vecteur de mesures λ), et des contraintes explicites (e.g. les temps de trajet des usagers ne doivent pas excéder une certaine limite). Cela signifie que $\Lambda_s(x)$ contient un vecteur λ de mesures évaluées par simulation (si x est réalisable d'après l'évaluation de la simulation), ou bien est vide (si x n'est pas réalisable d'après l'évaluation de la simulation).

La fonction objectif f_r du problème réel est approximée par la fonction f_s. Nous proposons de paramétrer cette fonction sur le vecteur λ: $f_s(x) = g(x, \lambda)$. Dans notre exemple, λ représente les temps de trajet des usagers, estimés par simulation, de la solution x, et $g(x, \lambda)$ est l'évaluation de la satisfaction des usagers pour cette solution x relativement aux temps estimés.

Le couplage optimisation-simulation explore l'ensemble des solutions C_s afin d'optimiser la fonction f_s. Pour chaque solution x, la simulation (représentée par les contraintes implicites de $\Lambda_s(x)$) évalue le vecteur λ. Si λ satisfait les contraintes explicites de $\Lambda_s(x)$, la solution x est candidate pour l'optimisation, et la fonction objectif $g(x, \lambda)$ peut être calculée. Sinon, la solution x est simplement rejetée.

Plusieurs méthodes peuvent être proposées pour résoudre le problème de couplage P_s. Une classification en quatre approches majeures a été établie dans [Merk94] et [Azad92]: recherche basée sur le gradient, approximation stochastique, surface de réponse et recherche heuristique. Ces méthodes sont robustes face aux changements de fonction objectif ou de contraintes du problème. Cependant, elles ne représentent que quelques techniques d'optimisation dont l'efficacité et la convergence ne sont pas toujours garanties.

3.2.1.2. Exemple

Dans notre exemple, nous proposons d'utiliser la métaheuristique de recherche tabou [Glov97, Herz97] pour le couplage optimisation-simulation P_s. Une bonne solution x est recherchée dans l'espace des solutions C_s qui contient les tournées de bus avec une longueur inférieur à T. Une tournée dans C_s est également contrainte à prendre la forme d'une géodésique afin de simplifier la recherche [Yon05]. Une géodésique est un circuit défini par un nombre limité de points dits de contrôle. Chaque point de contrôle est relié à son successeur dans le circuit par un plus court chemin (cf. figure 3.4).

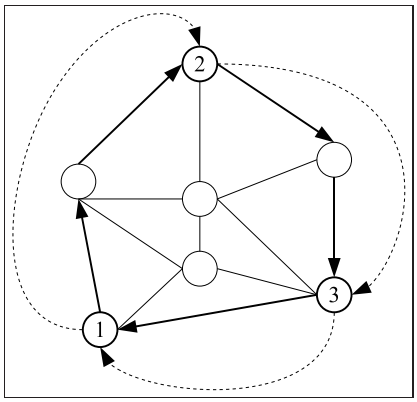

Figure 3.4: Exemple d'une géodésique à 3 points de contrôle.

Les structures de voisinage de la métaheuristique reposent sur le déplacement, l'ajout ou la suppression d'un point de contrôle. La liste tabou contient les points de contrôle qui ont été modifiés dans les dernières itérations. Nous avons implémenté une heuristique à deux niveaux utilisant un critère d'aspiration (acceptation d'un mouvement tabou sous certaines conditions) et une stratégie de diversification (changement de zone d'exploration dans l'espace des solutions).

Pour une solution donnée x du problème de couplage P_s, la simulation estime les temps de trajet t_d des usagers avec la demande d (rappel: toutes ces valeurs sont stockées dans le vecteur λ). En dehors des contraintes implicites liées à la simulation, aucune autre contrainte n'est définie dans Λ_s. Cela signifie que n'importe quelle solution $x \in C_s$ est réalisable pour P_s.

3.2.2. Optimisation théorique

3.2.2.1. Principe et formulation

Les techniques d'optimisation, indépendantes de la simulation, nécessitent souvent un modèle mathématique P_o qui est une approximation du problème réel P_r.

$$P_o \begin{cases} \text{optimiser } \ f_o(x),\ f_o \sim f_r \\ \text{avec } \ x \in C_o,\ C_o \sim C_r \end{cases}$$

Mais nous pouvons raisonnablement supposer que le problème de couplage optimisation-simulation P_s est plus proche du problème réel que le problème d'optimisation théorique P_o: dans notre exemple, le modèle d'optimisation P_o suppose que les temps d'attente aux arrêts sont déterministes, contrairement au modèle de simulation P_s qui, par nature, considère des temps d'attente stochastiques. Dans l'optique d'un enrichissement de modèle, nous émettons quelques hypothèses sur la structure de P_o:

$$P_o \begin{cases} \text{optimiser } \ f_o(x) = g(x, \lambda) \\ \text{avec } \ x \in C_o \supseteq C_s \\ \qquad \lambda \in \Lambda_o(x),\ \Lambda_o \sim \Lambda_s \end{cases}$$

Les contraintes C_s décrivent la structure fondamentale d'une solution réalisable. Ainsi, nous pouvons raisonnablement supposer que les contraintes qui définissent C_o sont des approximations, voire même des relaxations, des contraintes définissant C_s. Cela signifie que $C_o \supseteq C_s$. Par exemple, C_s peut forcer une tournée de bus à être un circuit élémentaire (i.e. qui ne passe pas deux fois par le même noeud), le couplage optimisation-simulation explorera ainsi des solutions plus réalistes; alors que C_o peut autoriser n'importe quel type de circuit par souci de simplification.

A l'instar de $\Lambda_s(x)$, $\Lambda_o(x)$ représente les contraintes qui déterminent λ pour une solution x donnée. Elles modélisent le calcul de λ dans le modèle mathématique. Nous considérons que les fonctions objectif f_o et f_s sont identiques: $f_o(x) = g(x, \lambda)$. Cependant, nous supposons que Λ_o est une approximation de Λ_s, ce qui induit que λ calculé à partir de Λ_o est une approximation de λ calculé à partir de Λ_s, et donc que f_o est implicitement une approximation de f_s. Dans notre exemple, $\Lambda_s(x)$ définit une estimation statistique des temps de trajet des usagers, alors que $\Lambda_o(x)$ définit un calcul déterministe.

Selon la structure du problème, diverses techniques d'optimisation sont envisageables pour résoudre P_o. Cependant, une fois la méthode retenue, il est difficile de gérer des changements dans la nature des contraintes du problème. Avec l'enrichissement de modèle, nous considérons des changements uniquement sur les contraintes qui décrivent Λ_o. Soit K_m l'ensemble de toutes les familles de contraintes pour Λ_o qui peuvent être supportées par la méthode m, i.e. le problème P_o peut être résolu par la méthode m seulement si $\Lambda_o \in K_m$.

3.2.2.2. Exemple

Parmi plusieurs méthodes de résolution possibles [Yon05], la recherche tabou a été retenue pour résoudre le problème d'optimisation P_o. L'heuristique est similaire à celle présentée pour résoudre le couplage P_s, mais cette fois-ci la solution est recherchée dans l'espace C_o. Il est facile de conserver la structure géodésique, ce qui signifie que $C_o = C_s$ dans notre exemple.

Nous avons affirmé précédemment que Λ_o est une approximation de Λ_s. Les contraintes de simulation qui définissent Λ_s sont remplacées par des contraintes déterminant les temps de trajet t_d dans Λ_o. Les temps d'attente w_i sont supposés fixes pour tous les arrêts i, dans l'hypothèse où la fréquence des bus sera adaptée pour rendre les temps d'attente indépendants du tracé de la tournée.

3.2.3. Enrichissement de modèle

3.2.3.1. Principe et formulation

D'après les hypothèses précédentes, si $\Lambda_o(x)$ n'est jamais vide, toute solution $x \in C_o$ est réalisable pour le problème P_o. Comme $C_o \supseteq C_s$, toute solution du couplage optimisation-simulation P_s est réalisable pour P_o. En particulier, tout solution optimale x_s^* de P_s est réalisable pour P_o. L'enrichissement de modèle consiste à rechercher une famille de contraintes $\Lambda_o \in K_m$ telle que la solution optimale x_o^* de P_o soit une solution optimale x_s^* de P_s. Le problème de l'enrichissement de modèle P_e peut se définir comme suit:

$$P_e \begin{cases} \text{optimiser } \ f_s(x_o^*) = g(x_o^*, \lambda_s) \\ \text{avec } \ \Lambda_o \in K_m, \ \Lambda_o \sim \Lambda_s \\ \qquad x_o^* \text{ solution optimale de } P_o \\ \qquad \lambda_s \in \Lambda_s(x_o^*) \end{cases}$$

Considérons l'évaluation théorique λ_o^* (fournie par l'optimisation de P_o) du vecteur λ pour la solution optimale x_o^*, et l'évaluation par simulation λ_s de λ pour la même solution x_o^*. P_e est un problème très difficile, mais une approche possible consisterait à affiner l'approximation Λ_o de Λ_s aussi précisément que possible. Ainsi, λ_s et λ_o^* auraient des valeurs similaires et l'évaluation théorique $g(x_o^*, \lambda_o^*)$ de la solution optimale x_o^* du problème P_o serait proche de son évaluation par simulation $g(x_o^*, \lambda_s)$.

Cette approche est inspirée de méthodes de décomposition comme celle de Benders ou la génération de colonnes [Lasd70], qui séparent un problème en deux parties: le problème maître relâché, qui est une relaxation du problème original, et le problème auxiliaire, dont la résolution fournit des informations utiles pour enrichir le problème maître. Par un processus itératif, la résolution de problèmes auxiliaires permet d'ajouter des contraintes ou des variables (selon l'approche de décomposition) dans le problème maître, qui tend progressivement vers le problème original, et permet ainsi d'obtenir une solution optimale.

Nous proposons une décomposition similaire pour l'enrichissement de modèle: P_o est le problème maître, et la simulation (représentée par Λ_s) d'une solution correspond à la résolution d'un problème auxiliaire (cf. figure 3.5). Dans notre exemple, le vecteur λ_s contenait jusqu'à présent les temps de trajet des usagers, nous lui ajoutons maintenant les temps d'attente. Le problème P_o suppose des temps d'attente constants, mais l'évaluation λ_s de sa solution optimale x_o^* par simulation fournit des estimations de ces temps. Il s'agit donc de trouver un moyen de modifier Λ_o à l'aide des mesures de λ_s de manière à mieux approximer Λ_s.

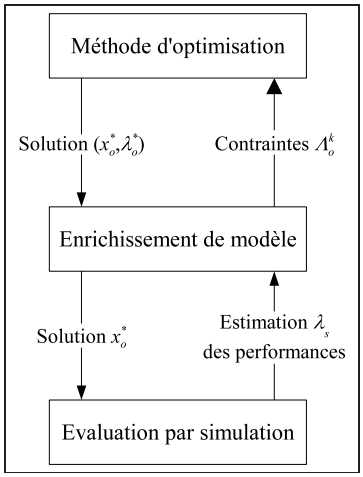

Figure 3.5: Approche par enrichissement de modèle.

L'algorithme 3.1 décrit l'approche heuristique de l'enrichissement de modèle modifiant progressivement Λ_o. Une fonction h applique, à partir de $\Lambda_o = \Lambda_o^k$ et de l'évaluation par simulation λ_s de la solution x_o^* à l'itération k, une transformation pour obtenir $\Lambda_o = \Lambda_o^{k+1}$ à l'itération $k+1$.

Algorithme 3.1: **Heuristique d'enrichissement de modèle.**

$k \leftarrow 0$;
soit n le nombre maximal d'itérations;
soit Λ_o^k une approximation de Λ_s;

répéter
 soit (x_o^*, λ_o^*) une solution optimale de P_o avec $\Lambda_o = \Lambda_o^k$;
 soit λ_s une évaluation par simulation de x_o^*;
 $\Lambda_o^{k+1} \leftarrow h(\Lambda_o^k, \lambda_s)$;
 $k \leftarrow k+1$;
jusqu'à $|g(x_o^*, \lambda_o^*) - g(x_o^*, \lambda_s)| < \varepsilon$ ou $k \geq n$;

Dans cette approche, la difficulté évidente est de définir la fonction h telle que l'évaluation théorique de la fonction objectif $g(x_o^*, \lambda_o^*)$ converge vers l'évaluation par simulation de la fonction objectif $g(x_o^*, \lambda_s)$. Cette question restera ouverte ici, car il est difficile pour l'instant d'y apporter une réponse générale. Elle sera uniquement étudiée dans le cadre de notre exemple.

3.2.3.2. Exemple

Dans le problème P_o de notre exemple, les temps d'attente w_i sont supposés constants à chaque arrêt i, quelle que soit la tournée de bus élaborée. Ces contraintes peuvent faire de Λ_o une mauvaise approximation de Λ_s. Comme la méthode utilisée pour résoudre P_o peut gérer n'importe quel temps d'attente constant, nous proposons de modifier ces temps d'attente théoriques à chaque itération de l'enrichissement de modèle (afin de faire de Λ_o une meilleure approximation de Λ_s).

L'approche retenue pour modifier les temps d'attente théoriques est relativement simple, mais il s'agit avant tout de montrer le potentiel de l'enrichissement de modèle. Pour que l'heuristique fonctionne, nous supposons que la solution optimale x_o^* fournie par P_o est assez robuste, ce qui signifie que si l'on change légèrement les temps d'attente, la nouvelle solution optimale x_o^* ne sera pas trop éloignée de la précédente. L'algorithme 3.2 résume l'heuristique d'enrichissement de modèle pour notre problème de tournées de bus.

Algorithme 3.2: **Enrichissement du modèle de tournées de bus.**

$k \leftarrow 0$;
pour chaque noeud $i \in X$ faire $w_i^k \leftarrow 0$;

répéter
 résoudre P_o avec les temps d'attente w_i^k;
 soit (x_o^*, λ_o^*) une solution optimale de P_o;
 soit $\lambda_s = (w\ t)$ l'évaluation par simulation de x_o^*;
 actualiser la moyenne M des temps d'attente estimés;

 [Calcul de $\Lambda_o^{k+1} \leftarrow h(\Lambda_o^k, \lambda_s)$ *]*
 pour chaque noeud $i \in X$ faire
  si $w_i \neq 0$ alors $w_i^{k+1} \leftarrow w_i^k + \Delta(w_i - w_i^k)$;
  sinon $w_i^{k+1} \leftarrow w_i^k + \Delta(M - w_i^k)$;
 fin pour;

 $k \leftarrow k + 1$;
jusqu'à $|g(x_o^*, \lambda_o^*) - g(x_o^*, \lambda_s)| < \varepsilon$ ou $k \geq n$;

Notons w_i^k les temps d'attente pour le problème P_o à l'itération k, et $\lambda_s = (w\ t)$, composé des vecteurs $w = (w_i)_{i \in X}$ (les temps d'attente estimés) et $t = (t_d)_{d \in D}$ (les temps de trajet estimés), l'évaluation par la simulation de la solution optimale théorique x_o^* à l'itération k. La fonction h transforme Λ_o^k en Λ_o^{k+1} en calculant les temps d'attente w_i^{k+1} à partir des temps w_i^k et des temps w_i estimés par simulation.

Si $w_i \neq 0$, cela signifie que le noeud i a été utilisé comme arrêt de bus au cours de la dernière simulation. Nous proposons donc de rapprocher w_i^{k+1} de w_i comme suit: $w_i^{k+1} \leftarrow w_i^k + \Delta(w_i - w_i^k)$. Si $w_i = 0$, alors aucun usager n'a attendu de bus sur le noeud i au cours de la dernière simulation. Néanmoins, nous proposons de rapprocher w_i^{k+1} de la moyenne M des temps d'attente (calculée depuis le début de l'algorithme) comme suit: $w_i^{k+1} \leftarrow w_i^k + \Delta(M - w_i^k)$. $\Delta < 1$ est un pas de progression qui doit être réglé. Il peut être choisi constant ou variable au fil des itérations (décroissant, il assure une convergence).

3.2.4. Résultats numériques

La performance et la qualité des solutions des trois approches d'optimisation discutées précédemment sont comparées pour le problème de tournées de bus avec un graphe avec 109 noeuds et 392 arcs qui représente le centre-ville de Clermont-Ferrand (cf. figure 3.1), et 12 demandes d'usagers. Afin de tester différentes structures de problèmes, les algorithmes ont été exécutés avec différentes valeurs imposées pour le nombre de points de contrôle des géodésiques, ainsi que pour la longueur des tournées. Les résultats présentés ici sont extraits de [Bach07b].

Points de contrôle	Longueur maximale T	Meilleure évaluation $f_s(x_s^*)$	Nombre d'évaluations	Temps de calcul (heures)
5	700000	10.33	3793	5h04
5	800000	10.30	5203	7h04
5	900000	10.32	4807	6h08
5	1000000	10.32	7573	9h44
10	700000	10.32	8843	11h20
10	800000	10.33	9706	10h12
10	900000	10.31	10732	13h52
10	1000000	10.34	8812	11h04
15	700000	10.31	6074	8h20
15	800000	10.32	8759	10h08
15	900000	10.33	10699	11h16
15	1000000	10.33	10302	11h08

Tableau 3.1: Performance du couplage optimisation-simulation.

Les temps d'exécution du couplage optimisation-simulation P_s pouvant dépasser la journée, nous avons choisi de limiter le nombre de diversifications. D'après le tableau 3.1, le nombre d'évaluations par simulation pour résoudre P_s est élevé, et sans notre restriction, il aurait dépassé les 100000 évaluations. Ainsi, les solutions obtenues ici ne sont pas toujours les meilleures que le couplage P_s peut fournir.

Points de contrôle	Longueur maximale T	Optimisation théorique Eval. théorique $f_o(x_o^*)$	Optimisation théorique Eval. simulation $f_s(x_o^*)$	Optimisation-simulation Meilleure évaluation $f_s(x_s^*)$	Temps de calcul (secondes)
5	700000	11.99	10.29 (-0.4 %)	10.33	5.0
5	800000	11.99	10.25 (-0.5 %)	10.30	5.0
5	900000	11.99	9.62 (-6.8 %)	10.32	5.1
5	1000000	11.99	10.08 (-2.4 %)	10.32	5.3
10	700000	11.97	10.18 (-1.4 %)	10.32	7.7
10	800000	11.99	9.92 (-4.0 %)	10.33	8.9
10	900000	11.99	10.05 (-2.5 %)	10.31	9.4
10	1000000	12.00	9.33 (-9.8 %)	10.34	10.3
15	700000	11.99	10.29 (-0.2 %)	10.31	8.0
15	800000	11.99	10.01 (-3.0 %)	10.32	9.0
15	900000	11.99	9.85 (-4.6 %)	10.33	9.8
15	1000000	12.00	9.56 (-7.5 %)	10.33	10.8

Tableau 3.2: Performance et qualité de l'optimisation théorique.

Le tableau 3.2 compare l'évaluation $f_o(x_o^*)$ de la meilleure solution x_o^* de l'optimisation théorique P_o avec son évaluation par simulation $f_s(x_o^*)$. La résolution de P_o est assez rapide (quelques secondes) comparée à celle du couplage P_s (plusieurs heures), mais les solutions obtenues n'ont pas toujours une très bonne évaluation par simulation. Entre parenthèses est indiquée la différence relative $(f_s(x_o^*) - f_s(x_s^*))/f_s(x_s^*)$.

Pour l'enrichissement de modèle, nous avons choisi de fixer $\Delta = 0.1$ et d'arrêter l'heuristique après $n = 100$ itérations à moins d'obtenir une convergence, i.e. si la différence entre l'évaluation théorique $f_o(x_o^*) = g(x_o^*, \lambda_o^*)$ et l'évaluation par simulation $f_s(x_o^*) = g(x_o^*, \lambda_s)$ est inférieure à $\varepsilon = 0.01$. Bien que la convergence ne puisse pas être garantie en théorie, le tableau 3.3 montre qu'elle a toujours été obtenue. Le tableau compare aussi l'évaluation par simulation $f_s(x_e^*)$ de la meilleure solution x_e^* de l'enrichissement de modèle P_e avec l'évaluation par simulation $f_s(x_o^*)$ de la meilleure solution x_o^* de l'optimisation théorique P_o. Entre parenthèses est indiquée la différence relative $(f_s(x_e^*) - f_s(x_o^*))/f_s(x_o^*)$.

Points de contrôle	Longueur maximale T	Enrichissement de modèle			Optimisation théorique	Optimisation simulation	Temps de calcul
		Eval. théorique $f_o(x_e^*)$	Eval. simulation $f_s(x_e^*)$	Nombre Itérations	$f_s(x_o^*)$	$f_s(x_s^*)$	(minutes)
5	700000	10.48	**10.48** (+1.8 %)	51	10.29	10.33	7'46
5	800000	10.38	**10.38** (+1.3 %)	51	10.25	10.30	7'47
5	900000	10.20	10.20 (+6.0 %)	45	9.62	**10.32**	6'53
5	1000000	10.23	10.23 (+1.5 %)	49	10.08	**10.32**	7'20
10	700000	10.47	**10.47** (+2.8 %)	50	10.18	10.32	9'08
10	800000	10.41	**10.41** (+4.9 %)	33	9.92	10.33	7'01
10	900000	10.12	10.12 (+0.7 %)	56	10.05	**10.31**	12'18
10	1000000	10.05	10.05 (+7.7 %)	64	9.33	**10.34**	15'00
15	700000	10.48	**10.49** (+1.9 %)	51	10.29	10.31	10'02
15	800000	10.42	**10.42** (+4.0 %)	44	10.01	10.32	9'17
15	900000	10.22	10.22 (+4.0 %)	56	9.85	**10.33**	12'35
15	1000000	10.06	10.06 (+5.2 %)	61	9.56	**10.33**	14'41

Tableau 3.3: Performance et qualité de l'enrichissement de modèle.

L'enrichissement de modèle est plus lent que l'optimisation théorique, mais vraiment plus rapide que le couplage optimisation-simulation car il nécessite beaucoup moins d'évaluations par simulation. L'évaluation par simulation de la solution obtenue par enrichissement de modèle est généralement proche de celle obtenue par couplage optimisation-simulation, et toujours meilleure que celle obtenue par optimisation théorique.

La figure 3.6 montre l'évolution et la convergence de l'évaluation théorique et de l'évaluation par simulation de la solution x_o^* à chaque itération de l'enrichissement de modèle. Au début, les évaluations sont éloignées, ce qui s'explique par le fait que les temps d'attente sont nuls dans le problème P_o. Progressivement, ces temps d'attente sont modifiés, ce qui conduit P_o à produire différentes solutions optimales (qui ne sont pas toujours très bonnes en pratique, ce qui explique les variations des premières itérations). Finalement, les évaluations convergent et l'heuristique se stabilise sur une bonne solution. La conver-

gence semble atteinte rapidement ici, cependant nous avons observé des cas où l'heuristique oscille entre plusieurs bonnes solutions [Bach07b], rendant ainsi la convergence lente.

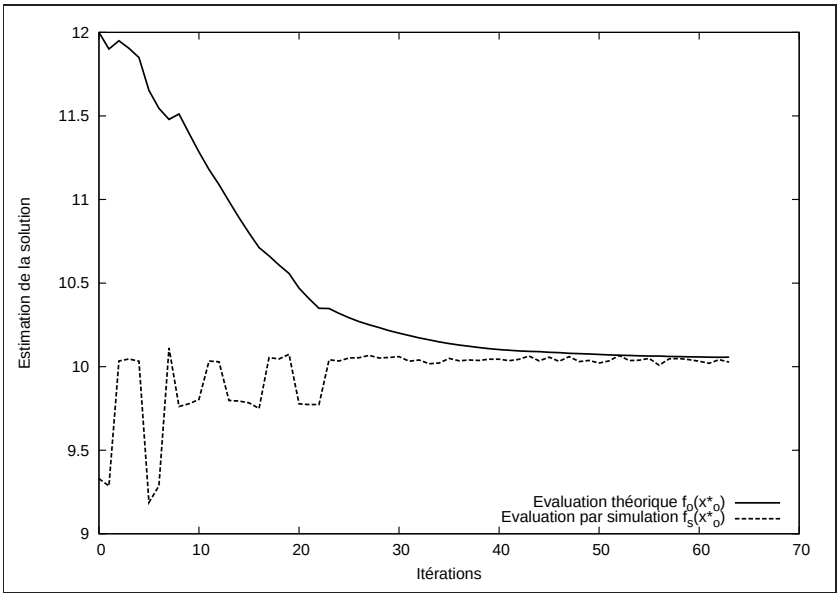

Figure 3.6: Evolution de l'évaluation des solutions au cours d'un enrichissement de modèle.

3.3. Simulation de flux discrets dans un réseau

Au cours de cette étude, un simulateur à événements discrets a été utilisé pour évaluer le système de tournées de bus. Les usagers et les bus sont représentés par des objets se déplaçant dans un graphe décrivant les rues de la ville. Plus généralement, il existe de nombreux problèmes d'optimisation liés à des réseaux routiers, des réseaux de communication... qui peuvent être modélisés sous la forme de flux discrets dans un graphe. Il paraît donc nécessaire de disposer d'un simulateur suffisamment générique pour considérer de nombreux modèles de flux discrets. Une intégration simple et efficace des codes d'optimisation et de simulation s'impose également, afin de permettre une souplesse dans le couplage entre optimisation et simulation tout en limitant les pertes de performance à l'exécution. Certains objets du modèle de simulation peuvent aussi faire appel à des algorithmes d'optimisation: e.g. un usager recherche un plus court chemin entre deux noeuds du réseau.

Un *framework* générique (au sens large) de simulation à événements discrets pour des flux discrets dans un réseau a donc été développé. Il a été conçu en C++ pour le moteur de simulation, et en Java pour la partie visuelle. Il est intégré à la bibliothèque B++ Library disposant d'algorithmes génériques pour la recherche opérationnelle (cf. chapitre 5), et d'une interface évoluée avec Java (cf. section 1.5) permettant une représentation visuelle et interactive du modèle de simulation.

L'architecture du simulateur repose sur une représentation des événements sous forme d'appels asynchrones à des méthodes d'objets de simulation. Sur cette base, un modèle générique de flux discrets dans un réseau est proposé, à partir duquel le modèle de tournées de bus est défini.

3.3.1. Architecture du simulateur

La classe `Simulator`, qui implémente le moteur du simulateur, représente un modèle de simulation (cf. figure 3.7). Tout objet manipulé au cours d'une simulation (i.e. entité du modèle ou événement) appartient à la classe `Object`. Elle implémente notamment un mécanisme permettant de déléguer l'allocation mémoire d'un type d'objets à un *pool*, i.e. un objet chargé d'optimiser l'allocation dynamique d'objets d'une même classe.

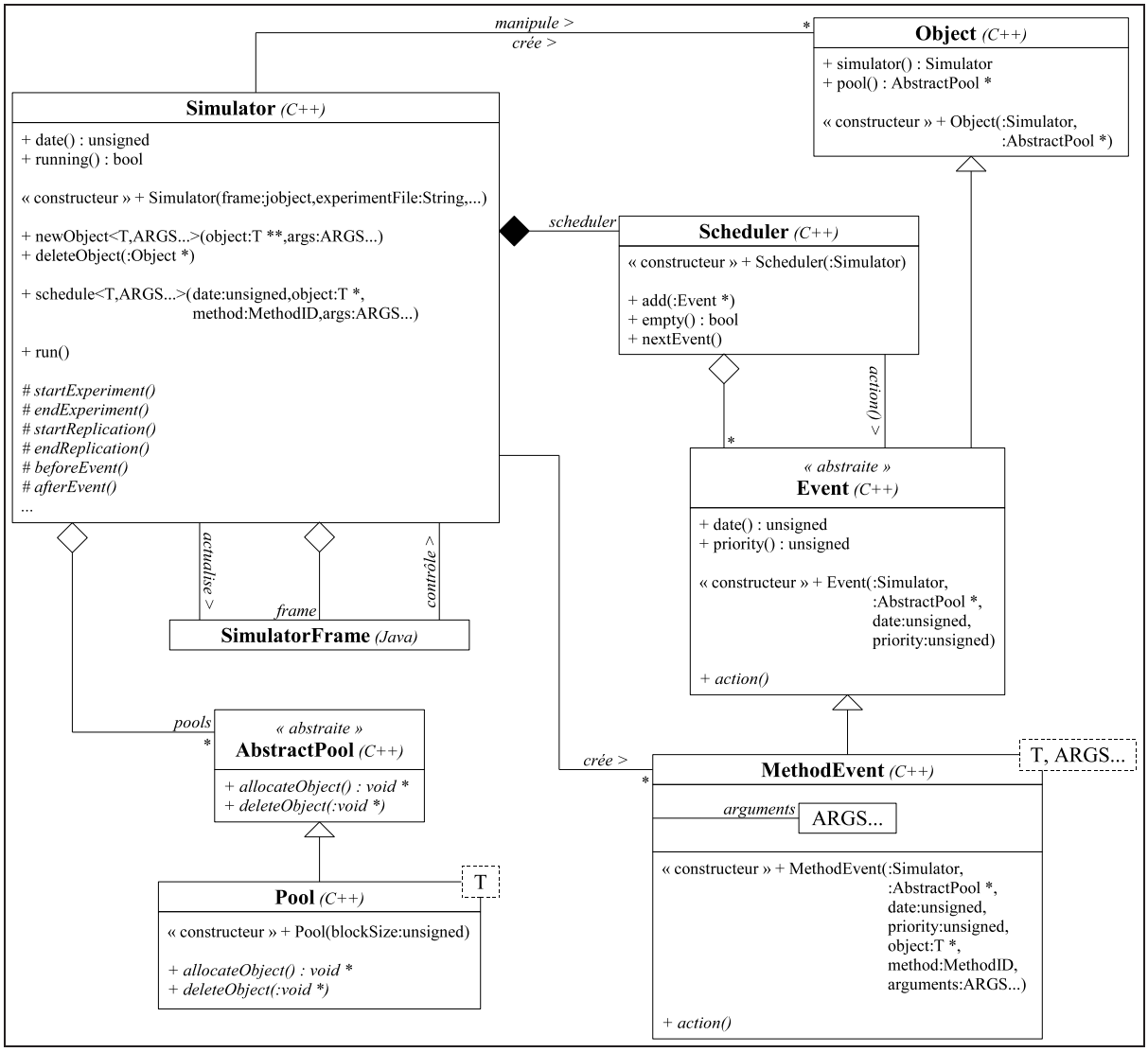

Figure 3.7: Architecture du simulateur.

La planification des événements (i.e. l'ordonnancement des événements et l'avancée du temps de simulation) est effectuée par un objet `Scheduler` qui manipule les événements sous la forme d'objets de la classe `Event`. Le modèle de simulation peut aussi être associé à un composant Java de la classe `SimulatorFrame` afin de disposer d'une interface graphique permettant de piloter l'avancée d'une simulation et d'inspecter l'état des différents objets de la simulation en cours.

3.3.1.1. Appel de méthode asynchrone

Pour simplifier la création d'événements, la sous-classe `MethodEvent` modélise un événement comme l'appel à une méthode d'un objet du modèle de simulation. Une difficulté de conception réside dans l'aspect asynchrone: l'appel à la méthode associée à l'événement se produit lorsque le temps de simulation coïncide avec la date de l'événement. Par exemple, considérons la planification de l'arrivée d'un bus à un arrêt, qui s'écrit de la manière suivante avec notre solution.

```
Bus *  bus = [...];
Spot * arret = [...];

simulator().schedule(dateArrivee,bus,method_id(Bus,arrive),arret);
```

La création de l'événement est prise en charge par le simulateur, via la méthode `schedule` qui reçoit les éléments suivants: la date de réalisation, l'objet qui subira l'événement, la méthode de cet objet qui sera appelée (via la macro `method_id` qui abstrait la manière d'identifier la méthode), et enfin les arguments de cette méthode. Le simulateur construit alors un objet `MethodEvent` mémorisant tous ces éléments et encapsulant dans une méthode `action` le code correspondant à l'exécution de la méthode. La classe `MethodEvent` est générique avec comme paramètres le type de l'objet concerné par l'événement, et les types des arguments qui seront fournis à la méthode de cet objet.

Cette solution a été retenue car la norme C++03, la plus récente à l'époque, ne dispose pas nativement d'un mécanisme d'appel asynchrone. A noter aussi que cette norme de permet pas la définition d'une liste variable de paramètres dans un composant générique (cf. *variadic templates* en C++11). Des techniques de métaprogrammation par macro-instructions ont donc été nécessaires pour générer différentes versions de la classe `MethodEvent` et de la méthode `schedule` selon le nombre d'arguments nécessaires à l'événement.

Les expressions lambda, introduite par la norme C++11, simplifieraient grandement notre conception. Une expression lambda permet d'encapsuler à la volée du code dans un objet (ou dans une fonction pour les cas les plus simples), et de différer ainsi l'exécution de ce code. Lors de l'encapsulation, les variables manipulées par le code sont capturées, i.e. mémorisées, pour un usage futur. L'exemple de planification précédent s'écrirait alors comme suit [11].

```
Bus *  bus = [...];
Spot * arret = [...];

simulator().schedule(dateArrivee,[=] () { bus->arrive(arret); });
```

[11] `http://en.cppreference.com/w/cpp/language/lambda`

La méthode `schedule` et la classe `MethodEvent` n'auraient alors plus qu'un seul paramètre, le type de l'expression lambda, au lieu de multiples paramètres.

```
template <class L> void schedule(unsigned date,L && lambda);
```

3.3.1.2. Indexation des classes et des méthodes

Pour plus de flexibilité, il est nécessaire de pouvoir ajouter des spécificités à certaines classes (ou méthodes) en fonction du modèle de simulation. Par exemple, dans le modèle de simulation des tournées de bus, les objets qui modélisent les usagers seront très nombreux, il est donc judicieux qu'ils soient gérés dans un *pool* d'objets, ce qui ne peut pas être généralisé à tout modèle qui manipulerait des usagers. La solution retenue consiste à utiliser un tableau, spécifique à chaque modèle de simulation (i.e. attribut de `Simulator`) et indexé sur l'identifiant (i.e. un numéro unique) de la classe, où chaque élément pointe sur le *pool* d'objets potentiellement associé à la classe. Par exemple, dans le code du modèle de simulation, on peut affecter un *pool* aux objets `Customer`.

```
pools[class_no(Customer)] = new Pool<Customer>();
```

Au moment de construire un nouvel objet (cf. méthode `newObject`), le simulateur vérifie si un *pool* existe pour lancer soit la construction normale, soit la construction par placement (i.e. pour laquelle la position en mémoire de l'objet est imposée [12], ici par le *pool* d'objets).

```
AbstractPool * pool = pools[class_no(T)];

if (pool) *object = new (pool->allocateObject()) T([...]);
else *object = new T([...]);
```

L'indexation des classes et des méthodes est réalisée par métaprogrammation [13], et l'accès aux index est facilité par les macros `class_no` et `method_no`. L'indexation des méthodes permet notamment de leur affecter un niveau de priorité utile pour déterminer l'ordre d'exécution d'événements programmés à la même date. Par exemple, on peut souhaiter qu'en cas d'événements simultanés d'arrivée d'un usager (e.g. méthode `Customer::arrive`) et de départ d'un bus (e.g. méthode `Bus::move`) sur un noeud, l'usager puisse monter dans le bus (i.e. une priorité plus grande pour l'événement d'arrivée).

```
priorities[class_no(Bus)][method_no(Bus,move)] = 10;
priorities[class_no(Customer)][method_no(Customer,arrive)] = 20;
```

3.3.2. Modèle générique de flux

La classe abstraite `PhysicalObject` représente tout objet qui peut avoir une représentation visuelle dans le simulateur (cf. figure 3.8). De tels objets sont dits physiques ici, même s'ils peuvent représenter des objets abstraits (e.g. des paquets dans un réseau de communication). Le modèle de simulation est un objet physique qui représente le système complet de l'étude. Il peut être composé d'autres objets physiques, ses enfants, qui peuvent être mobiles ou non. Tout objet physique peut avoir des enfants,

[12] `http://en.cppreference.com/w/cpp/language/new`
[13] Module *Standard/Class* de la B++ Library: `http://www.nawouak.net/?doc=bpp_library+ch=modules`

ce qui forme une structure hiérarchique du système. L'objet contenant les enfants est le parent de ces objets, et un objet mobile peut bouger d'un parent à un autre. Un objet physique peut être "creux", i.e. les enfants sont situés à l'intérieur de l'objet (e.g. les passagers d'un bus), ou "superficiel", i.e. les enfants sont situés sur l'objet (e.g. les usagers à un arrêt de bus).

Un objet physique peut être associé à une vue, i.e. un objet Java de la classe `VisualComponent`, qui gère sa représentation visuelle. Cette dernière est localisée à l'aide des méthodes `relativeX` et `relativeY` qui renvoient la position de l'objet relativement à la position de son parent (ces méthodes sont abstraites et peuvent être redéfinies, en fonction de la manière de bouger de l'objet). Les méthodes `absoluteX` et `absoluteY` donnent la position de l'objet dans le plan général du système.

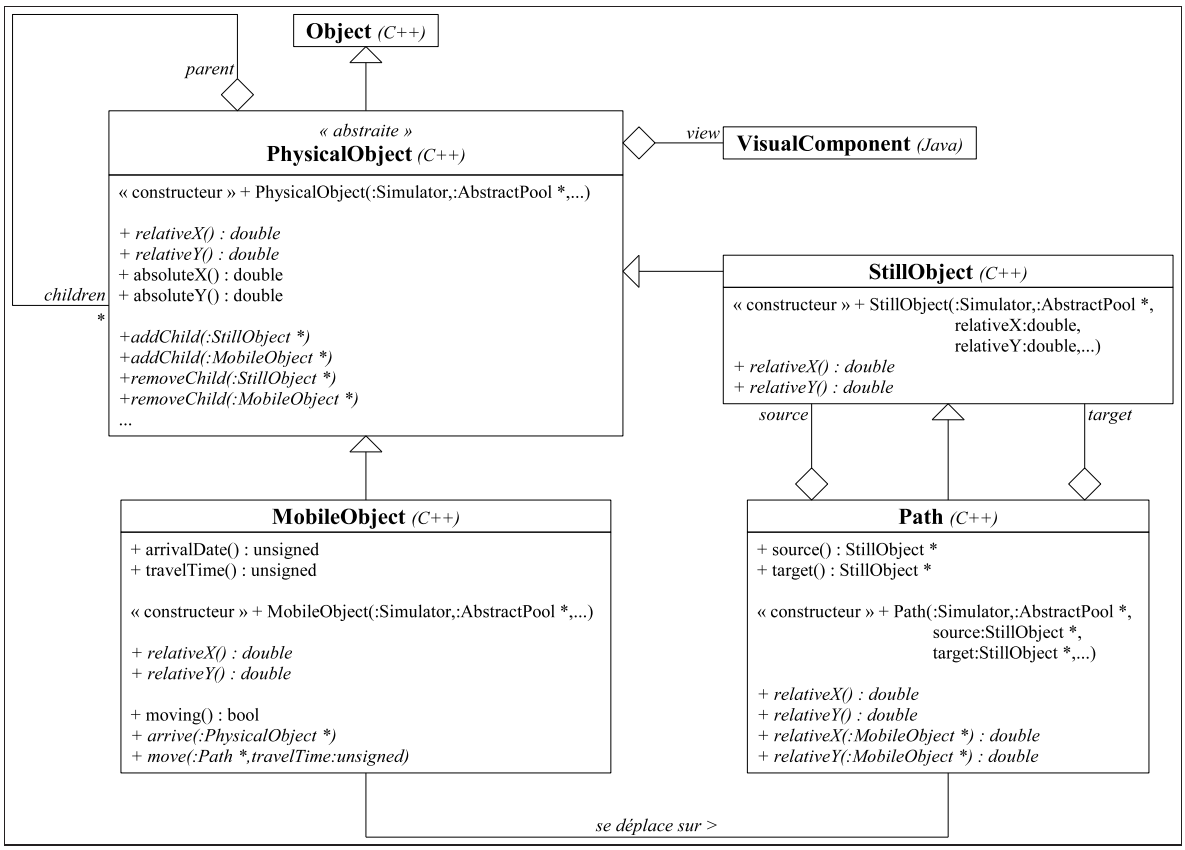

Figure 3.8: Modèle générique de flux discrets.

Les objets immobiles (classe `StillObject`) sont distingués des objets mobiles (classe `MobileObject`) uniquement pour des raisons d'efficacité: la position de leur vue n'a pas besoin d'être mise à jour à chaque avancée du temps de simulation. Les objets mobiles bougent d'un parent à un autre en suivant un chemin (classe `Path`). Un déplacement est initié par un événement d'appel à la méthode `move` de l'objet mobile, en indiquant quel chemin il va emprunter et dans combien de temps il arrivera à destination.

Voici un extrait de la méthode `move` qui déplace un objet le long d'un chemin (qui devient son parent) et planifie l'arrivée de l'objet à la destination du chemin.

```
void MobileObject::move(Path * path,unsigned travelTime) {
 this->parent->removeChid(this);
 path->addChild(this);

 this->arrivalDate = simulator().date();
 this->travelTime = travelTime;

 simulator().schedule(simulator().date()+travelTime,this,
                      method_id(MobileObject,arrive),path->target());
}
```

Les méthodes `relativeX` et `relativeY` sont redéfinies dans `MobileObject` afin de fournir une estimation de la position de l'objet lorsqu'il se déplace. Un objet mobile possède aussi des propriétés spécifiques à son mouvement comme la date d'arrivée sur son parent et le temps du trajet en cours. La méthode `arrive` est appelée quand l'objet arrive à destination: l'objet quitte le chemin pour se placer sur l'objet destination.

Les extrémités d'un chemin sont représentées par des objets physiques. Les méthodes `relativeX` et `relativeY` de la classe `Path` permettent aussi d'estimer la position d'un objet sur le chemin. Par défaut, un chemin est un segment de droite, mais la classe `Path` peut être étendue pour modéliser d'autres formes de chemin, en redéfinissant les méthodes `relativeX` et `relativeY`.

3.3.3. Application aux tournées de bus

Le modèle de tournées de bus s'appuie sur le modèle générique de flux, en réutilisant les classes déjà développées, soit par héritage afin de les spécialiser, soit par composition si les classes suffisent telles quelles. Lors de l'héritage, certaines méthodes seront redéfinies, généralement par complément, i.e. un appel à la version mère est présent dans le code de la redéfinition.

Le modèle de tournées de bus est représenté par la classe `Model` qui spécialise la classe `Simulator` du modèle générique (cf. figure 3.9). Certaines méthodes sont redéfinies, comme `startReplication` qui prépare le modèle avant le lancement d'une réplication de simulation: les objets initiaux de la simulation sont créés (e.g. le réseau routier, les bus, les flux d'usagers...), et des événements sont planifiés (e.g. les premiers mouvements des bus, la création des premiers usagers...). Au cours de la simulation, des données statistiques seront collectées (ici à l'aide d'objets de la classe `VarianceCollector` qui mesurent à la fois une moyenne et une variance): le temps moyen de trajet t_d des usagers de la demande d (cf. `effectiveTime` dans `CustomerTravel`), et le temps moyen d'attente w_i à l'arrêt i (cf. `waitingTime` dans `Spot`). Elles seront mémorisées à la fin de chaque réplication (cf. méthode `endReplication`) et synthétisées à la fin de l'expérience (cf. méthode `endExperiment`).

Le réseau est représenté par des noeuds (i.e. des objets de la classe `Spot` dérivée de `StillObject`) et des chemins (i.e. des objets de la classe `Path`). Les bus sont des objets mobiles creux, modélisés par la classe `Bus`, qui possèdent notamment une capacité, une vitesse de référence, et un trajet (i.e. une liste de chemins) décrivant la tournée à suivre. La méthode `arrive` est redéfinie par complément: la version de `MobileObject` est d'abord appelée avant de déposer certains passagers et d'en faire monter d'autres qui attendaient à l'arrêt.

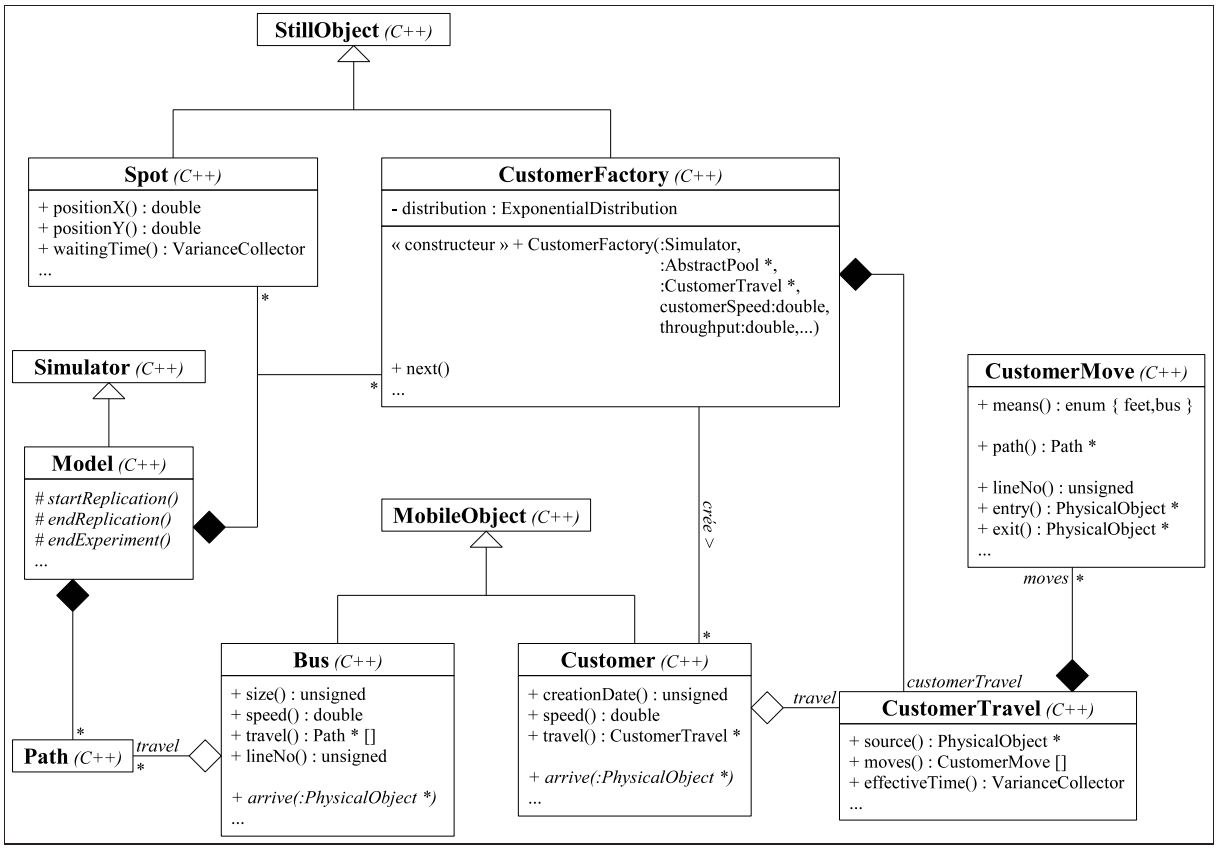

Figure 3.9: Modèle de tournées de bus.

Les usagers sont des objets mobiles superficiels, modélisés par la classe `Customer`, qui possèdent une vitesse de référence pour la marche à pied, et un trajet (classe `CustomerTravel`). Le trajet consiste en une liste de mouvements (classe `CustomerMove`) caractérisés par un chemin et un mode de transport (à pied ou en bus), et dans le cas d'un déplacement en bus, par des renseignements sur la ligne à utiliser (son numéro et les arrêts où monter et descendre). La méthode `arrive` est également redéfinie par complément: après son arrivée, l'usager doit soit attendre un bus, soit emprunter à pieds le prochain chemin de son trajet. Un flux d'usagers d'une demande d est représenté par un objet de la classe `CustomerFactory` qui émet des événements de création d'usager suivant une loi de probabilité prédéfinie.

3.4. Conclusion

Le couplage optimisation-simulation traditionnel fournit de bonnes solutions en pratique, mais peut nécessiter un long temps d'exécution à cause de nombreuses évaluations par simulation. A l'opposé, des méthodes d'optimisation peuvent résoudre efficacement des modèles mathématiques simplifiés, au risque de produire de mauvaises solutions pratiques. L'approche par enrichissement de modèle introduite ici propose d'améliorer la formulation du modèle mathématique, à partir d'évaluations par simulation, afin que la méthode d'optimisation produise de meilleures solutions pratiques.

Le principe de l'enrichissement de modèle a été formulé comme un problème d'optimisation qui semble très difficile à résoudre de manière générique. Nous avons néanmoins proposé une première approche heuristique avec des questions encore ouvertes, comme le choix de la fonction h qui modifie les contraintes du modèle théorique et doit garantir la convergence de l'algorithme. L'implémentation de l'heuristique pour un problème de tournées de bus a révélé une convergence entre évaluation théorique et évaluation pratique (estimée par simulation) qui montre le potentiel de la méthode d'enrichissement de modèle. Cette expérience apporte avant tout une nouvelle manière de penser le couplage entre optimisation et simulation.

L'enrichissement de modèle a consisté ici à modifier les contraintes du modèle mathématique en changeant les coefficients de contraintes linéaires. Il serait intéressant d'expérimenter l'approche sur un problème où certaines solutions du problème d'optimisation théorique ne sont pas réalisables en pratique (d'après l'évaluation par simulation), ce qui obligerait à ajouter de nouvelles contraintes dans le modèle mathématique théorique.

CHAPITRE 4
SIMULATION D'ECOSYSTÈMES PRAIRIAUX

Anticiper les impacts d'un changement climatique est un enjeu majeur. En particulier, il est important d'identifier les conséquences sur la production des prairies, et d'élaborer des stratégies pour garantir leur rendement. L'unité de recherche UREP de l'INRA développe deux modèles complémentaires sur le fonctionnement des prairies: GEMINI, qui simule la croissance d'une population de plantes, et PaSim qui simule l'évolution d'une parcelle. Ces modèles intègrent des actions humaines de gestion, comme la fauche, la fertilisation ou le pâturage. Il s'agit donc de trouver une bonne stratégie de gestion de la prairie qui minimise sa vulnérabilité et assure ainsi son rendement.

Ces modèles de simulation par intégration numérique sont composés de sous-modèles, avec un grand nombre d'équations différentielles ordinaires. Pour gérer la complexité et la modularité de tels modèles, une plateforme générique pour la simulation par intégration numérique a été développée. Minimiser la vulnérabilité d'un système nécessite un nombre prohibitif de simulations. Afin de réduire ce nombre, une approche consiste à construire une approximation du modèle de simulation (un modèle simplifié par sortie du modèle) à partir des données d'une analyse de sensibilité, et à minimiser ensuite la vulnérabilité sur la base des évaluations des modèles simplifiés à l'aide d'algorithmes d'optimisation traditionnels (principalement des métaheuristiques).

4.1. Plateforme de simulation par intégration numérique

Un modèle comme GEMINI (*Grassland Ecosystem Model with INdividual-centered Interactions*) [14] résulte de nombreux travaux de modélisation mécaniste des phénomènes liés aux plantes et aux sols, des connaissances qui évoluent perpétuellement et où plusieurs approches peuvent coexister. Il est donc nécessaire de disposer d'une structure modulaire, où le modèle est considéré comme un assemblage de sous-modèles pouvant être remplacés simplement. Des sous-modèles peuvent aussi apparaître ou disparaître en cours de simulation (e.g. naissance/dépérissement d'une feuille sur une plante).

Pour le modèle GEMINI et d'autres modèles à venir, un *framework* générique pour la conception et la simulation de modèles à intégration numérique a été développé: UNIF (*Unified Numerical Integration Framework*). Les modèles qu'il manipule sont exprimés sous la forme d'équations différentielles ordinaires du premier ordre, dont les fonctions sont dépendantes du temps [Cell05]. Sa conception objet offre une structure flexible qui facilite notamment la spécialisation de modèles, l'activation/désactivation de sous-modèles, et le couplage de modèles.

[14] `https://www1.clermont.inra.fr/urep/modeles/gemini.htm`

Il peut être nécessaire de compléter ces modèles continus avec des événements discrets (e.g. l'apparition d'une feuille si certaines conditions sont remplies). Le *framework* permet de déclencher des événements entre chaque pas d'intégration (i.e. chaque avancée du temps de simulation, où l'état du système est mis à jour à partir des équations différentielles). Afin d'assurer une certaine robustesse du simulateur, un contrôle des variables du modèle (e.g. contrôle de négativité ou de divergence) peut être demandé. Dans un souci d'efficacité et/ou de précision, il est possible d'implémenter différentes méthodes d'intégration (i.e. méthodes d'approximation comme Euler, Runge-Kutta... [Pres92] permettant d'estimer l'état du système au prochain pas de temps).

4.1.1. Modèle GEMINI

GEMINI modélise l'évolution de populations de plantes de prairie en compétition pour la lumière et les ressources du sol. Elles sont soumises à des événements externes liés à l'activité humaine comme la fertilisation, le pâturage d'animaux, la fauche... et à la météo (obtenue à partir d'archives de mesures ou de prédictions basées sur des scénarios de changement climatique). GEMINI est le couplage de deux modèles, Soilopt et Canopt, développés par l'équipe UREP de l'INRA.

Soilopt est développé depuis 1998. Il modélise un sol et ses populations microbiennes sous la forme de compartiments (9 organiques et 3 minéraux) représentant les stocks de différentes matières dans le sol. Les flux de carbone et d'azote échangés entre ces compartiments sont régis par des équations différentielles [Lois98]. Ce modèle intègre le *priming effect*: la matière fraîche résultant de la mort d'une plante permet aux microbes d'obtenir plus d'énergie pour décomposer la matière organique du sol [Font05].

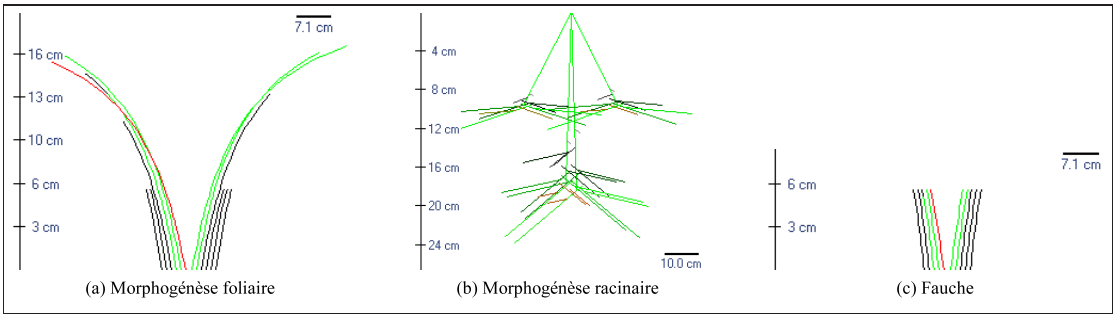

Figure 4.1: Représentation visuelle d'une plante dans UNIF.

Canopt est développé depuis 1996. Il modélise une ou plusieurs populations de plantes en compétition pour des ressources. Le modèle simule un individu moyen pour chaque population, avec des compartiments pour différentes substances de la plante [Sous00a, Sous00b]. La morphogénèse foliaire est détaillée: la croissance et la sénescence de chaque feuille sont considérées, ainsi que sa géométrie afin d'obtenir un modèle précis de la photosynthèse dans la compétition pour la lumière (figure 4.1a). La morphogénèse racinaire est aussi détaillée (figure 4.1b), permettant une estimation plus précise des prélèvements de la plante dans le sol.

Depuis 2004, les modèles Canopt et Soilopt sont couplés pour former GEMINI. La représentation simpliste d'une plante dans Soilopt a été remplacée par Canopt, et inversement, Soilopt a remplacé le modèle du sol de Canopt [Sous12]. Des sous-modèles représentant l'activité humaine sur les populations de plantes ont été introduits, comme la fauche qui consiste à prélever une partie des feuilles des plantes avec une hauteur de coupe donnée (figure 4.1c).

4.1.2. Modélisation par équations différentielles

Un modèle est constitué d'un ensemble d'équations différentielles. Prenons le cas simple du compartiment d'azote minéral, tel qu'il est défini dans le modèle Soilopt. Il représente, à l'instant t, une quantité $N_{min}(t)$ d'azote minéral qui peut varier en fonction de différents facteurs (figure 4.2):

- une fertilisation du sol, qui apporte une quantité $N_{in}(t)$;
- une minéralisation du sol, qui libère une quantité $DM(t)$;
- une immobilisation par des micro-organismes, qui prélèvent une quantité $DI(t)$;
- une dénitrification par des bactéries, qui prélèvent une quantité $N_{den}(t)$;
- un lessivage du sol, qui emporte une quantité $N_{leach}(t)$;
- une absorption par les plantes, qui prélèvent une quantité $N_{upt}(t)$.

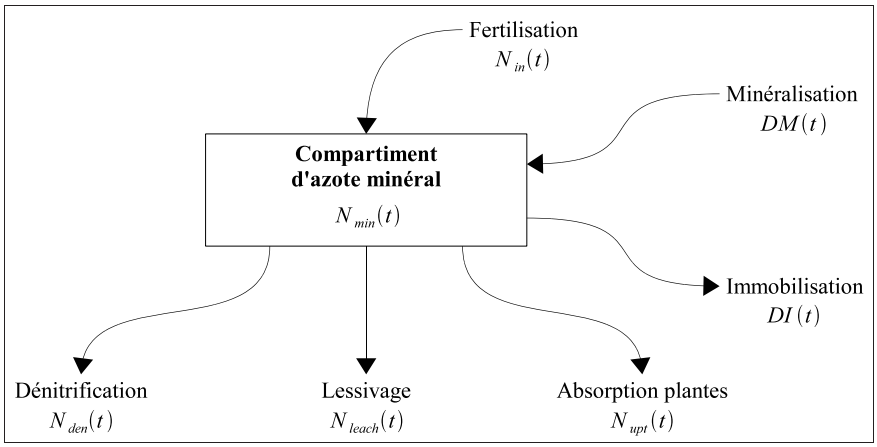

Figure 4.2: Flux du compartiment d'azote minéral.

Supposons de manière simpliste que la quantité absorbée par les plantes est un pourcentage $D_{plant}(t)$ (calculé à partir des besoins et de la densité de la population de plantes) de l'azote disponible: $N_{upt}(t) = D_{plant}(t)\, N_{min}(t)$. La quantité $N_{min}(t)$ du compartiment d'azote minéral peut être modélisée par l'équation différentielle du premier ordre suivante.

$$\frac{\mathrm{d}N_{min}}{\mathrm{d}t}(t) = N_{in}(t) - N_{den}(t) - N_{leach}(t) - D_{plant}(t)\, N_{min}(t) + DM(t) - DI(t) \qquad (4.1)$$

Les variables de cette équation peuvent être de différentes natures: il peut s'agir de paramètres du modèle, constants tout au long de la simulation, ou variables (les valeurs étant lues à partir d'un fichier); ou de variables d'intégration (i.e. des variables dont la valeur est définie par une équation différentielle).

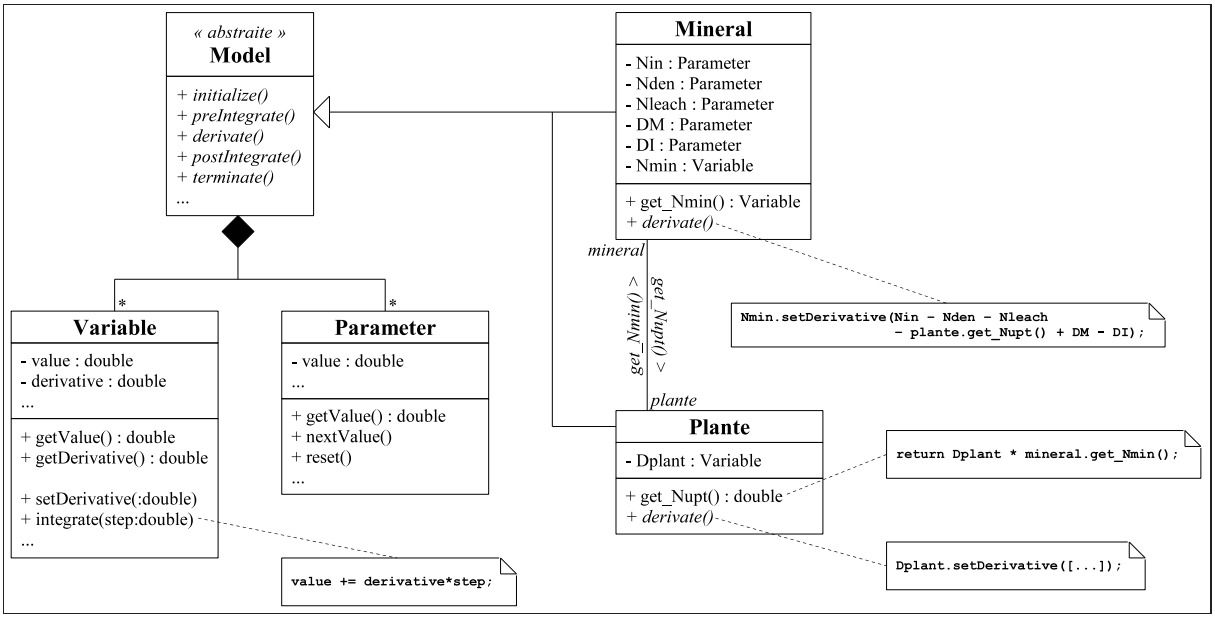

Figure 4.3: Définition d'un modèle dans UNIF.

Pour définir un modèle avec UNIF, la classe abstraite `Model` doit être étendue: dans notre exemple, la classe `Mineral` représente le compartiment d'azote minéral, et la classe `Plante` la population de plantes (cf. figure 4.3). Des paramètres (cf. classe `Parameter`) et des variables d'intégration (cf. classe `Variable`) sont ajoutées à un modèle sous la forme d'attributs, et les formules des équations du modèle apparaissent dans la redéfinition de la méthode `derivate` en décrivant le calcul des dérivées de toutes les variables d'intégration du modèle (nécessaires au processus de simulation).

4.1.3. Méthodes d'approximation et simulation

La simulation par intégration numérique d'un modèle consiste à déterminer les valeurs des variables qui décrivent l'état du système à chaque temps T_k, en intégrant les formules des équations différentielles. Dans notre exemple, la simulation va déterminer chaque valeur ${N_{min}}^k = N_{min}(T_k)$ du compartiment d'azote minéral, sachant qu'au temps initial T_0, ${N_{min}}^0$ est fixée. Théoriquement, la valeur ${N_{min}}^k$ se calcule de la manière suivante.

$${N_{min}}^k = \int_{T_0}^{T_k} \frac{\mathrm{d}N_{min}}{\mathrm{d}t}(t)\mathrm{d}t + {N_{min}}^0$$

Mais il est rarement possible d'obtenir une formulation permettant un calcul direct de cette valeur. Des méthodes d'approximation comme Euler ou Runge-Kutta sont donc utilisées [Pres92]. Leur principe repose sur une suite mathématique, où Δ représente le pas de temps de la simulation.

$$N_{min}{}^{k+1} = N_{min}{}^{k} + \Delta\, d^k \tag{4.2}$$

La valeur d^k est une approximation de la dérivée de N_{min} à l'instant T_k. Notons $N'_{min}{}^{k}$ le calcul de la dérivée de N_{min} à partir des valeurs des variables du modèle à l'instant T_k, en appliquant la formule de l'équation différentielle 4.1.

$$N'_{min}{}^{k} = N_{in}{}^{k} - N_{den}{}^{k} - N_{leach}{}^{k} - D_{plant}{}^{k}\, N_{min}{}^{k} + DM^k - DI^k \tag{4.3}$$

En supposant la dérivée constante pendant un pas de temps, la méthode d'Euler propose $d^k = N'_{min}{}^{k}$. L'erreur d'approximation de la méthode pouvant être assez grande, la méthode de Runge-Kutta d'ordre 4 est souvent préférée. L'approximation de la dérivée d^k est alors une somme pondérée de quatre dérivées $d_i{}^k$ évaluées à partir de différents états du système [Pres92]. La première dérivée est calculée au temps T_k, les deux suivantes au temps $T_k + \frac{\Delta}{2}$, et la dernière au temps $T_k + \Delta$.

$$d^k = \frac{d_1{}^k + 2d_2{}^k + 2d_3{}^k + d_4{}^k}{6}$$

$$\begin{aligned}
d_1{}^k &= N'_{min}{}^{k} & N^1_{min}{}^{k} &= N_{min}{}^{k} + \tfrac{\Delta}{2} d_1{}^k \\
d_2{}^k &= N^{1'}_{min}{}^{k} & N^2_{min}{}^{k} &= N_{min}{}^{k} + \tfrac{\Delta}{2} d_2{}^k \\
d_3{}^k &= N^{2'}_{min}{}^{k} & N^3_{min}{}^{k} &= N_{min}{}^{k} + \Delta d_3{}^k \\
d_4{}^k &= N^{3'}_{min}{}^{k}
\end{aligned}$$

L'état du système, nécessaire au calcul de chaque dérivée $d_i{}^k$, découle de la formule 4.2 appliquée avec la dérivée $d_{i-1}{}^k$. Par exemple, $N^2_{min}{}^{k}$ est la valeur de N_{min} au temps $T_k + \frac{\Delta}{2}$ en appliquant la dérivée $d_2{}^k$. $N^{2'}_{min}{}^{k}$ est la dérivée calculée à partir de cet état, de manière similaire à $N'_{min}{}^{k}$ (cf. formule 4.3).

Quelle que soit la méthode d'approximation, l'évolution des variables d'intégration se réduit au calcul de leur dérivée. Ce dernier est donc implémenté dans la méthode `derivate` de chaque modèle, ce qui permet d'écrire simplement une méthode d'approximation générique (i.e. indépendante du modèle manipulé). Plus globalement, le processus de simulation est décomposé en cinq étapes, identifiées par des méthodes dans la classe `Model` (cf. figure 4.3) qui sont exécutées automatiquement pour chaque modèle par le simulateur.

- Initialisation: Au début de la simulation, le simulateur appelle la méthode `initialize` du modèle. Elle fixe ses variables d'intégration pour définir l'état initial du système simulé.

- Pré-intégration: Avant chaque pas d'intégration, le simulateur appelle la méthode `preIntegrate` du modèle. L'état du système peut être analysé pour déclencher des événements discrets, e.g. l'apparition d'une nouvelle feuille dans le modèle de plante.
- Intégration: A chaque pas de temps, le simulateur exécute la méthode d'approximation sur le modèle, afin de mettre à jour ses variables d'intégration, en appelant une ou plusieurs fois la méthode `derivate` du modèle avant d'exécuter la méthode `integrate` de chaque variable.
- Post-intégration: Après chaque pas d'intégration, le simulateur appelle la méthode `postIntegrate` du modèle, pour les mêmes raisons que la pré-intégration.
- Finalisation: A la fin de la simulation, le simulateur appelle la méthode `terminate` du modèle. Les données collectées au cours de la simulation peuvent être récupérées et traitées.

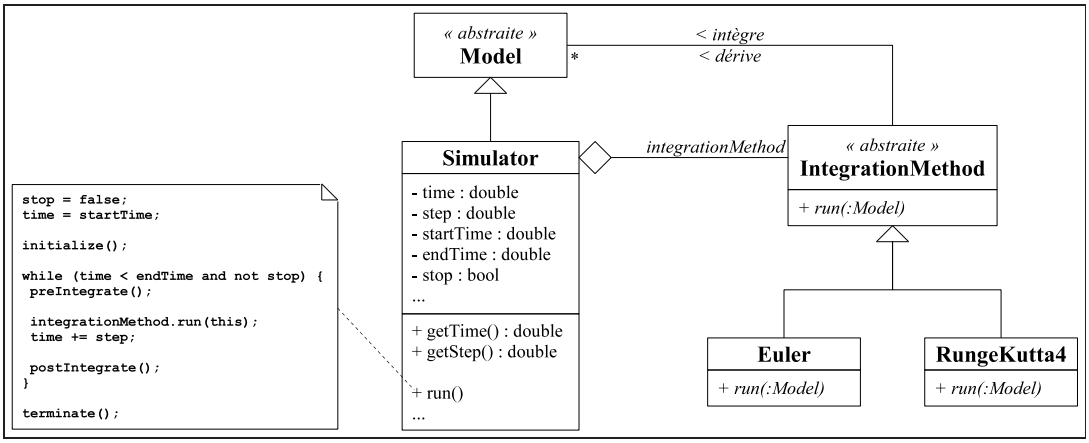

Figure 4.4: Modèle du simulateur dans UNIF.

Le simulateur est modélisé par la classe `Simulator` qui hérite de `Model`. Il est donc considéré comme un modèle, avec une méthode spécifique `run` permettant de lancer une simulation en contrôlant les cinq étapes présentées précédemment (cf. figure 4.4). L'application du patron *stratégie* [Gamm95] (cf. classe `IntegrationMethod`) permet au simulateur d'utiliser indifféremment les méthodes d'Euler ou de Runge-Kutta, ou tout autre technique d'approximation.

4.1.4. Modèles couplés

Un modèle est souvent l'assemblage de plusieurs modèles. Prenons le cas de GEMINI (cf. figure 4.5) qui est le couplage de deux modèles, Canopt et Soilopt, auquel s'ajoutent: un modèle environnemental (la météo) et des modèles de gestion (fertilisation, pâturage et fauche). Cette structure est statique, i.e. elle est fixée en début de simulation et ne change pas ensuite.

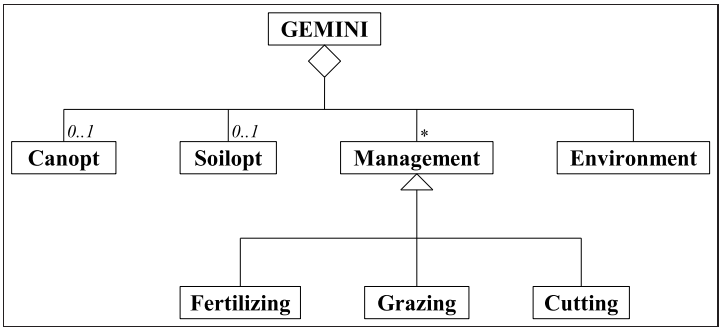

Figure 4.5: Structure du modèle GEMINI.

Le sous-modèle Canopt peut simuler différentes populations de plantes, et chaque modèle de plante peut avoir un modèle aérien et/ou racinaire (cf. figure 4.6). Ces modèles sont aussi composés respectivement de modèles de feuille et de racine, qui sont des objets pouvant apparaître ou disparaître pendant une simulation. Des modèles peuvent donc être créés et insérés en tant que sous-modèles d'un modèle en cours de simulation.

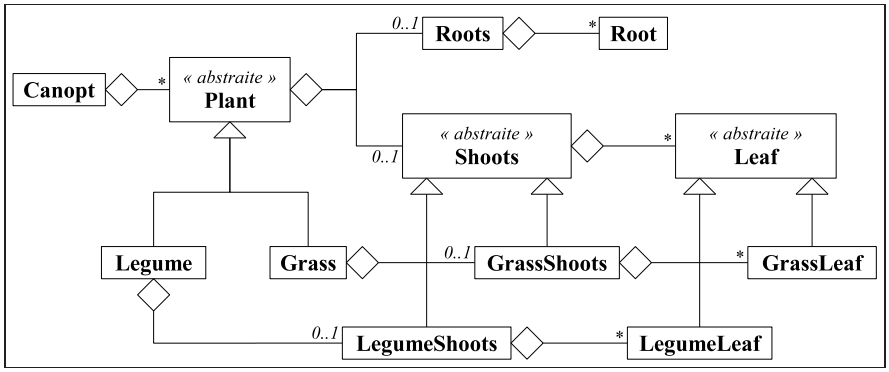

Figure 4.6: Structure du sous-modèle Canopt.

Canopt peut modéliser différentes espèces de plantes, à partir de deux familles: les graminées et les légumineuses. Les modèles de ces familles héritent d'un modèle abstrait de plante, et chaque espèce correspond à un paramétrage particulier de l'un des deux modèles: des mesures expérimentales permettent d'obtenir des données caractéristiques d'une espèce qui correspondent à des paramètres du modèle [Sous12].

Les modèles forment donc une arborescence dynamique. Le patron *Composite* [Gamm95] est souvent utilisé pour concevoir ce type de structure récursive (cf. figure 4.7a). Cependant, l'approche surcharge l'interface de la classe `Model` en mélangeant les aspects structure de données (i.e. la structure d'arbre) et les aspects simulation. Afin d'obtenir une séparation, le patron CRTP (*Curiously Recurring Template Pattern* [Copl96]) est utilisé, il s'agit d'une technique peu invasive permettant, entre autres, d'ajouter des fonctionnalités à une classe.

La classe générique `Tree` modélise un noeud d'un arbre et de manière récursive un arbre (cf. figure 4.7b). La classe `Tree` est paramétrée sur le type `T` de ses noeuds enfants et fournit les fonctionnalités pour les manipuler. A noter que `T` doit posséder les mêmes spécificités (en particulier la même interface) que `Tree`, ce qui est formalisé par le "concept" `TreeNode` (cf. chapitre 6). La classe `Tree<Model>` représente un arbre de modèles, donc pour qu'un modèle soit un arbre, la classe `Model` hérite de `Tree<Model>`. Dans cette structure récursive, un modèle est en charge d'appeler chacune des étapes de la simulation pour ses enfants, e.g. la méthode `derivate` d'un modèle doit exécuter la méthode `derivate` de chacun de ses enfants.

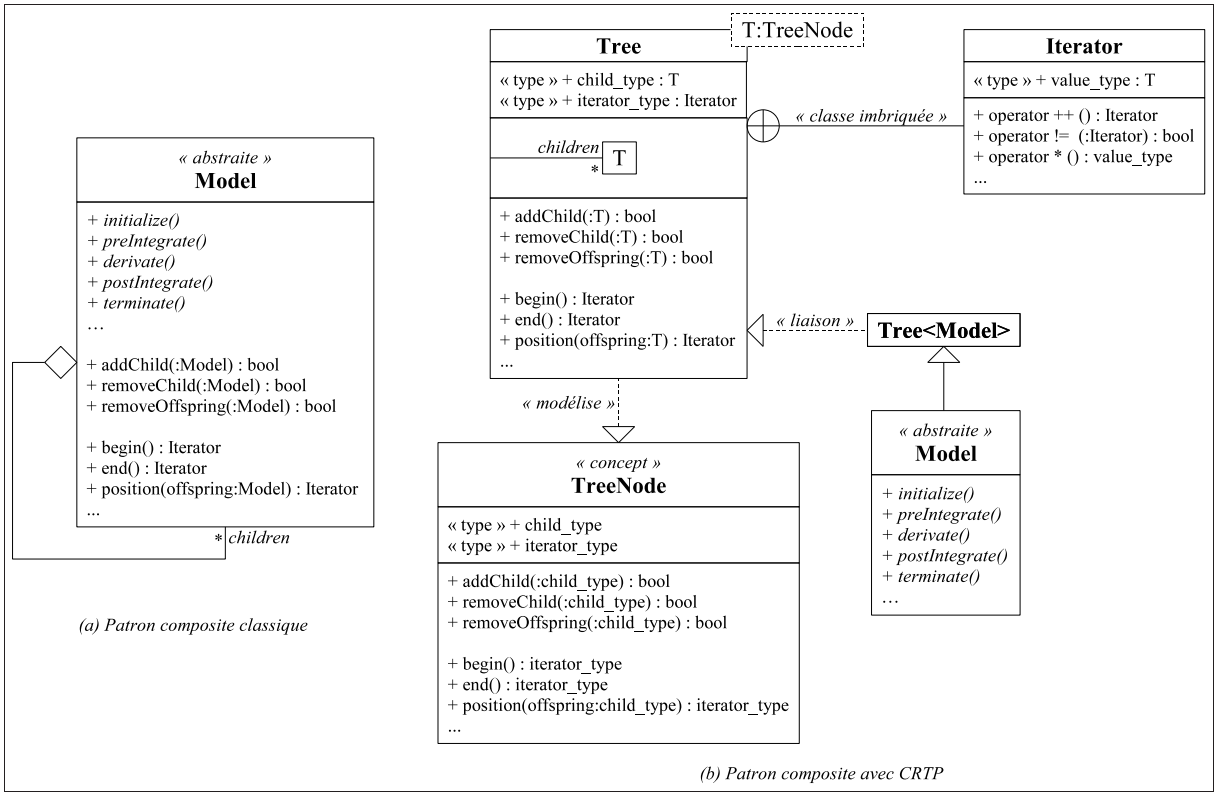

Figure 4.7: Composition de modèles avec UNIF.

L'ordre dans lequel les variables sont dérivées est défini manuellement dans chaque méthode `derivate`, ce qui peut provoquer des erreurs d'approximation dans une simulation. Par exemple, le couplage des modèles Canopt et Soilopt dans GEMINI crée des dépendances dans leurs équations différentielles qui obligent à être attentif à l'ordre d'appel des méthodes `derivate` pour ces deux modèles. Nous avons proposé des modifications de la plateforme UNIF afin d'identifier les dépendances entre modèles et automatiser leur ordre d'appel pour le calcul des dérivées [Gay06], mais une étude plus approfondie reste nécessaire afin d'obtenir une solution qui ne pénalise pas les temps de simulation, et gère aussi les dépendances au sein d'un modèle.

4.2. Minimisation de la vulnérabilité

La vulnérabilité d'un système peut être vue comme le degré de perturbation qu'il peut supporter avant d'être endommagé [Turn03]. Cette notion n'est que rarement observable directement et elle est relative à une situation de référence (afin de définir la notion de dommage sur le système). La vulnérabilité peut être estimée de manière qualitative ou quantitative à l'aide de différents indices [Lard12a]. L'analyse de vulnérabilité s'apparente à une forme d'analyse de sensibilité où un grand nombre de simulations du système est nécessaire. La recherche de paramètres du système qui minimiseraient sa vulnérabilité est encore plus coûteuse en simulations.

Nous avons proposé dans [Lard14] une méthodologie pour la minimisation de la vulnérabilité d'un système, sur la base d'une estimation quantitative, et en tentant de fournir un cadre générique vis-à-vis du modèle étudié et de l'indice de vulnérabilité choisi. Elle s'appuie sur l'approximation du modèle de simulation par des surfaces de réponse, afin de réduire le nombre initialement prohibitif de simulations.

Cette approche est appliquée au modèle PaSim (*Pasture Simulation*) [15], un modèle mécaniste et déterministe du fonctionnement d'une prairie, afin de rechercher la gestion optimale qui permettrait de minimiser la vulnérabilité du stock de matière organique du sol, élément clé dans l'atténuation des émissions de gaz à effet de serre, sous des conditions environnementales prévues par les scénarios les plus probables de changement climatique.

4.2.1. Modèle PaSim

PaSim modélise les flux de matières et d'énergie au sein d'une parcelle de prairie composée d'un sol, d'une végétation et d'herbivores, en réponse aux conditions climatiques et à une gestion déterminée de la parcelle [Ried98, Vuic07, Grau11]. Ce modèle prend notamment en compte des pratiques de fertilisation et d'exploitation de l'herbe par la fauche et/ou le pâturage. Cette gestion peut être planifiée à l'avance ou adaptée automatiquement par le modèle.

PaSim est un modèle à intégration numérique, implémenté en Fortran, et organisé en sous-modèles décrivant les mécanismes de la végétation, du climat, du sol et des herbivores (cf. figure 4.8). Le modèle s'appuie sur des données météorologiques horaires telles que la température, le vent, le rayonnement solaire, les précipitations... La politique de gestion de la parcelle est représentée par la classe `ManagementPolicy` qui peut réaliser séparément ou simultanément deux types de gestion: des règles prédéfinies dans le modèle (cf. classe `AutomaticManagement`) qui déclenchent automatiquement des interventions, ou la planification à des dates précises d'intervention (cf. classe `ScheduledManagement`).

[15] `https://www1.clermont.inra.fr/urep/modeles/pasim.htm`

Trois types d'interventions sont possibles: la fertilisation, en précisant la nature et la quantité d'engrais (cf. classe `Fertilizing`); la fauche (cf. classe `Cutting`); et le pâturage, en précisant la quantité d'animaux (cf. classe `Grazing`).

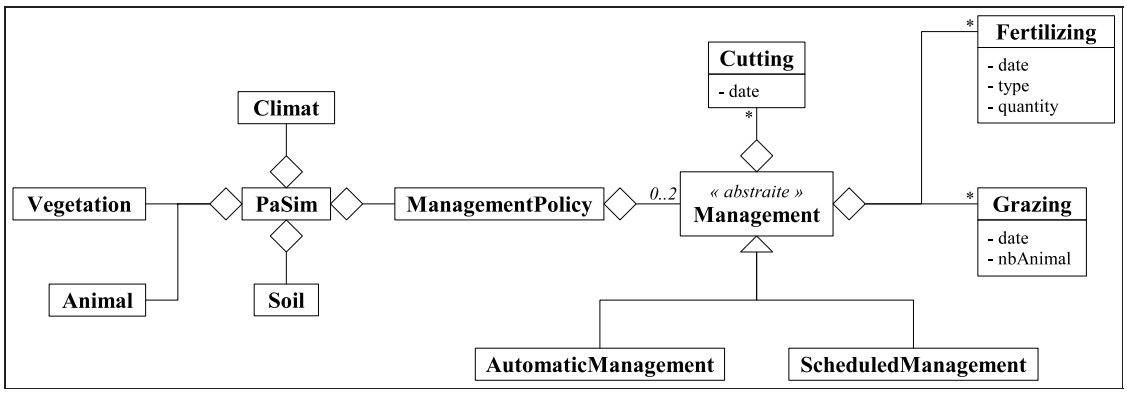

Figure 4.8: Structure du modèle PaSim.

4.2.2. Méthodologie

L'approche retenue se décompose en deux étapes (cf. figure 4.9): une analyse de vulnérabilité, sans adaptation du système (e.g. la politique de gestion de la parcelle n'est pas spécialement adaptée aux changements climatiques), suivie d'une minimisation de la vulnérabilité par l'adaptation du système (e.g. la politique de gestion est optimisée pour faire face aux changements climatiques). Cela permet d'estimer l'apport de la stratégie d'adaptation trouvée sur la vulnérabilité.

L'analyse de vulnérabilité se décompose en plusieurs phases. Un plan d'expériences, adapté à l'analyse de sensibilité, doit d'abord être établi afin de limiter la quantité de simulations à exécuter. Les résultats des simulations de ce plan permettent ensuite de calculer des indices de sensibilité (utiles pour étudier le comportement du modèle et adapter éventuellement le plan d'expériences), et plusieurs indices de vulnérabilité (il est montré dans [Lard13] la nécessité d'étudier simultanément différents indices pour une analyse pertinente).

Les résultats des simulations du plan d'expériences servent aussi à construire des surfaces de réponse, i.e. des modèles approximatifs du modèle étudié pour chaque sortie en fonction des entrées, qui remplaceront les simulations, généralement trop coûteuses en temps de calcul, dans la phase de minimisation de la vulnérabilité. Le plan d'expériences peut être complété si la qualité d'approximation des surfaces de réponse est jugée insuffisante.

A l'issue de cette phase d'analyse, des indices de la vulnérabilité du système, sans adaptation, sont obtenus. Il s'agit de mesurer la vulnérabilité du système sans que celui-ci ne s'adapte aux conditions des différents scénarios du plan d'expériences, e.g. sans changement de la politique de gestion de la parcelle.

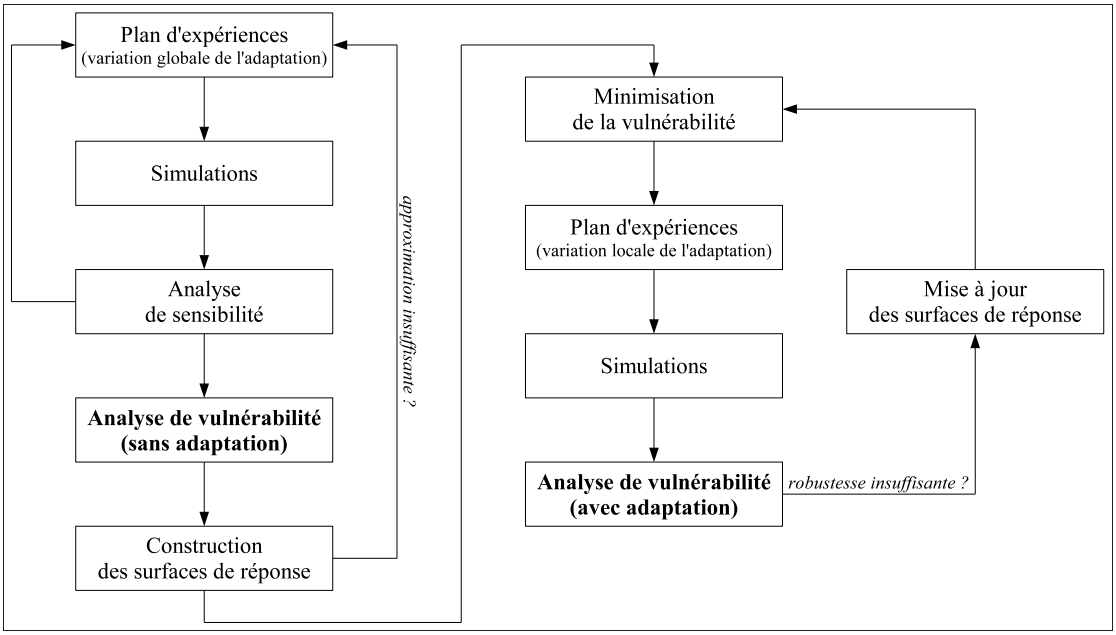

Figure 4.9: Méthodologie d'analyse et de minimisation de la vulnérabilité.

La minimisation de la vulnérabilité consiste ensuite à chercher une adaptation du système qui réduise sa vulnérabilité. Les paramètres d'entrée sont donc décomposés en deux groupes: les paramètres d'adaptation sur lesquels on peut agir (ils expriment la capacité d'adaptation du système), et les paramètres d'environnement qui sont "incontrôlables" (ils traduisent les contraintes de l'environnement sur le système). L'indice de vulnérabilité qui servira à l'évaluation d'une solution d'adaptation doit être sélectionné. Un algorithme d'optimisation (e.g. une métaheuristique) parcourt l'espace des solutions (dans les dimensions des paramètres d'adaptation) pour déterminer l'adaptation assurant une vulnérabilité minimale.

L'indice de vulnérabilité de chaque solution étudiée est calculé à partir d'un plan d'expériences utilisant les surfaces de réponse où l'adaptation varie localement. Les paramètres d'adaptation subissent de légères perturbations (afin de garantir une certaine robustesse), alors que les paramètres d'environnement évoluent dans les domaines de valeurs du plan d'expériences initial.

A l'issue du processus d'optimisation, une solution d'adaptation est fournie dont la vulnérabilité a été évaluée à partir de surfaces de réponse. Il est nécessaire de confirmer la qualité de la solution à partir de simulations, grâce à un plan d'expériences similaire à ceux utilisés pour la minimisation (i.e. où l'adaptation varie localement). Notamment, on évalue la robustesse de la solution par rapport aux paramètres d'adaptation. Si celle-ci est jugée insuffisante, les données collectées par les nouvelles simulations sont utilisées pour affiner les surfaces de réponse et le processus d'optimisation est relancé.

Cette méthodologie est adaptée à tout modèle agro-écologique dont les entrées-sorties sont conformes au modèle présenté à la figure 4.10. Notamment, les entrées sont classées en deux catégories (cf. sous-classes de `Input`) à des fins d'optimisation. Les entrées-sorties du modèle PaSim sont conformes à

ce modèle: les paramètres de climat, de végétation et de sol sont des entrées d'environnement, et les paramètres concernant les herbivores et les politiques de gestion sont des entrées d'adaptation. Ce modèle permet aussi d'exprimer des contraintes sur une ou plusieurs entrées (cf. classe `Constraint`), comme par exemple, imposer un intervalle de valeurs pour une entrée, ou exclure certaines combinaisons de valeurs (e.g. la somme des valeurs de deux entrées distinctes doit être inférieure à un certain seuil).

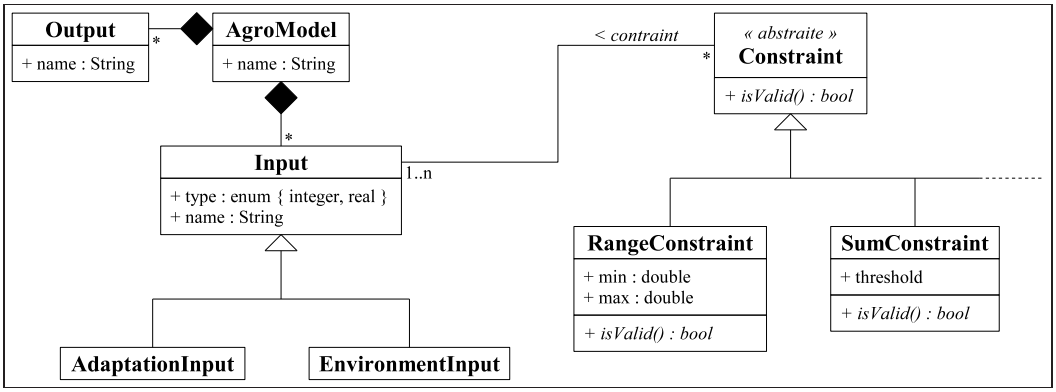

Figure 4.10: Modèle des entrées-sorties d'un modèle agro-écologique.

4.2.3. Analyse de vulnérabilité

4.2.3.1. Plan d'expériences

Un plan d'expériences est une suite d'essais, ici des simulations du modèle, avec différentes valeurs (appelées niveaux) des paramètres d'entrée (appelés facteurs), dans le but de mieux comprendre le comportement du modèle. Idéalement, toutes les combinaisons de niveaux devraient être testées, mais le temps d'exécution risque d'être prohibitif. Un plan d'expériences adapté doit fournir un échantillonnage des combinaisons de niveaux permettant une estimation suffisamment précise des caractéristiques étudiées sur le modèle. Dans notre cas, le plan d'expériences doit permettre des analyses de sensibilité et de vulnérabilité, et la construction de surfaces de réponse de bonne qualité.

Un plan d'expériences peut considérer les variations d'un seul facteur à la fois, on parle de plan OAT (*One At a Time*), ou bien considérer les combinaisons de niveaux de tous les facteurs, on parle alors de plan factoriel. Un plan factoriel est dit complet si toutes les combinaisons sont étudiées, ou bien fractionnaire sinon. Le temps de simulation et le nombre des combinaisons empêchent souvent d'effectuer un plan factoriel complet.

Dans le cas de PaSim, la quantité de facteurs et de niveaux (cf. tableau 4.1) nécessite d'appliquer un plan factoriel fractionnaire (e.g. *Latin Hypercube Design* - LHD [McKa79]). Le domaine de simulation a été réduit pour ne considérer que des climats et des sols représentatifs de la France. Les pratiques agricoles potentielles ont également été restreintes aux simples prairies fauchées. Chacun des 102 sols considérés

est caractérisé par plusieurs paramètres dans le modèle, mais au lieu d'avoir un facteur par paramètre, il a été choisi un facteur qualitatif pour représenter intégralement le sol (car toute combinaison de valeurs des paramètres du sol n'est pas forcément cohérente): chaque sol a un numéro qui correspond à un niveau du facteur. Cependant, les paramètres du sol sont bien utilisés comme entrées du modèle et des surfaces de réponse.

Facteur	Nombre de niveaux	Domaine des niveaux
Sol	102	1..102
$1^{ère}$ année climatique	408	1..408
$2^{ème}$ année climatique	408	1..408
$3^{ème}$ année climatique	408	1..408
Concentration CO_2	8	$[320; 390]$ ppm
Altitude	20	$[50; 1000]$ m
Nombre de fauche	4	1..4
Date $1^{ère}$ fauche	7	[1 avril; 1 mai]
Date $2^{ème}$ fauche	7	[16 mai; 15 juin]
Date $3^{ème}$ fauche	7	[1 juillet; 31 juillet]
Date $4^{ème}$ fauche	7	[16 août; 15 septembre]
Fraction de légumineuse	5	$\{0; 10; 20; 30; 40\}$ %
Taux de fertilisation azotée	6	$\{0, 40, 60, 80, 100, 120\}$ kg/ha

Tableau 4.1: Facteurs considérés pour l'étude avec PaSim.

Les données climatiques sont celles observées sur 12 sites pendant 34 ans (1972-2006). Chaque simulation s'effectue sur trois années consécutives, où le climat correspond pour chacune à un facteur de 408 niveaux (comme pour le sol, un facteur qualitatif représente le climat). La concentration atmosphérique en CO_2 et l'altitude du site ont été ajoutées séparément comme facteurs du climat. La gestion de la parcelle se limite à quatre fauches maximum, à un éventuel apport annuel d'engrais azoté et à décider de la fraction initiale de légumineuses par rapport aux graminées dans la parcelle.

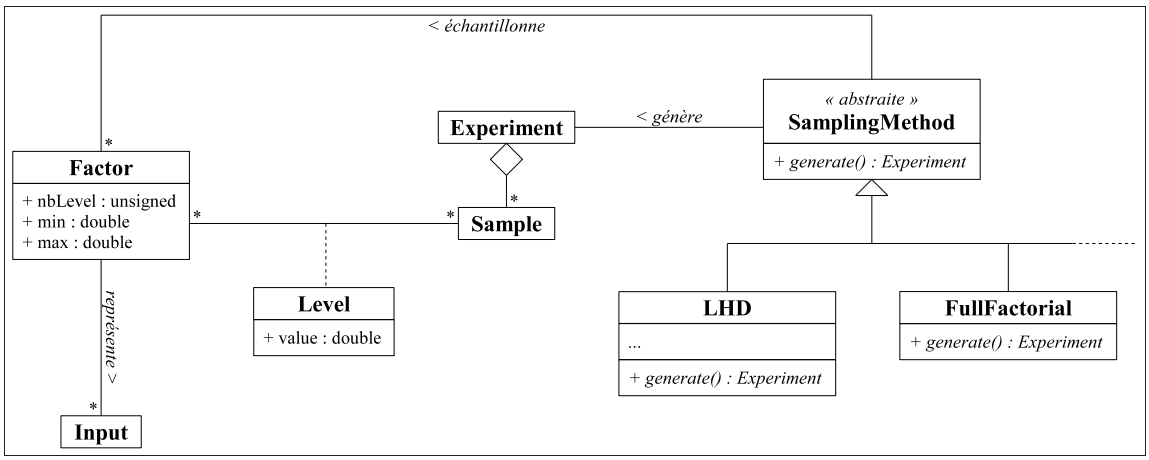

Figure 4.11: Modèle d'un plan d'expériences.

Un plan d'expériences peut être décrit par le modèle présenté à la figure 4.11. Un facteur représente une ou plusieurs entrées du modèle étudié, et se caractérise par un nombre de niveaux dans un domaine de valeurs. La méthode d'échantillonnage (cf. classe `SamplingMethod`) est chargée de construire un plan d'expériences (cf. classe `Experiment`) constitué d'échantillons (cf. classe `Sample`) des niveaux de facteurs (cf. classe `Level`).

4.2.3.2. Indices de vulnérabilité

La vulnérabilité d'un système peut être caractérisée de nombreuses manières, selon les comportements du système que l'on souhaite étudier: le temps de retour à un état normal du système suite à un phénomène extrême, la stabilité ou le risque d'endommagement du système face à des perturbations... Il en résulte qu'un seul indice ne peut pas être suffisant pour décrire la vulnérabilité d'un système [Lard13]. Dans notre étude avec PaSim, nous proposons d'analyser la vulnérabilité d'un système à l'aide de plusieurs indices quantitatifs introduits dans [Fost84] et [Luer03], où la vulnérabilité est mesurée à partir d'une sortie W du modèle, relativement à un seuil W_0 au delà duquel le système est jugé endommagé.

Considérons une parcelle dont la production annuelle W doit être supérieure à $W_0 = 750$ kg/ha de matière sèche pour qu'elle soit rentable. La parcelle est jugée vulnérable si la production est insuffisante, i.e. si $W < W_0$. Supposons que l'on obtienne par simulation la production W_i de la parcelle pour chaque année $i = 1..n$, et que l'on comptabilise le nombre d'années q où $W_i < W_0$. L'indice V_α de mesure généralisée de la pauvreté introduit dans [Fost84] peut être considéré comme un indice de la vulnérabilité.

$$V_\alpha = \frac{1}{n} \sum_{\substack{i=1..n \\ W_i<W_0}} \left(\frac{W_0 - W_i}{W_0}\right)^\alpha$$

Plus la valeur α est grande, plus l'indice est sensible aux individus (ici, les années) les plus vulnérables: pour $\alpha = 0$, il s'agit de la proportion d'individus qui sont en dessous du seuil, i.e. $V_0 = \frac{q}{n}$; pour $\alpha = 1$, il s'agit de l'écart moyen entre le seuil et les mesures W_i des individus vulnérables; pour $\alpha \geq 2$, il s'agit de la moyenne des écarts élevés à la puissance α, ce qui amplifie les écarts des individus les plus vulnérables.

La vulnérabilité peut aussi intégrer une forme de robustesse. Dans [Luer03], l'indice de vulnérabilité V_L est une fonction de la sensibilité $s(W, X)$ d'une sortie W du modèle par rapport à de faibles stress, i.e. de faibles variations d'un sous-ensemble X de paramètres du modèle (e.g. l'indice d'aridité du climat); et de l'écart relatif $\frac{W}{W_0}$ de la sortie W du modèle par rapport au seuil de vulnérabilité W_0.

$$V_L = \frac{s(W, X)}{W/W_0}$$

Dans [Luer03], la sensibilité est représentée par la valeur absolue de la différentielle $\mathrm{d}W$, mais d'autres indices de sensibilité, comme le coefficient de variation de W suite à de faibles variations des paramètres dans X, peuvent être utilisés. Cet indice peut également prendre en compte l'exposition d'un stress,

c'est-à-dire sa probabilité d'apparition dans le système réel (e.g. la probabilité d'avoir une année de sécheresse), ce qui permet ainsi d'affiner l'estimation de l'indice. Considérons plusieurs stress S_j, où $j = 1..p$, qui traduisent par exemple les conditions d'aridité de différents scénarios possibles de changement climatique. La sensibilité $s_j(W, X)$ de W par rapport au stress S_j (i.e. à des variations des paramètres dans X), ainsi que la probabilité d'apparition p_j du stress S_j, peuvent être déterminées. L'indice intégrant l'exposition s'écrit alors comme suit.

$$V_L = \sum_{j=1..p} \frac{s_j(W, X)}{W/W_0} p_j$$

L'analyse de vulnérabilité peut être décrite par le modèle présenté à la figure 4.12 qui prend en compte des indices quantitatifs reposant sur le seuil de vulnérabilité d'une sortie du modèle (cf. classe `Index`), comme les indices de Foster et Luers qui viennent d'être présentés. La distribution de probabilité de chaque facteur (cf. classe `Distribution`) peut être définie pour permettre l'estimation de l'exposition des stress.

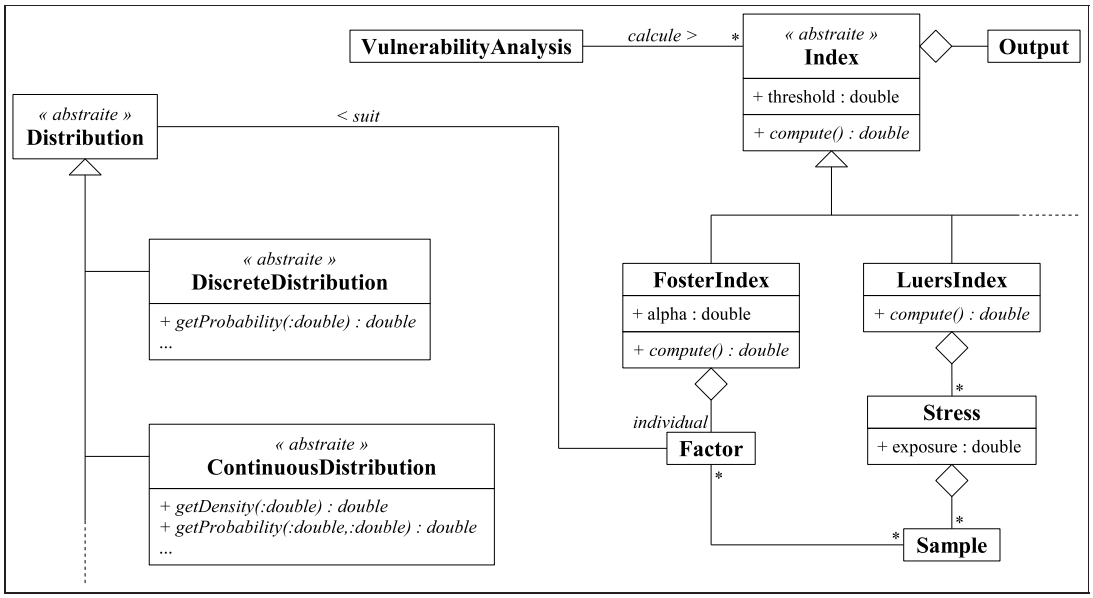

Figure 4.12: Modèle d'une analyse de vulnérabilité.

4.2.4. Adaptation

4.2.4.1. Surfaces de réponse

Une simulation pouvant être longue à exécuter, il semble judicieux de pouvoir remplacer l'évaluation de chaque sortie du modèle de simulation par un modèle simplifié, appelé ici surface de réponse, dépendant des mêmes paramètres d'entrée que le modèle initial. Ensemble, les surfaces de réponse de chacune des sorties peuvent se substituer au modèle initial lors du calcul de la vulnérabilité.

Ces surfaces de réponse doivent être suffisamment simples et précises. Plusieurs approches sont possibles pour les construire: régression polynômiale, splines, réseaux de neurones... Dans notre étude avec PaSim, une régression quadratique fournit une bonne approximation des sorties du modèle (i.e. les écarts entre les valeurs simulées et les valeurs approximées sont très faibles [Lard12b]), suffisante pour une analyse de vulnérabilité (i.e. les écarts relatifs entre les indices de vulnérabilité calculés avec les valeurs simulées et ceux calculés avec les valeurs approximées sont faibles, moins de 5 % [Lard14]).

4.2.4.2. Optimisation

L'objectif maintenant est de trouver la meilleure adaptation possible du système afin de minimiser sa vulnérabilité. Dans notre exemple, il s'agit de trouver une gestion de la parcelle qui minimise la vulnérabilité du stock de matière organique du sol, pour un ensemble de scénarios de climat et de sol donné. La recherche d'une solution se fait donc dans l'espace des possibilités d'adaptation (i.e. des combinaisons de valeurs des paramètres d'adaptation), et son évaluation s'appuie sur les surfaces de réponse, qui prennent comme entrées à la fois les paramètres d'adaptation et d'environnement. Cette évaluation permet d'estimer l'impact d'un jeu de valeurs des paramètres d'adaptation pour un ensemble de scénarios d'environnement (i.e. de combinaisons de valeurs des paramètres d'environnement).

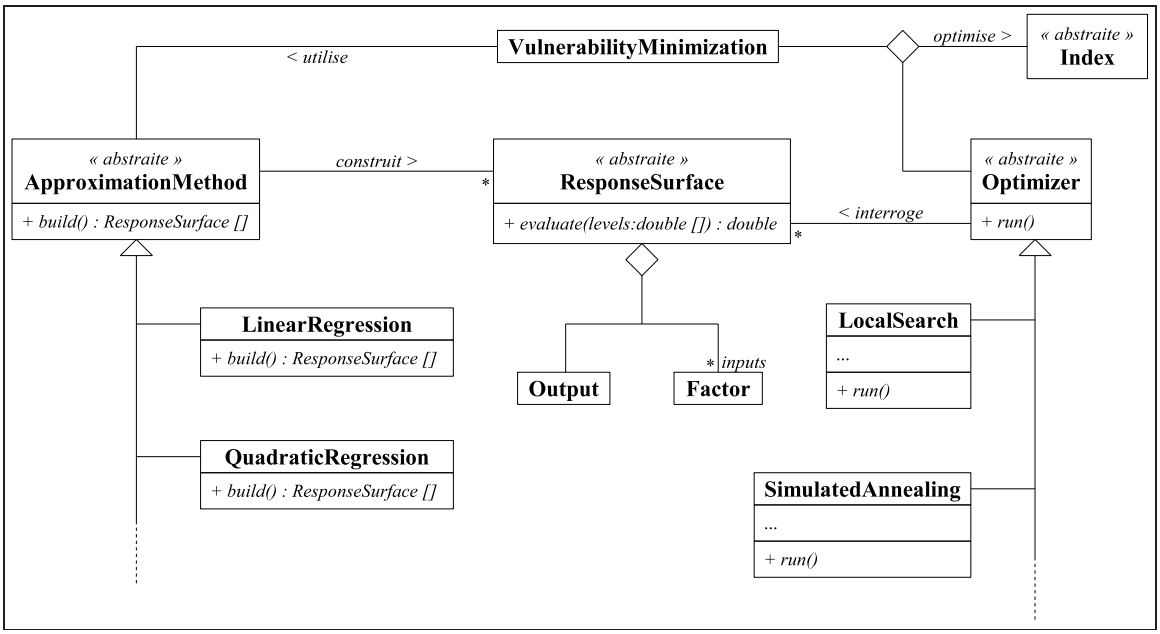

Figure 4.13: Modèle d'une minimisation de la vulnérabilité.

Contrairement à la phase d'analyse qui manipule plusieurs indices, la minimisation de la vulnérabilité s'effectue sur un seul indice, afin d'obtenir une mesure quantitative permettant de comparer deux solutions d'adaptation. Dans notre exemple, une certaine robustesse de la solution d'adaptation étant recherchée, l'indice de Luers est retenu, avec une sensibilité estimée par un coefficient de variation. Il pourrait être envisagé une optimisation multicritère, afin de déterminer une solution d'adaptation qui soit

un compromis pour plusieurs indices de vulnérabilité, ce qui impliquerait des techniques d'optimisation spécifiques (cf. section 2.4) qui diffèrent des algorithmes monocritère et nécessitent des temps de calcul beaucoup plus importants.

De nombreuses méthodes peuvent être envisagées pour résoudre ce problème d'optimisation. Mais dans un cadre générique, il semble pertinent de proposer des approches qui sont indépendantes du type de surface de réponse, même si elles ne garantissent pas d'obtenir une solution optimale. Pour montrer la faisabilité de notre approche, nous avons retenu deux algorithmes simples d'optimisation: une méta-heuristique de recherche locale et le recuit simulé [Lard14]. Il sera possible par la suite de proposer des algorithmes plus adaptés, en fonction du type de surface de réponse, mais également du modèle étudié: des connaissances d'experts peuvent être introduites dans l'algorithme pour guider la recherche vers une adaptation optimale. Le processus d'optimisation peut être décrit par le modèle présenté à la figure 4.13.

4.2.5. Ingénierie des modèles

Afin de fournir un outil logiciel suffisamment générique pour l'analyse et la minimisation de la vulnérabilité d'un modèle agro-écologique, il a été choisi d'aborder la conception de l'application par l'ingénierie des modèles (ou ingénierie dirigée par les modèles), qui permet la manipulation de modèles pour la production de code informatique [Favr04]. Cette approche repose avant tout sur le concept de métamodèle: un métamodèle est un modèle qui décrit une classe de modèles; un code pouvant ainsi vérifier qu'un modèle est conforme à la description exprimée par un métamodèle.

L'approche repose également sur le concept de transformation de modèle: un modèle défini dans un formalisme donné peut être transformé en un modèle représentant le même système exprimé dans un autre formalisme; un code pouvant ainsi transformer, par exemple, un diagramme UML en un code informatique. Avec cette approche, notre méthodologie d'analyse et de minimisation de la vulnérabilité peut être décrite par le métamodèle formé des figures 4.10 à 4.13.

La plateforme de développement EMF (*Eclipse Modeling Framework*[16]) intègre les principes de l'ingénierie des modèles, ce qui permet de définir notre métamodèle et des transformations pour les modèles qui lui sont conformes [Lard13]. L'utilisateur peut alors instancier le métamodèle (créant ainsi un modèle conforme) via l'interface d'EMF, en décrivant les entrées-sorties du modèle étudié et en préparant le plan d'expériences (choix des facteurs et de la méthode d'échantillonnage), l'analyse de vulnérabilité (choix et paramétrage des indices), et la minimisation de la vulnérabilité (choix des méthodes d'approximation et d'optimisation). Les transformations de modèles prédéfinies sont ensuite appliquées pour générer automatiquement les scripts nécessaires à l'analyse et à la minimisation de la vulnérabilité à partir du modèle de l'utilisateur (cf. figure 4.14).

[16] `https://eclipse.org/modeling/emf/`

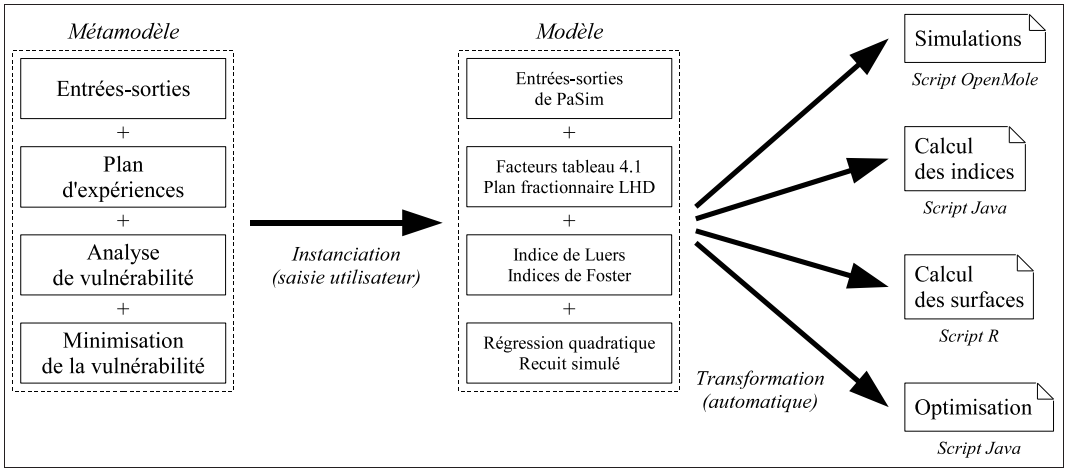

Figure 4.14: Transformations de modèles pour l'analyse et la minimisation de la vulnérabilité.

4.2.6. Résultats numériques

La solution d'adaptation obtenue doit être considérée avec précaution, car elle est issue de données produites par les surfaces de réponse. Même si ces approximations semblent suffisantes pour une analyse de vulnérabilité sans adaptation (cf. section 4.2.4), il convient de réévaluer l'indice de vulnérabilité de la solution d'adaptation à l'aide de simulations. Le plan d'expériences ayant servi au calcul de l'indice avec les surfaces de réponse est donc relancé avec des simulations. Dans notre étude avec PaSim, il apparaît que les indices issus des surfaces de réponse sous-estiment la vulnérabilité par rapport à ceux issus des simulations, alors que les tendances pour différents scénarios de changement climatique sont similaires [Lard13]. Il est probable que ces erreurs soit liées à la nature des surfaces de réponse.

Indice	Période de référence	Futur proche	Futur lointain
V_0	33.3 %	55.1 %	75.7 %
V_1	73.0 %	84.7 %	92.7 %
V_L	65.9 %	67.5 %	67.8 %

Tableau 4.2: Réduction de la vulnérabilité après optimisation de l'adaptation.

A défaut de pouvoir confirmer que la solution d'adaptation obtenue est bien optimale, une comparaison des indices de vulnérabilité sans et avec adaptation (calculés à partir de simulations) permet d'affirmer que l'adaptation trouvée dans notre exemple induit une réduction importante de la vulnérabilité du stock de matière organique du sol. Le tableau 4.2, extrait de [Lard13], montre l'amélioration relative de la vulnérabilité pour trois scénarios climatiques: la période de référence, 1970-2006, qui s'appuie sur des relevés météorologiques; le futur proche, 2018-2051, qui prévoit un réchauffement de $1\ ^\circ\mathrm{C}$ à $2\ ^\circ\mathrm{C}$; et le futur lointain, 2067-2100, qui prévoit un réchauffement de $2\ ^\circ\mathrm{C}$ et $3\ ^\circ\mathrm{C}$.

Si la solution d'adaptation n'est pas satisfaisante (i.e. la robustesse ou la réduction de la vulnérabilité est insuffisante), les simulations qui ont servi à confirmer les indices de vulnérabilité, ou bien de nouvelles

simulations issues d'un plan d'expériences complémentaire (couvrant par exemple des zones supposées contenir des minima de vulnérabilité), peuvent être exploitées pour construire de nouvelles surfaces de réponse et permettre la recherche d'une meilleure solution d'adaptation.

4.3. Conclusion

La plateforme générique UNIF, qui a servi au développement du modèle agro-écologique GEMINI, permet la conception et la simulation de modèles à intégration numérique complexes, formés d'une hiérarchie de sous-modèles qui peut évoluer en cours de simulation (e.g. apparition ou disparition d'une feuille sur une plante). A partir de nombreuses expériences biologiques, le modèle GEMINI a pu être paramétré et validé [Sous12] afin d'étudier les effets de la biodiversité sur les prairies [Mair13]. Ces études ont nécessité environ 340 000 simulations réalisées en 30 jours sur un calculateur parallèle, ce qui équivaut à un temps effectif d'une année. Le modèle GEMINI s'est révélé très fiable, avec un taux d'échec des simulations inférieur à 0.01 %.

Le modèle GEMINI est à la frontière des connaissances scientifiques, il est donc naturel que certains de ses sous-modèles évoluent, ou que l'on souhaite comparer plusieurs approches de modélisation. Permettre de remplacer à volonté les sous-modèles semble indispensable, mais soulève plusieurs difficultés. Tout d'abord, il faut gérer la possibilité de changement de granularité d'un sous-modèle, en particulier de ses entrées-sorties qui interviennent dans le couplage avec d'autres modèles. Pour cela, il convient de réviser la structure de la plateforme UNIF pour y intégrer les échanges entre modèles comme modèles à part entière [Wini05].

Le remplacement d'un sous-modèle soulève aussi le problème de déterminer l'ordre dans lequel toutes les variables du modèle seront calculées à chaque pas d'intégration. En effet, le couplage de modèles peut induire des dépendances dans leurs équations [Gay06], il faut donc être capable d'analyser les formules pour ordonner le processus d'intégration en conséquence. La plateforme UNIF étant développée en C++, on peut envisager la conception d'un langage dédié embarqué (*Embedded Domain Specific Language* - EDSL [Fowl10]) pour décrire les équations, cette solution reposant sur des techniques de métaprogrammation générique (cf. section 6.3).

Ce chapitre présente aussi une méthodologie pour l'analyse et la minimisation de la vulnérabilité d'un système agro-écologique. Une première étape, qui s'appuie sur le calcul de plusieurs indices quantitatifs, permet d'estimer la vulnérabilité d'un système face à un ensemble de scénarios environnementaux, sans qu'aucune stratégie d'adaptation particulière ne soit appliquée. Ensuite, pour un indice choisi, une stratégie d'adaptation minimisant la vulnérabilité du système est recherchée. L'approche est générique afin de permettre l'étude de tout système agro-écologique, en choisissant les indices de vulnérabilité, le plan d'expériences de l'analyse, ainsi que les méthodes d'approximation du modèle et d'optimisation de la stratégie d'adaptation.

Cette méthodologie a été appliquée dans le cadre d'études sur la vulnérabilité du stock de matière organique du sol d'une prairie face à de probables changements des conditions climatiques, à l'aide des estimations du modèle PaSim [Lard13]. L'utilisation de surfaces de réponse pour la recherche d'un minimum de vulnérabilité permet un gain conséquent en temps de calcul, mais l'approximation qui est induite entraîne une incertitude difficilement quantifiable sur la qualité de la solution d'adaptation trouvée, même si l'on a pu constater une diminution significative de la vulnérabilité pour le cas discuté dans ce chapitre.

Nous avons montré qu'il est possible de minimiser la vulnérabilité d'un système en employant des méthodes d'optimisation simples et génériques (une métaheuristique de recherche locale et le recuit simulé). Cependant, il convient d'explorer d'autres approches, notamment des techniques d'optimisation adaptées au type de surface de réponse, qui pourraient garantir l'optimalité de la solution d'adaptation obtenue; ou des heuristiques dédiées au système étudié, où les connaissances d'experts pourraient servir à guider la recherche vers une solution d'adaptation optimale.

Partie III - Développement
Générique et Métaprogrammation

CHAPITRE 5
COMPOSANTS GÉNÉRIQUES POUR L'OPTIMISATION

La recherche opérationnelle utilise des méthodes mathématiques sophistiquées pour optimiser des systèmes. Leur implémentation est souvent difficile et prendre en considération des problématiques de génie logiciel avancées la rend encore plus complexe. En outre, la taille de certains problèmes pratiques implique une implémentation efficace des algorithmes si l'on souhaite obtenir des temps de calcul acceptables. L'objectif ici est de montrer comment concevoir des algorithmes génériques et efficaces pour l'optimisation (ou tout autre domaine scientifique où le calcul est omniprésent) en combinant la programmation objet et la programmation générique.

Ce chapitre étudie des solutions de conception de structures de données et d'algorithmes génériques dans le cadre de problèmes d'optimisation dans les graphes et compare leur efficacité. Des patrons de conception connus sont étudiés dans leur version générique, et de nouveaux schémas sont proposés pour répondre à des problèmes récurrents dans le développement d'algorithmes d'optimisation, notamment concernant la nécessité d'étendre une structure de données pour les besoins d'un algorithme.

5.1. Performance et généricité

Le terme algorithme représente ici un traitement complexe que l'on cherche à rendre générique au sens large, c'est-à-dire que l'algorithme doit être extensible (son comportement doit pouvoir être adapté simplement pour répondre à divers objectifs, ce qui contribue fortement à la réutilisabilité) et indépendant (bien qu'il puisse interagir fortement avec d'autres composants logiciels, structures de données ou algorithmes, il doit rester le plus indépendant possible de ceux-ci). En résumé, on cherche à concevoir des composants adaptables au plus de situations possibles et interchangeables lorsque leurs rôles sont similaires. La difficulté est d'atteindre ces objectifs tout en conservant une très bonne efficacité des algorithmes, fondamentale dans des domaines scientifiques comme l'optimisation.

La principale manière de rendre un composant (une classe, une méthode ou une fonction) générique (au sens large) consiste à faire abstraction du type véritable des objets ou des données qu'il manipule. En programmation objet, cette abstraction passe par l'héritage et les méthodes virtuelles. Cependant, une partie du mécanisme est réalisée à l'exécution, ce qui entraîne un surcoût (notamment à cause de *downcast* et du *dynamic binding*). En programmation générique, l'abstraction se fait par une paramétrisation supplémentaire des composants: de manière traditionnelle, un composant manipule des valeurs inconnues (attributs, arguments de fonction...); avec la programmation générique, un composant peut également être paramétré par des types inconnus. Dans certains langages comme C++, le mécanisme est réalisé exclusivement à la compilation, ce qui n'induit que rarement un surcoût à l'exécution.

5.1.1. Programmation objet

Diverses raisons peuvent conduire à une conception par objets inefficace, des mauvais choix du concepteur aux limitations du paradigme. Plus spécifiquement, les applications scientifiques ne semblent pas très bien adaptées à un usage important de l'héritage, et malheureusement, lorsqu'on cherche l'extensibilité, ce concept semble incontournable dans une conception par objets. Souvent les algorithmes manipulent de nombreuses données de petite taille (comme les éléments d'une matrice) et le simple fait d'accéder à ces données par une méthode virtuelle peut conduire à de mauvaises performances. Une méthode virtuelle ne peut être liée au code qui l'appelle qu'au moment même de l'appel, par le mécanisme appelé liaison dynamique (*dynamic binding*), et nécessite donc plus de temps qu'une méthode classique, liée statiquement, pour être exécutée.

Si le langage permet l'*inlining* d'une méthode (i.e. le remplacement de l'appel d'une méthode par le code de cette méthode), il est possible d'obtenir un temps d'exécution de la méthode bien inférieur à celui d'une exécution classique [Lipp96a, ORio02]. Ce gain s'explique tout d'abord par le mécanisme d'appel de fonction qui est évité, mais surtout par les optimisations qui peuvent alors être réalisées par le compilateur: le code de la méthode est directement placé dans le contexte d'utilisation, ce qui permet des arrangements du code qui sont inenvisageables à travers un appel de méthode. Comme la liaison dynamique empêche l'*inlining* d'une méthode virtuelle, il est nécessaire d'éviter la virtualité sur certaines méthodes critiques (i.e. au contenu très rapide d'exécution et appelées très souvent).

5.1.2. Programmation générique

La programmation générique propose donc de paramétrer les composants non plus sur des valeurs uniquement, mais également sur des types [Muss89]. De tels composants sont dits génériques ou bien appelés patrons de composant. Un composant générique peut donc avoir deux catégories de paramètres: les arguments qui sont des valeurs inconnues et les paramètres génériques (que l'on nommera plus simplement paramètres par la suite) qui sont des types inconnus. Prenons l'exemple C++ de la fonction générique `max` qui retourne le maximum de deux valeurs dont le type n'est pas spécifié par avance.

```
template <class T>
inline const T & max(const T & a,const T & b) { return (a>b ? a : b); }
```

Le type `T` est un paramètre générique et les deux variables `a` et `b` sont des arguments de la fonction générique. Un composant générique est un modèle de composant, il sert à instancier des composants, i.e. à produire des composants en levant l'inconnue sur les paramètres. Contrairement aux arguments, les valeurs des paramètres sont connues à la compilation, l'instanciation peut donc être effectuée à la compilation. Avec C++, un composant sera produit pour chaque jeu de valeurs des paramètres, et l'instanciation équivaut à un copier-coller-remplacer.

Dans notre exemple, la fonction générique `max` peut être instanciée de la manière suivante.

```
max<double>(3.5,4.2);
```

La fonction `max<double>`, qui est une instance du modèle `max` où `T = double`, est strictement équivalente d'un point de vue performance à la fonction suivante.

```
inline const double & max_double(const double & a,const double & b)
{ return (a>b ? a : b); }
```

Cette manière d'instancier les composants génériques, spécifique à quelques langages comme C++, garantit un surcoût quasi toujours nul à l'exécution; contrairement à d'autres langages, comme Java, qui pratiquent une technique d'effacement de type (*type erasure* [Bracha04]) pour éviter d'avoir une instance par jeu de paramètres, mais qui revient à appliquer une approche par héritage avec tous les défauts de performance présentés précédemment. Nous avons retenu le langage C++ pour nos travaux car il présente de nombreux avantages pour la performance du code généré: un mécanisme de généricité sans surcoût, la possibilité d'*inlining* et la possibilité de spécialisation des composants génériques (ce qui augmente la capacité d'extension des composants et permet la mise en oeuvre de techniques de métaprogrammation, cf. chapitre 6).

5.2. Structures de données génériques

5.2.1. Généricité et héritage

Considérons une classe `Graph` qui représente un graphe orienté en recherche opérationnelle: un graphe est composé de noeuds et d'arcs, où chaque arc relie un noeud source à un noeud cible. Tentons ici de fournir une structure de données unique qui peut être utilisée pour modéliser différentes sortes de graphes: par exemple des graphes de flot qui modélisent des flux circulant à travers un réseau, ou des graphes géographiques qui modélisent des positions et des routes qui les séparent. La structure de données doit alors être capable de porter divers types de données à la fois sur les arcs et sur les noeuds du graphe.

Un modèle utilisant l'héritage est d'abord proposé (cf. figure 5.1). Les interfaces `NodeData` et `ArcData` représentent de manière abstraite les données portées respectivement par les noeuds et par les arcs du graphe. Ainsi, une sous-classe `Flow` devra être définie pour modéliser les données des arcs d'un graphe de flot par exemple. Un algorithme manipulant un graphe de flot supposera alors que les données sur les arcs appartiennent à la classe `Flow`. En demandant au graphe les données portées par un arc, l'algorithme obtiendra un objet du type `ArcData` qu'il devra alors convertir en objet du type `Flow` pour accéder aux données de flot. Cette opération de conversion d'une classe vers l'une de ses sous-classes, appelée *downcast*, n'est pas immédiate (contrairement à l'opération inverse) et nécessite une vérification qui ne peut être effectuée qu'à l'exécution, ce qui engendre un surcoût.

Avec cette approche, un algorithme est dépendant des types des données portées par le graphe. Dans l'exemple précédent, l'algorithme manipule explicitement des objets de type `Flow`. Si l'on souhaite utiliser cet algorithme avec des types de données différents (mais avec la même interface que `Flow`) sur les arcs du graphe, on est obligé d'étendre la classe `Flow` par héritage, ce qui implique la définition de méthodes virtuelles dans l'interface de la classe `Flow`. Les algorithmes d'optimisation sur les graphes font généralement énormément appel à ces méthodes, ce qui risque de conduire à un surcoût significatif.

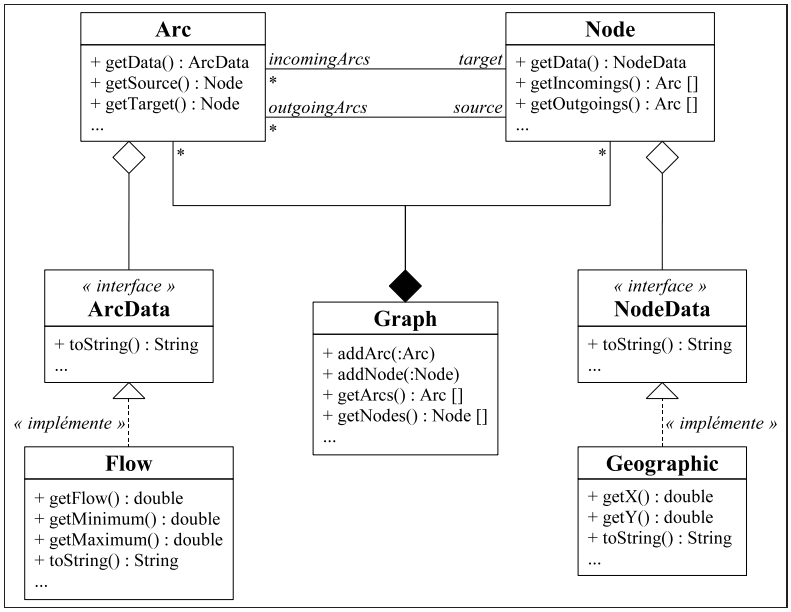

Figure 5.1: Graphe modélisé par héritage.

De ce constat, un modèle utilisant la généricité est proposé (cf. figure 5.2). La classe `Graph` devient générique avec les paramètres `TN` et `TA` qui représentent respectivement les types des données portées par les noeuds et par les arcs. Ces paramètres sont contraints par les concepts `NodeData` et `ArcData` (un concept est un ensemble de spécifications syntaxiques et sémantiques, cf. section 6.1.3). Ainsi, une classe `Flow` peut être définie pour représenter les données des arcs d'un graphe de flot, à condition que `Flow` modélise le concept `ArcData` (i.e. respecte les spécifications du concept), ce qui garantit que l'instanciation de la classe générique `Graph` avec `TA = Flow` est possible.

Un algorithme manipulant un graphe de flot n'est alors plus forcément dépendant des types de données portées par le graphe, puisque ces types peuvent être des paramètres de l'algorithme. De cette manière, il n'y a plus de problème de *downcast* (car le type concret des données est manipulé directement dans l'instance de l'algorithme générique), et utiliser l'algorithme avec d'autres types de données n'implique plus systématiquement de l'héritage (il suffit d'instancier l'algorithme et le graphe génériques avec d'autres paramètres).

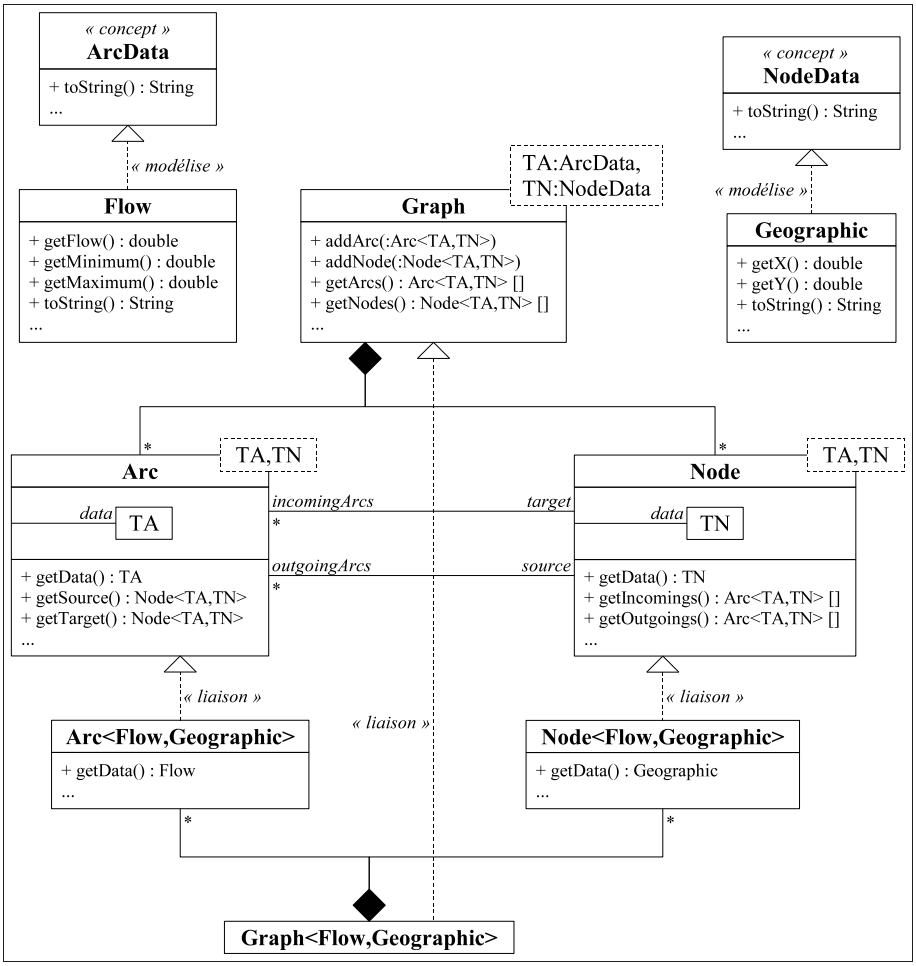

Figure 5.2: Graphe modélisé par généricité.

Nous proposons de comparer ces deux approches, en mesurant sur une implémentation en C++ le temps d'accès aux données portées par les arcs d'un graphe de flot. Nous étudions trois situations:

- Avec l'approche par héritage, lorsqu'on accède aux données d'un arc, on obtient un objet du type `ArcData` qu'il faut convertir dans le type `Flow`. Cette conversion descendante n'est pas automatique et sa validité doit être vérifiée à l'exécution, avec l'instruction `dynamic_cast`:

  ```
  double f = dynamic_cast<Flow &>(arc.getData()).getFlow();
  ```

- Toujours avec l'approche par héritage, il est possible de forcer une conversion descendante et d'éviter ainsi une vérification à l'exécution, avec l'instruction `static_cast`:

  ```
  double f = static_cast<Flow &>(arc.getData()).getFlow();
  ```

- Avec l'approche par généricité, aucune conversion n'est nécessaire, puisque le code est instancié spécifiquement pour des données de type `Flow` sur les arcs:

  ```
  double f = arc.getData().getFlow();
  ```

Le tableau 5.1, extrait de [Bach06a], montre le temps nécessaire pour calculer la somme des flots des arcs d'un graphe avec chacune des trois situations exposées précédemment [17]. On s'aperçoit que la liaison dynamique nécessaire à l'approche par héritage est très coûteuse. En revanche, le bénéfice apporté par une conversion descendante statique (i.e. sans vérification) semble bien faible par rapport aux risques que son utilisation engendre (l'application est moins robuste).

Approche	Temps d'exécution
(1) Héritage avec conversion dynamique	36.1 s
(2) Héritage avec conversion statique	31.2 s
(3) Généricité	7.4 s

Tableau 5.1: Performance des différentes modélisations d'un graphe.

5.2.2. Indépendance des structures de données

De nombreux algorithmes sont par nature indépendants des structures de données, des collections, qu'ils manipulent. D'un point de vue logiciel, il est donc nécessaire de les concevoir de manière à ce que les collections soient interchangeables autant que possible. Pour cela, une solution communément employée consiste à insérer un objet intermédiaire entre un algorithme et une collection, cet intermédiaire implémentant la même interface quelle que soit la collection associée. Notamment, pour le parcours d'une collection, le patron de conception *itérateur* [Gamm95] est couramment utilisé. La figure 5.3 montre ce patron dans sa version générique (toute classe représentant un itérateur doit modéliser le concept `Iterator` au lieu d'implémenter une interface).

Certaines conceptions (e.g. *Standard Template Library* - STL [Aust99]) proposent qu'un algorithme soit paramétré sur les types des itérateurs qu'il manipule. De cette manière, l'algorithme est totalement indépendant des collections associées (cf. figure 5.3). Par exemple, un algorithme `MaxFlow` qui recherche le flot maximum porté par les arcs d'un graphe peut s'écrire de la manière suivante [18].

```
template <class I> double MaxFlow::run(I && first,I && last) const {
 double m = std::numeric_limits<double>::min();

 while (first!=last) {
  m = std::max(m,first->getData().getFlow());
  ++first;
 }

 return m;
}
```

Supposons maintenant un algorithme qui calcule le degré (i.e. le nombre d'arcs reliés) de tous les noeuds du graphe. Deux approches sont possibles: les noeuds sont analysés un par un et les arcs reliés sont comptés; ou bien les arcs sont analysés un par un, et pour leurs noeuds source et cible, un compteur est

[17] Tests complets: `http://www.nawouak.net/?cat=informatics.generic_or`
[18] Les arguments sont des références "universelles" ici (syntaxe `&&`), i.e. on ne se préoccupe pas de leur aspect constant.

incrémenté. Dans la première version, l'algorithme a besoin d'itérateurs sur les noeuds, alors que dans la seconde, il lui faut des itérateurs sur les arcs. Cela signifie des interfaces différentes pour deux versions d'un même algorithme, ce qui n'est pas satisfaisant.

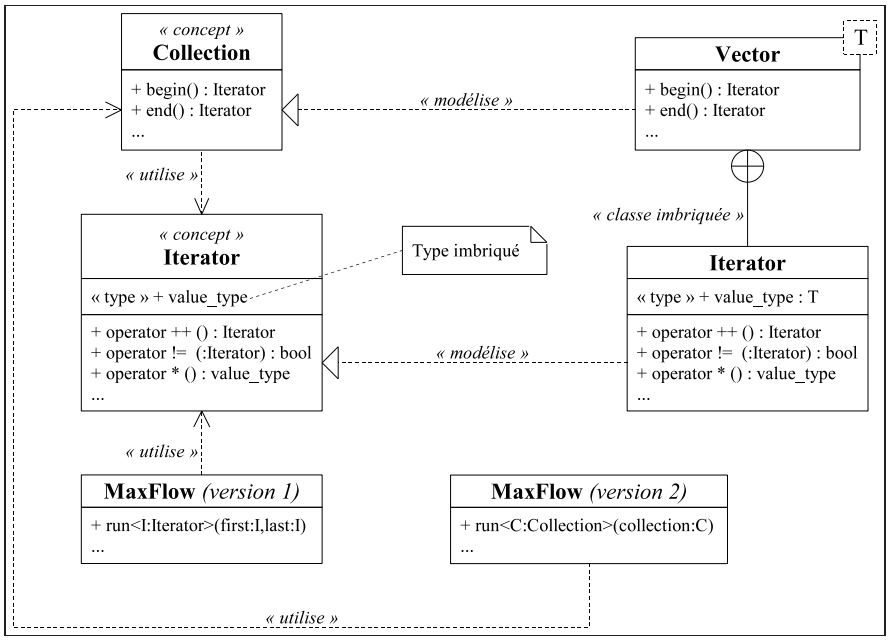

Figure 5.3: Version générique du patron de conception itérateur.

Une meilleure conception (e.g. *Boost Graph Library* - BGL [Siek02]) consiste à proposer un algorithme paramétré sur le type de la collection qu'il manipule tout en conservant le mécanisme des itérateurs: l'algorithme parcours toujours la collection à l'aide d'itérateurs, mais ils sont fournis par la collection elle-même via une interface commune à toutes les collections (modélisée ici par le concept `Collection`, elle garantit que les collections sont interchangeables). L'algorithme `MaxFlow` s'écrira alors de la manière suivante [19].

```
template <class C> double MaxFlow::run(const C & collection) const {
 typename C::Iterator first = collection.begin();
 typename C::Iterator last  = collection.end();

 [...]
}
```

Pour notre implémentation C++, nous avons analysé l'impact des itérateurs dans le parcours des listes d'arcs et de noeuds d'un graphe. Nous avons repris la procédure de la section précédente (test n^{o}3), et nous avons mesuré, dans les mêmes conditions, le temps d'exécution avec des itérateurs (test n^{o}4). Le tableau 5.2, extrait de [Bach06a], montre que l'utilisation d'itérateurs (dans leur version générique) n'a quasiment aucun impact sur les temps de calcul, ce qui est possible grâce à l'*inlining* des méthodes des itérateurs.

[19] Le mot-clé `typename` est nécessaire ici pour confirmer qu'il s'agit bien d'un type imbriqué.

Approche	Temps d'exécution
(3) Généricité (sans itérateur)	7.4 s
(4) Généricité (avec itérateurs)	7.5 s

Tableau 5.2: Performance des itérateurs.

5.3. Algorithmes génériques

Un algorithme générique doit être indépendant des composants qu'il manipule, les structures de données comme les algorithmes. La section précédente explique comment abstraire les collections, nous rappelons maintenant une solution de conception classique qui permet la même chose pour les algorithmes. Diverses techniques sont ensuite étudiées pour rendre un algorithme extensible. Dans un souci de clarté, un graphe sera représenté par une simple classe, sans paramètre générique, tout au long de cette section.

5.3.1. Abstraction des algorithmes

Dans la section précédente, un algorithme est représenté par une classe avec une méthode `run`. Une telle classe peut posséder des attributs permettant de définir des paramètres pour l'algorithme, chaque objet de la classe représentant ainsi l'algorithme avec différents paramètres. Reposant sur cette modélisation, le patron de conception *stratégie* [Gamm95] permet d'identifier une famille d'algorithmes. Cette modélisation propose de représenter une famille d'algorithmes par une classe abstraite (e.g. `ShortestPathAlgo` pour les algorithmes de plus court chemin dans un graphe), et chaque algorithme sera une sous-classe (e.g. `BellmanAlgo`, `DijkstraAlgo`... pour différents algorithmes de plus court chemin [Ahuj93]) comme le montre la figure 5.4.

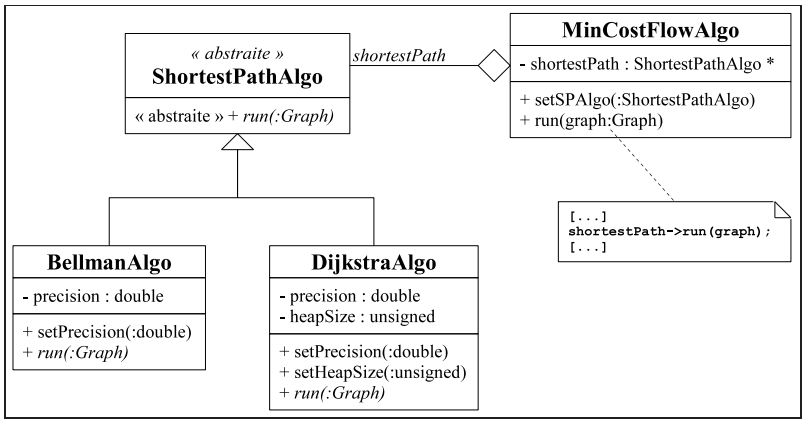

Figure 5.4: Patron de conception stratégie.

La méthode `run` de la classe `ShortestPathAlgo` est abstraite et doit être redéfinie dans les sous-classes. De cette manière, les différents algorithmes de plus court chemin deviennent interchangeables dans tout composant qui manipule un objet `ShortestPathAlgo` (e.g. `MinCostFlowAlgo`, un algo-

rithme de flot de coût minimum dans un graphe qui utilise la recherche de plus courts chemins). On peut estimer que le surcoût engendré par la virtualité de la méthode `run` sera insignifiant comparé au temps d'exécution généralement important de ce genre d'algorithme. Au cas où le surcoût serait problématique, une version générique du patron stratégie peut être envisagée (cf. politique, section suivante).

Pour être interchangeables, les algorithmes respectent l'interface imposée par `ShortestPathAlgo`. Il n'est donc pas possible de fournir les valeurs de paramètres à l'algorithme au moment de l'appel à la méthode `run`, cela doit se faire avant et par d'autres méthodes propres à la classe représentant l'algorithme (e.g. `setPrecision`, `setHeapSize` dans la figure 5.4). Supposons que les deux algorithmes de plus court chemin définissent comme paramètre la précision avec laquelle comparer la longueur de deux arcs; et que l'algorithme de Dijkstra utilise une structure de tas dont la taille est aussi un paramètre. La construction et l'appel de l'algorithme `MinCostAlgo` avec comme algorithme de plus court chemin `DijkstraAlgo` se fera de la manière suivante.

```
DijkstraAlgo    algoPCC;
MinCostFlowAlgo algoFlot;
Graph           graphe = [...];

algoFlot.setSPAlgo(algoPCC);
algoPCC.setPrecision(1e-6);
algoPCC.setHeapSize(1000);
algoFlot.run(graphe);
```

5.3.2. Extension des algorithmes

Cette section discute des manières de rendre un algorithme extensible, l'idée étant de délocaliser certaines parties de son code dans des méthodes séparées (que nous appellerons méthodes paramètres) qui pourront être remplacées par l'utilisateur. De cette manière, le comportement de l'algorithme peut être adapté tout en conservant sa structure générale. En outre, il n'est pas nécessaire que l'utilisateur connaisse tous les détails d'implémentation de l'algorithme, seules quelques informations pertinentes sur les méthodes qu'il peut remplacer suffisent.

5.3.2.1. Approche par méthode patron

Le patron de conception *méthode patron* [Gamm95] est une solution classique pour rendre un algorithme extensible. Il permet d'externaliser certaines parties de la méthode `run` d'un algorithme dans des méthodes virtuelles de la même classe appelées méthodes patrons. Ces méthodes, exécutées à partir de la méthode `run` peuvent être redéfinies par héritage pour en modifier le comportement, laissant le code principal intact.

La figure 5.5 montre comment rendre un algorithme de plus court chemin (cf. classe `ShortestPathAlgo`) indépendant de la manière de calculer la longueur d'un arc. Cela permet de rechercher un plus court chemin en termes de distance, de temps ou de tout autre critère. La méthode `run` de l'algorithme utilise la méthode patron `getLength` pour connaître la longueur d'un arc. Pour adapter

l'algorithme, il suffit de définir une sous-classe (e.g. `ShortestTimePathAlgo`) et de redéfinir la méthode patron.

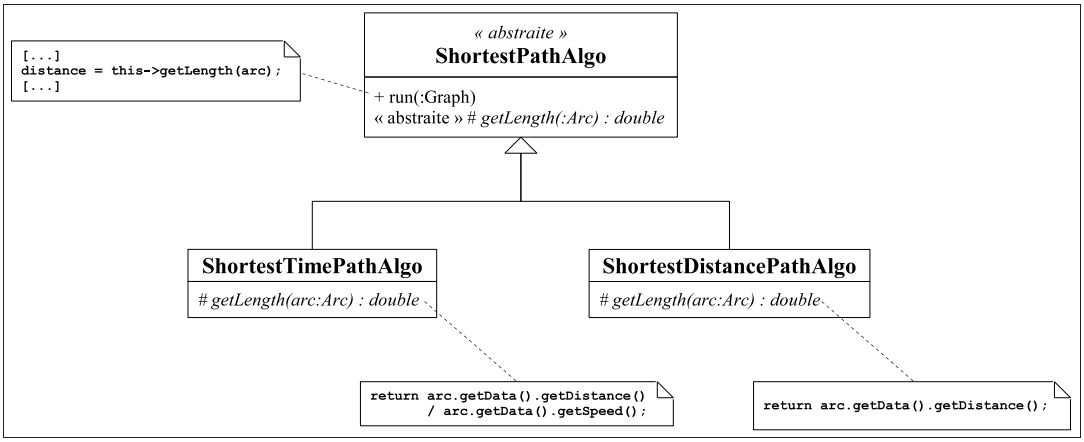

Figure 5.5: Extension d'un algorithme, approche par méthode patron.

Un surcoût conséquent doit être envisagé dans la liaison dynamique induite par les méthodes virtuelles, surtout si les méthodes patrons sont petites et appelées souvent. L'approche est également trop rigide. La relation entre l'algorithme et ses méthodes patrons est statique (il n'est donc pas possible d'associer dynamiquement un algorithme avec des méthodes paramètres), et plus important, dans le cas où il y a plusieurs méthodes paramètres, toute combinaison de versions de méthodes paramètres conduit à définir une sous-classe.

5.3.2.2. Approche par stratégie

Pour rendre l'extension d'un algorithme plus flexible, il est possible d'appliquer le patron stratégie présenté précédemment. Chaque méthode paramètre est représentée par une interface avec une méthode `run`. Pour définir une nouvelle version d'une méthode paramètre, il suffit d'implémenter sa méthode `run` par héritage. L'algorithme principal possède des objets représentant les méthodes paramètres et utilise leur méthode `run`.

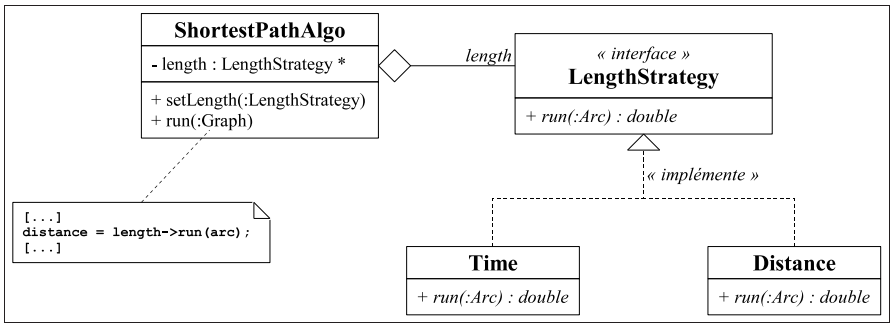

Figure 5.6: Extension d'un algorithme, approche par stratégie.

La figure 5.6 reprend l'exemple de la section précédente. Une interface `LengthStrategy` représente la méthode paramètre qui permet d'obtenir la longueur d'un arc. Pour adapter l'algorithme, il suffit de définir une sous-classe (e.g. `Time`) et d'implémenter sa méthode `run`. L'algorithme final sera construit et utilisé de la manière suivante.

```
Time            temps;
ShortestPathAlgo algoPCC;
Graph           graphe = [...];

algoPCC.setLength(temps);
algoPCC.run(graphe);
```

Avec cette approche, il est possible d'associer dynamiquement des méthodes paramètres à un algorithme (e.g. `setLength`). En revanche, la virtualité est encore appliquée sur des méthodes critiques, ce qui risque de conduire à un surcoût significatif.

5.3.2.3. Approche par politique

Pour éviter toute liaison dynamique, la classe de l'algorithme doit devenir générique avec pour paramètres les types des méthodes paramètres. L'approche reste celle du patron stratégie, mais le terme politique est généralement privilégié dans ce contexte générique. Au lieu d'être modélisées par des interfaces, les méthodes paramètres sont décrites par des concepts qui vont contraindre les paramètres génériques de l'algorithme.

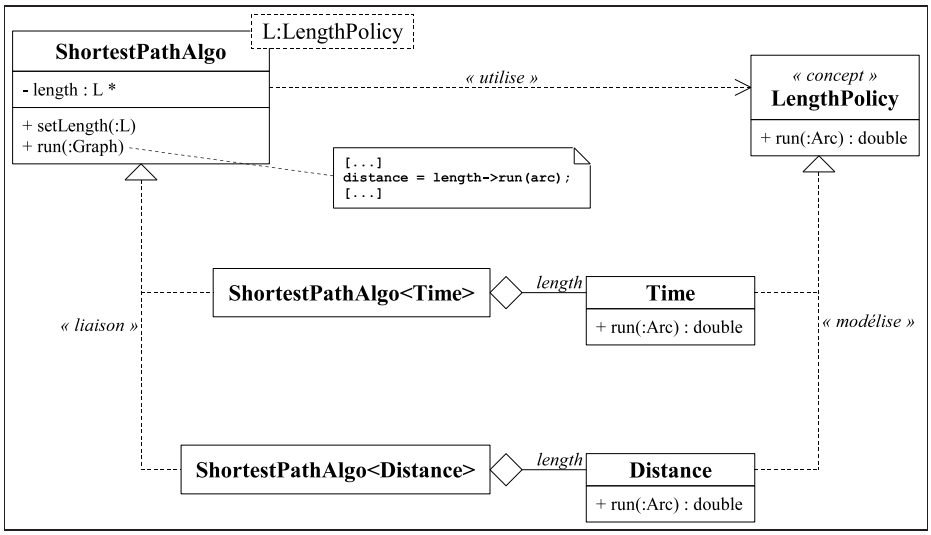

Figure 5.7: Extension d'un algorithme, approche par politique.

La figure 5.7 reprend l'exemple de la section précédente. Un concept `LengthPolicy` décrit la méthode paramètre qui permet d'obtenir la longueur d'un arc, en imposant notamment une méthode `run`. Pour adapter l'algorithme, il suffit de définir une classe (e.g. `Time`) qui modélise le concept `LengthPolicy` et d'implémenter sa méthode `run`. L'algorithme final sera construit et utilisé de la manière suivante.

```
Time                    temps;
ShortestPathAlgo<Time> algoPCC;
Graph                   graphe = [...];

algoPCC.setLength(temps);
algoPCC.run(graphe);
```

Il n'y a donc plus de liaison dynamique pouvant engendrer un surcoût. Néanmoins, l'utilisateur est libre de définir une méthode paramètre avec une méthode `run` virtuelle s'il souhaite obtenir toute la flexibilité du patron stratégie (i.e. la sélection dynamique des méthodes paramètres).

5.3.2.4. Comparaison des approches

L'implémentation en C++ de notre exemple permet de comparer les approches d'extension présentées avec l'approche classique consistant à écrire deux versions distinctes de l'algorithme (pour le temps et la distance). Le tableau 5.3, extrait de [Bach06a], montre les écarts de performance entre les différentes solutions.

Approche	Temps d'exécution
(a) Méthode patron	20.9 s
(b) Stratégie	20.9 s
(c) Politique	18.7 s
(d) Classique	18.7 s

Tableau 5.3: Performance des différentes approches d'extension d'un algorithme.

Les approches par méthode patron et par stratégie sont finalement équivalentes, puisque le défaut est le même dans les deux: l'appel aux méthodes paramètres via des méthodes virtuelles. L'approche par politique montre quant à elle des performances équivalentes à l'approche classique. L'écart n'est ici que de 10 % car les appels à la méthode paramètre ne sont pas si fréquents par rapport au volume d'opérations de l'algorithme. Un test similaire à celui présenté à la section 5.2, qui consisterait à faire simplement la somme des longueurs sur les arcs (obtenues par la méthode paramètre), aurait creusé les écarts entre les méthodes.

5.4. Structures de données extensibles

La conception d'algorithmes génériques peut nécessiter d'étendre une collection manipulée, de manière à la compléter d'informations additionnelles utiles pour l'exécution de l'algorithme. Par exemple, certains algorithmes de résolution du problème de flot de coût minimal requièrent l'affectation d'un potentiel sur les noeuds du graphe à optimiser. Cependant, les noeuds d'un graphe de flot n'ont pas été prévus pour porter cette information. Il s'agit d'un détail d'implémentation de l'algorithme que l'utilisateur n'a pas à connaître.

Comme réponse à ce problème, la bibliothèque BGL (*Boost Graph Library* [Siek02]) propose la notion de *property map* qui fournit un cadre générique pour gérer des données additionnelles. L'approche consiste à stocker ces données dans une structure associative (association noeud-potentiel) indépendante du graphe. Cependant, une telle solution nécessite de synchroniser la structure associative avec les modifications éventuelles du graphe, et ne permet pas un accès aux données additionnelles d'un noeud en temps constant (i.e. en $O(1)$ opérations). Nous étudions donc ici une solution pour stocker des données additionnelles, que nous appellerons extensions, directement sur les éléments d'une collection, ce qui permet d'obtenir un accès direct aux extensions et de supporter naturellement les modifications dans la collection.

5.4.1. Modélisation d'une extension

Il peut être nécessaire de stocker plusieurs extensions sur un élément. Elles seront donc stockées dans une liste, mais comme les données peuvent être de différentes natures, la seule possibilité d'abstraction des types des extensions est l'héritage via une classe abstraite ou une interface qui représente n'importe quel type d'extension (cf. figure 5.8).

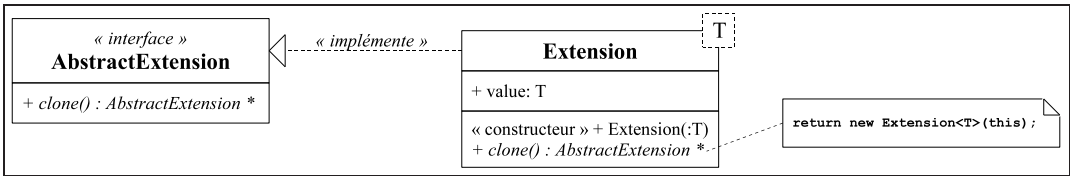

Figure 5.8: Abstraction du type de l'extension.

5.4.2. Gestion des extensions

L'approche consiste ensuite à remplacer la collection à étendre par un intermédiaire qui modélise le même concept augmenté des fonctionnalités d'extension. Le principe est celui du patron de conception *proxy* [Gamm95], mais avec les avantages de la programmation générique (l'abstraction de type se fait par paramètre générique). Une fois à la place de la collection, l'intermédaire reçoit les appels de méthode et redirige ceux faisant partie du concept commun vers l'objet qu'il remplace (il s'agit du mécanisme de délégation [Gamm95]).

Prenons l'exemple d'un graphe et supposons que sa collection de noeuds est une instance `Vector <Node>` de la classe générique `Vector` qui modélise le concept `Collection`. On souhaite rendre cette collection extensible, afin qu'un algorithme de flot de coût minimal puisse ajouter un potentiel sur chaque noeud. L'intermédiaire peut être représenté par une classe générique `ExtensionManager` qui modélise également le concept `Collection` (cf. figure 5.9). Cette classe est paramétrée sur le type `C` de la collection dont elle est l'intermédiaire, ainsi que sur le type `T` des éléments de la collection.

Dans notre exemple, la collection de noeuds sera alors remplacée par un objet du type `ExtensionManager<Vector<Node>,Node>`. La classe `ExtensionManager` propose une interface pour gérer l'ajout et la suppression d'extensions sur les éléments de la collection associée. Notamment, un algorithme ajoutera une extension sur une collection en appelant la méthode `attach` de l'intermédiaire, avec comme argument un objet modèle qui sera cloné (cf. méthode `clone`) et placé sur chaque élément.

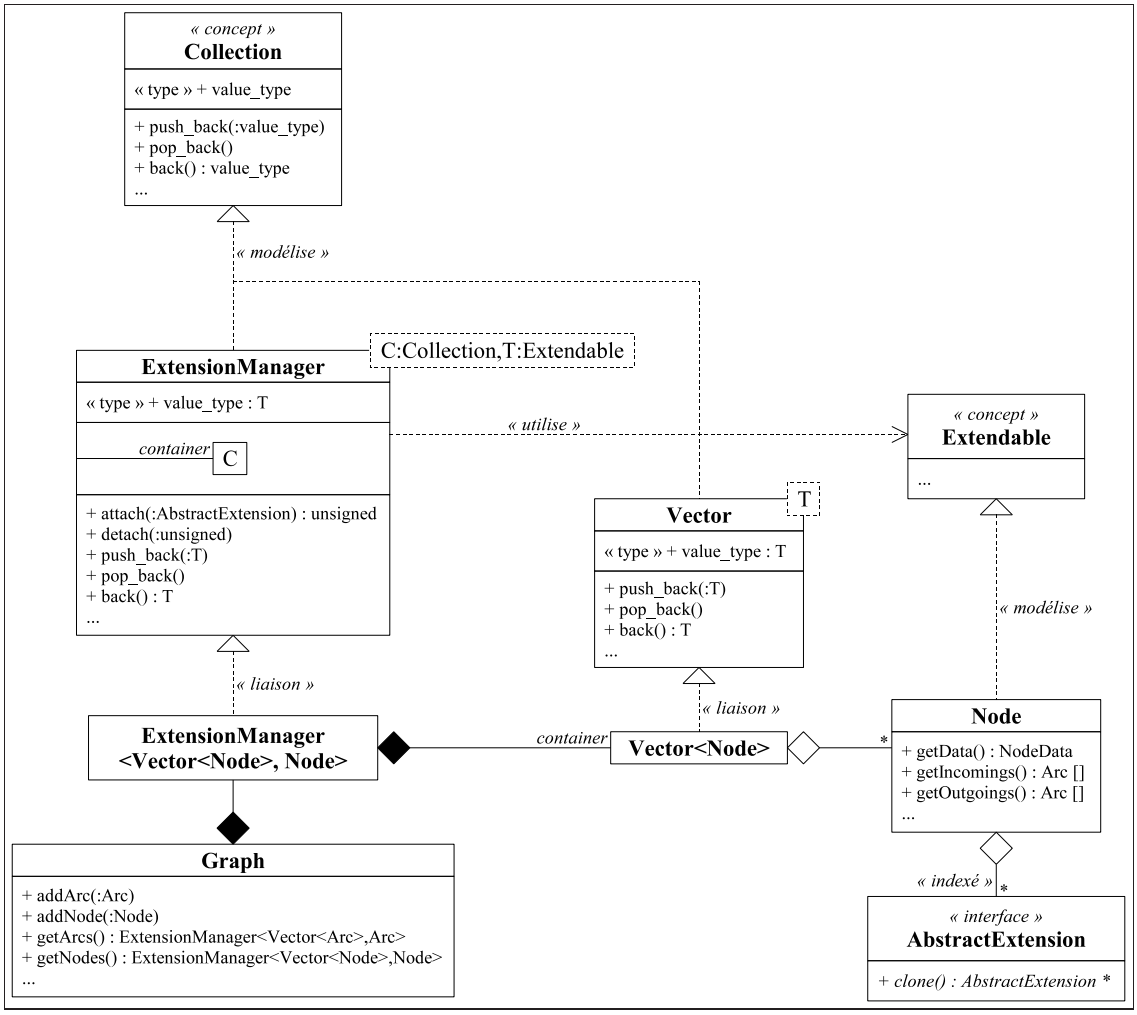

Figure 5.9: Extension d'une structure de données.

Les éléments doivent donc implémenter une interface spécifique (cf. concept `Extendable`) pour permettre la communication avec `ExtensionManager`. La méthode `attach` retourne alors l'indice où trouver l'extension nouvellement ajoutée dans la liste des extensions de chaque noeud. L'ajout d'un potentiel sur chaque noeud d'un graphe peut se faire de la manière suivante.

```
unsigned potentiel = graphe.getNodes().attach(Extension<double>(1e12));
```

5.4.3. Elément extensible

Plusieurs solutions sont envisageables pour implémenter le concept `Extendable`: (i) la classe `Node` peut modéliser le concept et déléguer la gestion des extensions à une autre classe (e.g. `ExtensionSet`, figure 5.10); (ii) le concept `Extendable` peut être modélisé par une classe `Extendable` qui gère les extensions (comme `ExtensionSet` dans la première solution), et la classe `Node` hérite de `Extendable` (cf. figure 5.11).

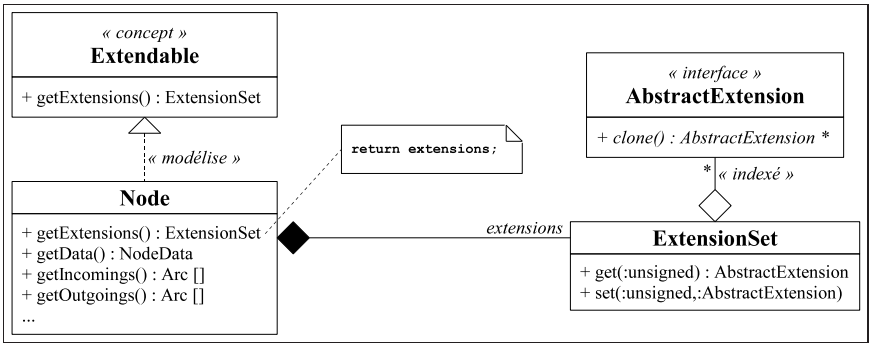

Figure 5.10: Elément extensible par délégation.

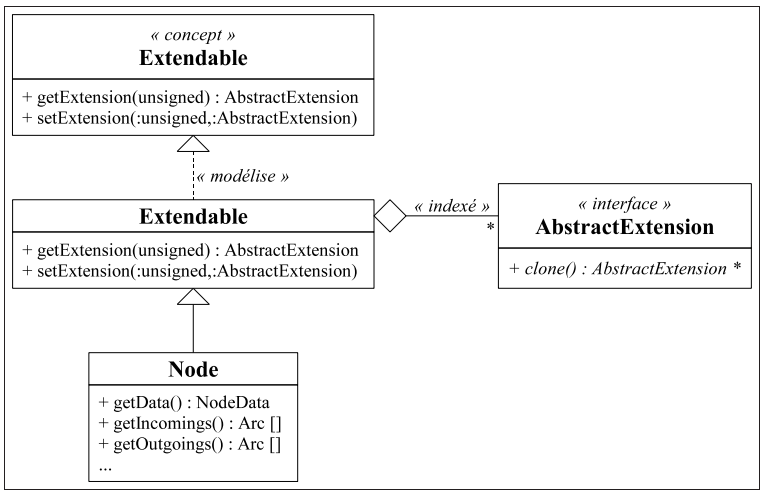

Figure 5.11: Elément extensible par héritage.

Dans les deux cas, aucune liaison dynamique n'est impliquée. L'impact de la seconde solution sur la définition de la classe `Node` est minimal, mais il empêche un éventuel héritage dans les langages où l'héritage multiple est impossible. Dans notre exemple, l'accès au potentiel d'un noeud se réalise de la manière suivante avec la seconde approche.

```
dynamic_cast<Extension<double> &>(node.getExtension(potentiel)).getValue();
```

Bien que la liaison dynamique ait été évitée, l'accès aux extensions nécessite une conversion à partir du type abstrait `AbstractExtension` pour qu'elles retrouvent leur type concret (e.g. `Extension<double>`). Cette conversion descendante nécessite une vérification à l'exécution qui risque de produire un surcoût important si l'accès aux extensions est fréquent. Comme présenté à la section 5.2, la conversion peut être forcée et la vérification abandonnée.

5.4.4. Performance

Dans notre exemple, l'algorithme de plus court chemin nécessite deux extensions: un potentiel (meilleure distance connue) et un marquage (indicateur de visite) sur chaque noeud du graphe. Dans l'implémentation de la section précédente, ces données sont des attributs de la classe `Node`, ce qui est très efficace mais établit une dépendance entre la structure de graphe et l'algorithme. Les tests (a) à (d) de la section précédente sont repris en modifiant les classes `Node` et `Arc` pour qu'ils modélisent le concept `Extendable` et la classe `Graph` afin qu'elle manipule les listes de noeuds et d'arcs par l'intermédiaire d'objets `ExtensionManager`.

Test	Sans extension	Avec extension
(a) Méthode patron	20.9 s	21.5 s
(b) Stratégie	20.9 s	21.5 s
(c) Politique	18.7 s	19.2 s
(d) Classique	18.7 s	19.2 s
(e) Politique avec conversion dynamique		24.8 s
(f) Politique avec conversion statique		19.8 s

Tableau 5.4: Influence du mécanisme d'extension d'une collection.

Le tableau 5.4, extrait de [Bach06a], montre l'impact de l'installation (i.e. sans utilisation) du mécanisme d'extension pour les tests (a) à (d). L'installation engendre un léger surcoût, dû principalement à l'augmentation de la taille des noeuds et des arcs en mémoire. Des tests utilisant pleinement les extensions des noeuds dans l'algorithme de plus court chemin sont ensuite présentés: le test (e) utilise une conversion dynamique (dont le mécanisme de vérification entraîne un surcoût d'environ 25 %) pour accéder à l'extension, alors que le test (f) utilise une conversion statique (qui n'affaiblit pas vraiment la robustesse du code puisque les extensions sont uniquement contrôlées par l'algorithme et que l'utilisateur n'intervient pas sur ces données).

En résumé, le mécanisme d'extension d'une collection entraîne, pour nos tests, un surcoût de moins de 3 % pour son installation, et d'environ 6 % lorsqu'il est sollicité. Ce surcoût semble acceptable pour obtenir un code beaucoup plus maintenable et évolutif. La solution d'extension présentée ici permet le remplacement d'une collection par un objet `ExtensionManager` sans modification du code manipulant la collection originale (e.g. `Graph`). La classe qui représente les éléments extensibles (e.g. `Node`) doit être légèrement modifiée: soit elle possède un attribut `ExtensionSet` et les accesseurs associées, soit elle hérite de la classe `Extendable`. Une fois le mécanisme installé, tout algorithme peut en bénéficier.

5.5. Conclusion

Ce chapitre montre qu'il est possible de concevoir des algorithmes génériques (au sens large) tout en maintenant une efficacité proche de celle d'un développement dédié. Pour obtenir un composant générique, deux conditions doivent être réunies: le composant doit être remplaçable (le maximum d'indépendance entre les composants doit être conservé) et extensible (aussi adaptable que possible à diverses situations).

Des techniques pour concevoir des structures de données et des algorithmes génériques ont été étudiées ici sur les aspects performance et réutilisabilité. Il ressort que les patrons de conception classiques utilisant la programmation objet doivent souvent être adaptés à l'aide de la programmation générique. L'héritage est un mécanisme pertinent pour abstraire un type de données, mais son utilisation combinée à des méthodes virtuelles peut conduire à des surcoûts importants à l'exécution. Si l'abstraction est statique (i.e. le type concret peut être déduit à la compilation), elle peut se faire par une paramétrisation, via programmation générique, du composant pour le type concerné. Des tests numériques ont montré sur des cas concrets d'algorithmes d'optimisation que l'utilisation de la généricité en C++ n'engendre pas de surcoût significatif à l'exécution.

Les techniques étudiées ici ont été exploitées dans le développement de trois projets de recherche ayant pour thèmes l'approximation neuronale [Duha03], la synchronisation hypermédia [Bach03a] et les réseaux de transport [Yon04a]. Le premier projet a permis de développer des techniques d'approximation de la qualité de service dans les réseaux de communication qui ont été intégrées à des modèles d'optimisation, en partenariat avec un opérateur téléphonique. Le deuxième projet, répondant à des problèmes de synchronisation dans la présentation de documents hypermédia, a notamment abouti au développement d'une bibliothèque logicielle de recherche opérationnelle [20] intégrée au système multimédia HyperProp (cf. chapitre 1). Le troisième projet a permis de développer des stratégies d'élaboration de tournées de bus dans un réseau urbain en vue d'optimiser la satisfaction des clients, en particulier grâce à un couplage optimisation-simulation (cf. chapitre 3).

Dans cette étude, nous avons utilisé la notion de "concept" en programmation générique, pour représenter de manière très abstraite les spécifications (on pourrait dire aussi les contraintes) que doit respecter un composant pour pouvoir être utilisé dans un contexte donné. Cette notion n'existe pas explicitement en langage C++, elle est éventuellement suggérée à travers une documentation du code. Cependant, elle est fondamentale pour consolider la fiabilité des codes et faciliter leur réutilisation (cf. chapitre 6). Des tentatives sont en cours pour intégrer cette notion dans une future norme du C++ [Greg07, Sutt13a, Sutt15].

[20] B++ Library: `http://www.nawouak.net/?doc=bpp_library`

CHAPITRE 6
MÉTAPROGRAMMATION ORIENTÉE CONCEPT

En programmation générique, les composants logiciels (appelés patrons) sont paramétrés sur des types. Dans un langage comme C++, un mécanisme de spécialisation statique permet de définir, pour certaines caractéristiques des paramètres, des versions plus adaptées d'un patron que sa version primaire. Le mécanisme normal de spécialisation de patron du C++ s'appuie sur les motifs de type des paramètres pour discriminer une spécialisation, ce qui n'est pas toujours pertinent. Les "concepts", qui définissent des spécifications, à la fois syntaxiques et sémantiques, applicables à des types, semblent plus adaptés pour servir de discriminants dans la spécialisation de patron.

Les concepts, qui présentent aussi un avantage indéniable pour la qualité du code, ne sont pas intégrés pour l'instant dans la norme du C++. Une bibliothèque reposant sur des techniques de métaprogrammation générique est donc proposée pour permettre la définition de spécialisations de patron utilisant les concepts comme discriminants. Cette forme de spécialisation statique se positionne alors comme une alternative solide à l'héritage lorsque la nature dynamique du polymorphisme associé n'est pas indispensable. Le potentiel de cette approche est illustré par une révision des patrons d'expressions (*expression templates*), une technique de métaprogrammation, qui propose une version statique orientée concept du célèbre patron de conception *visiteur*.

6.1. Programmation générique en C++

L'objectif de la programmation générique est de fournir des composants paramétrés sur des types de données, notamment des algorithmes et des structures de données, aussi généraux que possible, fortement adaptables et interopérables [Jaza98], et aussi efficaces que des composants non paramétrés. La programmation générique en C++ s'appuie sur la notion de composant générique, appelé aussi patron de composant (ou *template*), qui peut être aussi bien une classe, une fonction ou une méthode, avec des paramètres qui sont des types ou des valeurs entières statiques inconnus, à l'opposé des valeurs dynamiques que représentent les arguments d'une fonction ou d'une méthode (cf. section 5.1.2).

Avec les compilateurs modernes, l'utilisation de composants génériques n'entraîne que très rarement une perte d'efficacité, en raison d'un mécanisme d'instanciation où les paramètres d'un patron sont liés (i.e. associés à une valeur) à la compilation pour produire une version dédiée du patron pour chaque combinaison de valeurs des paramètres (cf. section 5.2). Le langage C++ propose aussi un mécanisme de spécialisation de patron qui joue un rôle essentiel en permettant d'assembler des composants à la compilation d'une manière optimale, e.g. en sélectionnant le code le plus approprié pour un algorithme en fonction des valeurs associées à ses paramètres.

6.1.1. Spécialisation de patron

A l'instar de l'héritage en programmation objet qui permet la spécialisation de classes, le langage C++ dispose d'un mécanisme pour spécialiser les composants génériques (e.g. [Vand03], chapitre 12). Considérons la version primaire d'une classe générique `ArrayComparator` permettant la comparaison de deux tableaux contenant chacun `N` éléments de type `T`.

```
template <class T,unsigned N> class ArrayComparator {
 public:
  static int run(const T * a,const T * b) {
   unsigned i = 0;
   while (i<N && a[i]==b[i]) ++i;
   return (i==N ? 0 : (a[i]<b[i] ? -1 : 1));
  }
};
```

On peut supposer que la comparaison de tableaux de caractères est plus efficace en utilisant la fonction standard `memcmp`. Par conséquent, une spécialisation du patron pour le cas `T = char` sera définie, en imposant que le paramètre `T` de la version primaire soit le type `char` dans la version spécialisée.

```
template <unsigned N> class ArrayComparator<char,N> {
 public:
  static int run(const char * a,const char * b) { return memcmp(a,b,N); }
};
```

Plus généralement, une spécialisation se caractérise par les contraintes qu'elle impose à ses paramètres. Dans le cas d'un paramètre représentant une valeur statique, il s'agit d'imposer une valeur, et dans le cas d'un paramètre représentant un type, il s'agit d'imposer un motif de type. Par motif de type, on entend un type (cf. l'exemple) ou un type paramétré (e.g. `T *` ou `vector<T>`) ou bien une classe générique (e.g. `template <class> class U`) appelée aussi paramètre *template template* [Wei01]. Au moment de l'instanciation, le compilateur sélectionne alors une version, primaire ou spécialisée, dont les contraintes sont respectées par la combinaison de valeurs liées aux paramètres. La version la plus contraignante, qui est aussi considérée la plus spécialisée, est alors retenue.

6.1.2. Métaprogrammation

Le mécanisme des *templates* s'est révélé bien plus puissant qu'il ne l'avait été envisagé lors de son introduction dans le langage C++. Il est en effet possible d'écrire de véritables programmes exécutables par le compilateur et il est d'ailleurs établi que les *templates* constituent un langage Turing-complet [Veld03]. Ils permettent donc une forme de métaprogrammation, i.e. d'écriture de code qui génère du code, appelée métaprogrammation générique (*template metaprogramming* [Abra04]), qui s'appuie sur une récursivité rendue possible grâce à la spécialisation de patron. Cela permet de nombreuses techniques: l'évaluation partielle [Veld99], les classes de traits [Myer96], les métafonctions [Veld96a, Abra04], les listes de types [Czar00a, Alex01], les patrons d'expressions [Veld96b]... Pour la compréhension de la suite du chapitre, nous détaillons plus particulièrement les métafonctions et les listes de types.

La métaprogrammation générique s'inscrit dans les principes de l'ingénierie des modèles (cf. section 4.2.5): un patron est un métamodèle et l'instanciation d'un patron correspond à la production d'une classe ou d'une fonction, i.e. d'un modèle conforme au métamodèle; un métaprogramme correspond à une forme élaborée de métamodèle où une procédure algorithmique décrit comment produire un modèle conforme; et enfin le modèle est compilé, i.e. transformé, pour produire un code exécutable.

6.1.2.1. Métafonctions

Plusieurs métafonctions élémentaires sont nécessaires à l'implémentation de notre bibliothèque, il s'agit de composants classiques des bibliothèques de métaprogrammation générique (e.g. *Boost Metaprogramming Library* - MPL). Une métafonction, représentée par une classe générique, est similaire à une fonction ordinaire, mais au lieu de manipuler des valeurs dynamiques, elle gère des données exploitables à la compilation, notamment des types et des valeurs statiques entières, appelées métadonnées [Abra04]. Afin de manipuler indifféremment des types et des valeurs statiques dans les métafonctions, les métadonnées sont embarquées dans des classes de la manière suivante [21].

```
template <class TYPE> struct gnx_type { typedef TYPE type; };

template <class TYPE,TYPE VALUE>
struct gnx_value { static const TYPE value = VALUE; };

typedef gnx_value<bool,true>  gnx_true;
typedef gnx_value<bool,false> gnx_false;
```

Le patron `gnx_type<T>` [22] représente le type `T` et possède un membre `type` qui est un alias de `T`. De la même manière, le patron `gnx_value<T,V>` représente une valeur statique `V` de type `T` et possède un attribut `value` égal à la valeur `V`. A partir du patron `gnx_value`, les types `gnx_true` et `gnx_false` sont définis pour représenter les valeurs booléennes.

Les paramètres d'une métafonction, qui sont les paramètres du patron représentant la métafonction, sont supposés être des métadonnées (i.e. des classes avec un membre `type` ou `value`). La "valeur de retour" d'une métafonction est implémentée par héritage: la métafonction hérite d'une classe représentant une métadonnée. De cette manière, la métafonction elle-même possède un membre `type` ou `value`, et peut donc devenir le paramètre d'une autre métafonction. La spécialisation de patron est souvent nécessaire pour implémenter l'algorithme d'une métafonction.

```
template <class TYPE1,class TYPE2> struct gnx_same : gnx_false {};

template <class TYPE> struct gnx_same<TYPE,TYPE> : gnx_true {};
```

Par exemple, la métafonction `gnx_same` détermine si deux types sont identiques: `gnx_same<T1,T2>` hérite de `gnx_true` si `T1` et `T2` sont le même type, ou de `gnx_false` sinon. Ainsi, la valeur retournée par la métafonction `gnx_same<T1,T2>` se trouve dans son attribut `value`.

[21] Le mot-clé `struct` est équivalent à `class` mais allège l'écriture grâce à des attributs et un héritage publics par défaut.
[22] Nous choisissons de préfixer toutes les métafonctions et macros de notre bibliothèque par `gnx_`.

```
template <class TEST,class THEN,class ELSE,bool = TEST::value>
struct gnx_if : ELSE {};

template <class TEST,class THEN,class ELSE>
struct gnx_if<TEST,THEN,ELSE,true> : THEN {};
```

La métafonction `gnx_if` est similaire à l'instruction `if` traditionnelle: `gnx_if<T,A,B>` hérite de `A` si `T::value` est vraie, ou de `B` sinon. Si `A` et `B` représentent des métadonnées, alors `gnx_if<T,A,B>` hérite du membre présent dans `A` ou `B`. A partir de cette métafonction, il est possible, par exemple, de définir l'opérateur logique *ou*.

```
template <class TYPE1,class TYPE2>
struct gnx_or : gnx_if<TYPE1,gnx_true,TYPE2> {};
```

Ainsi, déterminer si `X` est le type `int` ou `long` s'écrit comme suit.

```
gnx_or< gnx_same<X,int>,
        gnx_same<X,long> >
```

6.1.2.2. Listes de types

Pour les besoins de notre bibliothèque, il est nécessaire de stocker des types dans une collection qui puisse être manipulée à l'aide de métafonctions. Une technique commune consiste à définir une liste chaînée statique, appelée liste de types (*typelist* [Alex01, Czar00a]), de la manière suivante.

```
template <class CONTENT,class NEXT> struct gnx_list {
 typedef CONTENT content;
 typedef NEXT    next;
};

struct gnx_nil {};
```

Le type `gnx_nil` symbolise "aucun type" (`void` n'est généralement pas utilisé, car selon les applications, il peut être nécessaire de le stocker dans une liste) et sert à indiquer la fin de la liste. Par exemple, la liste composée des types `short`, `int` et `long` est définie comme suit.

```
typedef gnx_list< short,
                  gnx_list< int,
                            gnx_list<long,gnx_nil>
                          >
                > liste_t;
```

Des métafonctions peuvent être définies pour implémenter les opérations classiques des listes chaînées [Alex01]. Par exemple, la recherche d'un type dans une liste s'écrira de la manière suivante [23].

```
template <class ELEMENT,class LIST> struct gnx_search_list
: gnx_or< gnx_same<ELEMENT,typename LIST::content>,
          gnx_search_list<ELEMENT,typename LIST::next>
        > {};

template <class ELEMENT>
struct gnx_search_list<ELEMENT,gnx_nil> : gnx_false {};
```

[23] Le mot-clé `typename` est nécessaire pour affirmer que les membres `content` et `next` sont bien des types.

6.1.3. Les concepts

L'instanciation d'un patron soulève deux problèmes: (i) comment s'assurer qu'un type lié à un paramètre satisfait bien les spécifications du composant générique (e.g. tout type lié au paramètre `T` doit fournir les opérateurs `<` et `==` dans la classe `ArrayComparator`, cf. section 6.1.1); (ii) comment sélectionner la spécialisation d'un composant générique la plus appropriée pour une combinaison de valeurs donnée des paramètres (e.g. si le type `char` est lié au paramètre `T`, alors la spécialisation `ArrayComparator<char,N>` est sélectionnée, mais comment faire pour qu'un autre type puisse bénéficier de cette même spécialisation).

Pour répondre à ces problèmes, la notion de concept a été introduite [Aust99]. Un concept représente des exigences pour un type, en lui imposant des contraintes syntaxiques (i.e. sur son interface) et sémantiques (i.e. sur son comportement). Quand un type satisfait les spécifications d'un concept, on dit qu'il modélise le concept. La notion de spécialisation entre concepts est appelée affinement: un concept X affine un concept Y lorsqu'il inclut les spécifications de Y. Par exemple, définissons le concept `Entier` qui définit les spécifications d'un nombre entier, et le concept `Numerique` qui définit les spécifications de tout type de nombre. On peut établir que le type `int` modélise le concept `Entier` et que ce dernier affine le concept `Numerique`.

Les concepts se sont révélés être une notion complexe à intégrer au C++. Après une première tentative avortée [Greg07], une seconde proposition, *Concepts Lite* [Sutt13a], a été implémentée dans un prototype GCC [24] et publiée comme spécification technique ISO/IEC [Sutt15]. Cette extension du langage C++ ne définit qu'une partie des fonctionnalités des concepts, en permettant notamment l'utilisation de prédicats pour contraindre les paramètres des composants génériques. L'objectif à long terme est de proposer une définition complète des concepts à partir du retour d'expériences de cette première extension.

6.1.3.1. Vérification de concept

Les concepts servent donc à contraindre les paramètres d'un patron: lors de l'écriture d'un composant générique, les spécifications de chaque paramètre sont formulées par des concepts à modéliser. Lors de l'instanciation, une vérification peut être réalisée pour s'assurer que les types liés aux paramètres du patron satisfont les spécifications (i.e. qu'ils modélisent les concepts requis). Pour l'instant, les concepts ne peuvent pas être définis explicitement en C++, et n'apparaissent au mieux que dans la documentation associée aux composants génériques (e.g. *Standard Template Library*). Sans aucune vérification à l'instanciation d'un patron, certaines erreurs sont détectées très tard dans le processus de compilation, ce qui conduit souvent à des messages du compilateur difficilement compréhensibles [Siek00]: considérons par exemple l'instanciation `ArrayComparator<X,10>`, si le type `X` n'a pas d'opérateur `<`, alors l'erreur sera détectée au niveau de la méthode `run`, et non au niveau de l'instanciation.

[24] `http://concepts.axiomatics.org/~ans/`

Dans certains langages, des mécanismes spécifiques sont utilisés pour supporter la vérification de concept [Garc03]. En Java notamment, les interfaces servent à contraindre les paramètres d'un patron, mais cette approche limite les concepts à des spécifications syntaxiques [Bracha04] et impose d'anticiper les relations entre classes et concepts au moment de la conception des classes. Les concepts doivent permettre plus de flexibilité: une classe doit pouvoir modéliser un concept soit implicitement parce qu'elle satisfait automatiquement toutes les spécifications d'un concept (cf. *auto concept* [Greg06b]), soit explicitement en déclarant *a posteriori* la relation de modélisation avec le concept et en décrivant comment elle satisfait les spécifications (cf. *concept map* [Greg06b]).

La bibliothèque BCCL (*Boost Concept Check Library* [Siek00]) offre des possibilités de vérification de concept en C++. Nous l'avons utilisée pour la conception d'un *framework* de génération de séquences parallèles de nombres pseudo-aléatoires sur GPGPU (*General-Purpose Graphics Processor Units*), appelé *ShoveRand* [Pass11]. Son rôle est de permettre le déploiement fiable sur un environnement parallèle de générateurs de nombres pseudo-aléatoires définis à partir de trois éléments constituants (cf. figure 6.1): le paramétrage du générateur (classe `ParameterizedStatus`), son état interne (classe `SeedStatus`) et l'algorithme d'évolution de la séquence (e.g. classe `MersenneAlgo`). Les deux premiers sont modélisés par des classes paramétrées sur le type de l'algorithme, et le dernier est représenté par une classe paramétrée sur le type des valeurs à produire et doit modéliser le concept `RNGAlgo`.

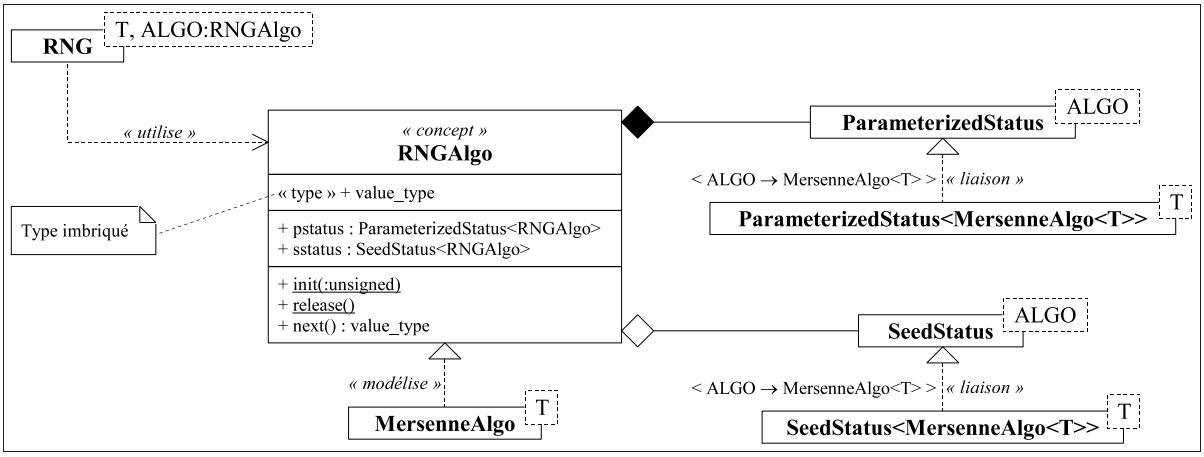

Figure 6.1: Architecture de ShoveRand.

Avec la bibliothèque BCCL, le concept `RNGAlgo` est formalisé à l'aide d'une classe générique dont l'unique paramètre `ALGO` représente un type testé pour sa conformité avec le concept (cf. code ci-après). La vérification consiste à manipuler le type `ALGO` à travers tous les éléments de l'interface imposée par le concept `RNGAlgo`. Dans la définition du concept, la macro `BOOST_CONCEPT_USAGE` sert à identifier la méthode qui contient la séquence de test. Elle vérifie notamment que les méthodes `init`, `release` et `next` du type `ALGO`, ainsi que ses attributs `pstatus` et `sstatus` existent et sont conformes aux spécifications du concept.

```
template <class ALGO> struct RNGAlgo {
 BOOST_CONCEPT_USAGE(RNGAlgo) {
  ALGO                      algo;
  typename ALGO::value_type value;
  ParameterizedStatus<ALGO> pstatus;
  SeedStatus<ALGO>          sstatus;

  ALGO::init(27);
  ALGO::release();

  same_type(value,algo.next());
  same_type(pstatus,algo.pstatus);
  same_type(sstatus,algo.sstatus);
 }

 template <class U> void same_type(const U &,const U &) {}
};
```

Pour déclencher effectivement une vérification, e.g. s'assurer que le type `MersenneAlgo` modélise le concept `RNGAlgo`, l'instanciation `RNGAlgo<MersenneAlgo>` doit être réalisée: toute erreur à ce niveau signifie que le type ne modélise pas le concept. Lors de la définition de la classe générique `RNG` qui représente un générateur de nombres pseudo-aléatoires paramétré sur le type `T` des valeurs produites et sur le type `ALGO` de l'algorithme employé, la macro `BOOST_CONCEPT_ASSERT` sert à s'assurer que le type `ALGO` modélise le concept `RNGAlgo` (en réalisant l'instanciation `RNGAlgo<ALGO>`).

```
template <class T,class ALGO> class RNG {
 [...]
 BOOST_CONCEPT_ASSERT((RNGAlgo<ALGO>));
 [...]
};
```

6.1.3.2. Spécialisation de patron

Les concepts servent aussi à contrôler la spécialisation d'un patron à la place des motifs de type, car ces derniers peuvent conduire à des ambiguïtés (le compilateur ne peut pas décider entre deux versions possibles du patron) ou à de mauvaises spécialisations (le compilateur sélectionne une version du patron inadaptée). En outre, le motif de type n'est pas toujours pertinent pour discriminer une spécialisation: comment s'assurer que deux types "similaires" (i.e. qui ont certaines fonctionnalités communes), avec des motifs de type différents, conduisent à la même spécialisation lorsqu'ils sont liés au même paramètre.

Nous proposons d'illustrer certaines de ces difficultés avec la conception d'une classe générique `Serializer` qui permet de mémoriser l'état d'un objet dans un tableau d'octets (opération de "sérialisation", méthode `deflate`) ou de restaurer l'état d'un objet à partir d'un tableau d'octets (opération de "désérialisation", méthode `inflate`). La version primaire du patron, qui réalise une copie bit-à-bit d'un objet en mémoire, est définie de la manière suivante.

```
template <class T> struct Serializer {
 static int deflate(char * copy,const T & object);
 static int inflate(T & object,const char * copy);
};
```

Cette version ne convient pas toujours aux objets complexes, comme les collections, où l'état interne peut contenir des pointeurs qui ne doivent pas être mémorisés (les versions bit-à-bit de sérialisation et désérialisation conduiraient à une incohérence en mémoire après la restauration). Définissons une version spécialisée de `Serializer` pour les conteneurs en séquence de la STL (*Standard Template Library*), e.g. les vecteurs (`std::vector`) et les listes (`std::list`).

```
template <class T,class ALLOC,template <class,class> class CONTAINER>
struct Serializer< CONTAINER<T,ALLOC> > {
 static int deflate(char * copy,const CONTAINER<T,ALLOC> & container);
 static int inflate(CONTAINER<T,ALLOC> & container,const char * copy);
};
```

Dans cette spécialisation, le paramètre `T` de la version primaire devient le paramètre `CONTAINER` qui est contraint par le motif de type des conteneurs en séquence: ce sont des classes génériques avec deux paramètres, le type `T` des éléments stockés et le type `ALLOC` de l'objet utilisé pour allouer les éléments. Considérons maintenant les conteneurs associatifs de la STL, e.g. les ensembles (`std::set`) et les associations (`std::map`). Leur motif de type est différent de celui des conteneurs en séquence (ils possèdent au moins un paramètre supplémentaire `COMP` pour comparer les éléments), alors qu'ils ont en commun certaines opérations dans leur interface qui leur permettrait d'utiliser la version spécialisée des conteneurs en séquence. Mais comme le mécanisme de spécialisation repose sur les motifs de type, une nouvelle version du patron `Serializer` est nécessaire.

```
template <class T,class COMP,class ALLOC,
          template <class,class,class> class CONTAINER>
struct Serializer< CONTAINER<T,COMP,ALLOC> > { [...] };
```

Remarquons que cette spécialisation de `Serializer` n'est valide que pour les ensembles, et non pour les associations car leur motif de type est différent: `std::map` possède un paramètre additionnel `K` pour le type des clés associées aux éléments du conteneur. Une nouvelle spécialisation est encore une fois nécessaire pour les associations, alors qu'elles partagent des opérations communes avec les ensembles. La spécialisation pour les ensembles a été conçue en pensant uniquement aux conteneurs de la STL, alors que tout type qui est conforme au motif de type pourrait être associé involontairement à cette version du patron `Serializer`. Par exemple, la classe `std::string` de la bibliothèque standard du C++ est l'alias d'un type qui correspond au motif de type des ensembles:

```
std::basic_string< char,std::char_traits<char>,std::allocator<char> >
```

Les différents problèmes évoqués ici peuvent être réglés à l'aide des concepts. Des types avec des fonctionnalités communes peuvent modéliser le même concept pour lequel une version spécialisée d'un patron sera définie. Ainsi, des types similaires avec des motifs de type différents peuvent être associés à la même spécialisation. Les concepts peuvent aussi éviter les spécialisations inadaptées: avec une spécialisation orientée concept, tout paramètre d'un patron est contraint par un concept, et uniquement les types qui modélisent ce concept peuvent être liés au paramètre. De cette manière, seuls les types qui satisfont les exigences d'une spécialisation sont considérés.

Une taxonomie de concepts, i.e. des concepts et leurs relations, est donc définie pour notre exemple de sérialisation (cf. figure 6.2). Les concepts `SingleObject` et `STLContainer` sont définis pour permettre deux spécialisations de `Serializer`: l'une reposant sur une copie bit-à-bit et l'autre reposant sur des opérations communes à tous les conteneurs STL. Comme les conteneurs en séquence et associatifs sont de natures différentes, on peut envisager des manières distinctes d'optimiser la sérialisation. Pour cette raison, le concept `STLContainer` est affiné en deux concepts `STLSequence` et `STLAssociative` pour permettre des spécialisations de `Serializer` adaptées respectivement aux conteneurs en séquence et aux conteneurs associatifs.

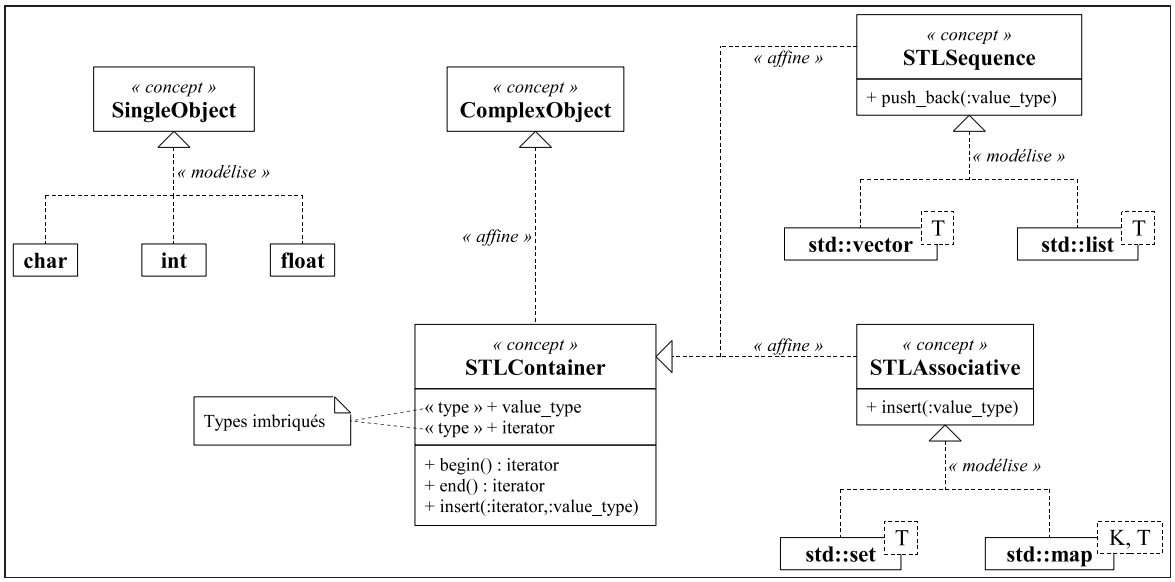

Figure 6.2: Taxonomie de concepts pour l'exemple de sérialisation.

Des solutions existent pour la spécialisation de patron orientée concept en C++ [Jarv03, McNa00], mais elles ne considèrent pas les relations d'affinement entre concepts qui sont pourtant nécessaires pour départager plusieurs versions candidates d'un patron lors d'une instanciation en sélectionnant celle qui est la plus adaptée. Par exemple, le type `vector<int>` modélise à la fois les concepts `STLContainer` et `STLSequence`, mais lors de l'instanciation `Serializer<vector<int>>`, seule la version la plus spécialisée du patron doit être retenue, i.e. celle associée au concept `STLSequence`.

6.2. Spécialisation orientée concept

En attendant que les concepts fassent partie de la norme C++, nous avons proposé une solution pour représenter partiellement les concepts et rendre possible la spécialisation de patron orientée concept [Bach13]. Dans un souci de portabilité, notre solution se présente sous la forme d'une bibliothèque générique compatible C++03 ne nécessitant aucun autre outil que le compilateur.

Cette bibliothèque, appelée C4TS++ [25] (*Concepts for Template Specialization in C++*), repose sur des techniques de métaprogrammation générique, et n'utilise des macros qu'en frontal pour fournir une syntaxe allégée à l'utilisateur. La version présentée ici exploite aussi quelques éléments de C++11 pour simplifier encore la syntaxe, mais le noyau du code reste celui introduit dans [Bach13].

La bibliothèque C4TS++ fournit une syntaxe pour déclarer des concepts, ainsi que des relations de modélisation et d'affinement. A partir de ces déclarations, la spécialisation de patron orientée concept peut être réalisée. Les concepts servent alors à contraindre les paramètres d'un patron dans la définition d'une spécialisation. Ensuite, lors d'une instanciation, la version la plus appropriée du patron est automatiquement sélectionnée sur la base des concepts modélisés par les types liées aux paramètres du patron: un métaprogramme détermine, pour chaque type, son concept le plus spécialisé parmi ceux utilisés pour contraindre le paramètre associé. La structure proposée reste ouverte aux extensions: la taxonomie peut être complétée et de nouvelles spécialisations peuvent être définies n'importe où dans le code.

6.2.1. Syntaxe proposée

Nous reprenons l'exemple du patron `Serializer` pour illustrer l'utilisation de la bibliothèque C4TS++. Les concepts doivent avant tout être déclarés, à l'aide de la macro `gnx_declare_concept`.

```
gnx_declare_concept(SingleObject);
gnx_declare_concept(ComplexObject);
gnx_declare_concept(STLContainer);
gnx_declare_concept(STLSequence);
gnx_declare_concept(STLAssociative);
```

Les relations de modélisation et d'affinement peuvent ensuite être ajoutées, à l'aide de la macro `gnx_add_models`. Notons que l'instruction permet de déclarer indifféremment les deux types de relations.

```
template <> gnx_add_models(char,SingleObject);
template <> gnx_add_models(int,SingleObject);
template <> gnx_add_models(float,SingleObject);

template <class T>         gnx_add_models(std::vector<T>,STLSequence);
template <class T>         gnx_add_models(std::list<T>,STLSequence);
template <class T>         gnx_add_models(std::set<T>,STLAssociative);
template <class K,class T> gnx_add_models(std::map<K,T>,STLAssociative);

template <> gnx_add_models(STLContainer,ComplexObject);
template <> gnx_add_models(STLSequence,STLContainer);
template <> gnx_add_models(STLAssociative,STLContainer);
```

A partir de maintenant, une taxonomie de concepts est définie, mais elle peut être étendue à tout moment par des déclarations de concepts et/ou de relations (à condition de suivre quelques règles pour éviter "l'effet Schrödinger" [26] des *templates* [Bach15]).

[25] `http://forge.clermont-universite.fr/projects/cpp-concepts`
[26] Métaphore reprise de: `http://www.codeproject.com/Articles/776770/Automatic-Static-Counter`

Définir des versions spécialisées d'un patron à l'aide de concepts est simple, mais la version primaire du patron doit être préparée. D'abord, nous avons besoin d'un identifiant, appelé contexte de spécialisation, pour chaque définition de patron avec spécialisations. La raison est que notre mécanisme a besoin de connaître tous les concepts qui sont impliqués dans les spécialisations d'un patron donné; comme l'analyse de code n'est pas possible avec une approche reposant uniquement sur le langage C++, la solution consiste à déclarer explicitement les concepts auprès du contexte de spécialisation. Ce contexte doit être statique, il peut s'agir d'un type existant ou bien déclaré spécialement pour l'occasion.

```
struct SerializerContext;
```

Ensuite, pour chaque paramètre `T` qui sera contraint par des concepts dans les spécialisations, un paramètre additionnel `C` est nécessaire dans le patron pour représenter le concept modélisé par `T` le plus spécialisé dans le contexte de spécialisation (la valeur de `C` sera automatiquement déduite par la métafonction `gnx_best_concept_t`).

```
template < class T,
           class = gnx_best_concept_t<SerializerContext,T> >
struct Serializer;
```

Le patron est maintenant prêt pour la spécialisation orientée concept: le paramètre additionnel peut être contraint par n'importe quel concept pour définir une version spécialisée du patron. Il faut seulement déclarer le concept auprès du contexte de spécialisation, à l'aide de la macro `gnx_add_uses`.

```
template <> gnx_add_uses(SerializerContext,SingleObject);

template <class T> struct Serializer<T,SingleObject> { [...] };

template <> gnx_add_uses(SerializerContext,STLContainer);

template <class T> struct Serializer<T,STLContainer> { [...] };

[...]
```

De nouveaux concepts et/ou relations peuvent être ajoutés à tout moment. Ils seront automatiquement considérés dans le processus de spécialisation tant que le patron n'a pas été instancié [Bach15].

6.2.2. Indexation des concepts

Nous choisissons de représenter les concepts par des types: la macro `gnx_declare_concept` définit une classe vide pour déclarer un concept. Les concepts doivent être stockés dans une liste statique, afin de permettre leur manipulation par des métafonctions, notamment pour parcourir la taxonomie de concepts. Malheureusement, la liste de types présentée à la section 6.1.2.2 est trop statique pour nos besoins: les opérations de modification d'une liste de types produisent une nouvelle liste, i.e. un nouveau type [Bach13]. Nous avons besoin d'une solution qui puisse véritablement modifier la liste pour ajouter de nouveaux concepts à tout moment. Les listes de types restent néanmoins utiles pour certaines métafonctions où des opérations de recherche et de fusion de listes sont nécessaires.

Pour modéliser la liste des concepts, nous proposons de les indexer à l'aide de la métafonction `gnx_concept`. Cette dernière reçoit comme paramètre une valeur entière et retourne le concept associé à ce nombre. L'ajout d'un nouveau concept à la liste consiste à définir une spécialisation de la métafonction.

```
template <int ID> struct gnx_concept : gnx_type<gnx_nil> {};

template <> struct gnx_concept<1> : gnx_type<STLContainer> {};
template <> struct gnx_concept<2> : gnx_type<STLSequence> {};
[...]
```

Indexer les concepts manuellement n'est pas acceptable, une solution pour obtenir le nombre de concepts déjà présents dans la liste est nécessaire. Pour cela, une version préliminaire de la métafonction `gnx_nb_concept` est proposée. Elle parcourt tous les concepts de la liste en incrémentant un index jusqu'à trouver `gnx_nil` comme concept associé.

```
template <int N = 0> struct gnx_nb_concept
: gnx_if< gnx_same<typename gnx_concept<N+1>::type,gnx_nil>,
          gnx_value<int,N>,
          gnx_nb_concept<N+1>
        > {};
```

Dans l'objectif d'une indexation automatique des concepts, on souhaiterait utiliser la valeur de retour de la métafonction `gnx_nb_concept` pour déterminer l'index du prochain concept déclaré.

```
template <> struct gnx_concept<gnx_nb_concept<>::value+1>
: gnx_type<STLContainer> {};
```

L'invocation `gnx_nb_concept<>` induit l'instanciation du patron `gnx_concept` de `gnx_concept<0>` à `gnx_concept<N+1>`, où `N` est le nombre de concepts indexés. La spécialisation de `gnx_concept` telle qu'elle est présentée dans l'exemple précédent n'est donc pas possible, car `gnx_concept<N+1>` est déjà instancié avec la version primaire de `gnx_concept`. Pour éliminer ce défaut, un paramètre additionnel, appelé observateur, est ajouté aux deux métafonctions `gnx_concept` et `gnx_nb_concept`.

```
template <int ID,class OBS = gnx_nil>
struct gnx_concept : gnx_type<gnx_nil> {};

template <class OBS,int N = 0> struct gnx_nb_concept
: gnx_if< gnx_same<typename gnx_concept<N+1,OBS>::type,gnx_nil>,
          gnx_value<int,N>,
          gnx_nb_concept<OBS,N+1>
        > {};
```

Le principe est de fournir un observateur différent à chaque fois que les concepts doivent être comptés pour déterminer l'index du prochain concept déclaré: le nouveau concept lui-même sera l'observateur. Avec cette solution, compter les concepts avec l'observateur `OBS` implique l'instanciation de `gnx_concept<N+1,OBS>`, donc toute spécialisation pour l'index `N+1` avec un observateur autre que `OBS` reste possible. Finalement, les concepts sont indexés comme suit.

```
template <class OBS>
struct gnx_concept<gnx_nb_concept<STLContainer>::value+1,OBS>
: gnx_type<STLContainer> {};
```

Pour déclarer un concept en une simple instruction, comme présenté dans l'exemple en début de section, la macro `gnx_declare_concept` est définie de la manière suivante.

```
#define gnx_declare_concept(CONCEPT)                          \
 struct CONCEPT {};                                           \
                                                              \
 template <class OBS>                                         \
 struct gnx_concept<gnx_nb_concept< CONCEPT >::value+1,OBS> \
 : gnx_type< CONCEPT > {}
```

Pour conclure, la métafonction `gnx_nb_concept` requiert $O(n)$ opérations, où n est le nombre de concepts déjà déclarés dans le programme. Ainsi, à la compilation, indexer n concepts nécessite $O(\sum_{i=1}^{n} i) = O(n^2)$ opérations.

6.2.3. Analyse d'une taxonomie

La représentation des relations de modélisation et d'affinement qui définissent une taxonomie, ainsi que plusieurs métafonctions permettant d'analyser une taxonomie, sont présentées. Les relations de modélisation, entre un type et un concept, et les relations d'affinement, entre deux concepts, sont déclarées indifféremment à l'aide de la métafonction `gnx_models_concept`.

```
template <class TYPE_OR_CONCEPT,class CONCEPT>
struct gnx_models_concept : gnx_false {};
```

La version primaire du patron retourne faux et les relations sont déclarées en spécialisant le patron: si le type `X` modélise le concept `C` (ou le concept `X` affine le concept `C`), alors la spécialisation `gnx_models_concept<X,C>` doit retourner vrai.

```
template <> struct gnx_models_concept<X,C> : gnx_true {};
```

Pour déclarer simplement une relation, comme présenté en début de section, la macro `gnx_add_models`, valide uniquement en C++11, est définie de la manière suivante.

```
#define gnx_add_models(...) \
 struct gnx_models_concept< __VA_ARGS__ > : gnx_true {}
```

Notons que `gnx_models_concept` fournit une réponse pour une relation directe uniquement. Si un type `T` modélise un concept `C1` qui affine un concept `C2`, la métafonction retourne faux pour une relation entre `T` et `C2`. Plusieurs métafonctions supplémentaires, nécessaires pour trouver toute relation entre un type et un concept (ou entre deux concepts), sont brièvement décrites ci-après.

La métafonction `gnx_direct_concepts<X>` fournit une liste (plus précisément une liste de types) de tous les concepts directement modélisés par un type (ou affinés par un concept) `X`. Elle parcourt la liste indexée des concepts déclarés et vérifie si `X` modélise (ou affine) chaque concept à l'aide de la métafonction `gnx_models_concept`. En supposant qu'obtenir un concept à partir de son index (i.e. appeler la métafonction `gnx_concept`) est une opération en temps constant, la métafonction `gnx_direct_concepts` requiert $O(n)$ opérations, où n est le nombre de concepts déclarés dans le programme.

La métafonction `gnx_all_concepts<X>` fournit une liste de tous les concepts directement ou indirectement modélisés par un type (ou affinés par un concept) `X`. Elle appelle `gnx_direct_concepts` pour lister les concepts directs de `X`, et récupère récursivement tous les concepts associés à chacun des concepts directs. Cette métafonction requiert $O(n^2 + rn)$ opérations, où r est le nombre de relations de modélisation et d'affinement dans le programme: au pire, les n concepts sont analysés (i.e. n appels à la métafonction `gnx_direct_concepts`), ce qui requiert $O(n^2)$ opérations; et au pire, r relations sont détectées, avec chaque fois la fusion de la liste des concepts trouvés précédemment avec celle des concepts nouvellement trouvés, ce qui requiert $O(rn)$ opérations (au pire $2n$ opérations sont nécessaires pour la fusion, car elle évite les doublons).

La métafonction `gnx_matches_concept<X,C>` indique si un type (ou un concept) `X` modélise (ou affine) un concept `C`, directement ou indirectement. Cette métafonction cherche `C` dans la liste des concepts fournie par la métafonction `gnx_all_concepts`, ce qui requiert $O(n^2 + rn)$ opérations: $O(n^2 + rn)$ pour construire la liste et $O(n)$ pour la recherche.

6.2.4. Sélection d'une spécialisation

Le mécanisme de spécialisation contrôlée par des concepts, et notamment la manière de sélectionner la version la plus adaptée d'un patron lors d'une instanciation, sont détaillés. Comme expliqué dans l'exemple en début de section, la spécialisation orientée concept d'un patron nécessite une préparation: définir un contexte de spécialisation (cf. `SerializerContext`) unique pour chaque patron, et ajouter un paramètre supplémentaire `C` pour chaque paramètre original `T` du patron qui sera contraint par des concepts dans les spécialisations.

```
template < class T,
           class C = gnx_best_concept_t<SerializerContext,T> >
struct Serializer;
```

Le paramètre `C` représente le concept modélisé par `T` le plus spécialisé dans le contexte de spécialisation. Sa valeur est déduite par la métafonction `gnx_contextual_concept` dont l'appel est simplifié grâce à l'alias `gnx_best_concept_t` (au lieu d'une macro en C++03). Le fait d'attribuer une valeur par défaut à `C` le rend invisible lors de l'utilisation du patron et automatise l'appel à `gnx_best_concept_t`.

```
template <class CONTEXT,class TYPE> using gnx_best_concept_t
= typename gnx_contextual_concept<CONTEXT,TYPE>::type;
```

Le contexte de spécialisation, qui est un paramètre de la métafonction `gnx_contextual_concept`, joue un double rôle. Tout d'abord, il sert d'observateur pour appeler la métafonction `gnx_nb_concept` utilisée par les métafonctions qui analysent une taxonomie, elles-mêmes appelées par la méthode `gnx_contextual_concept`. Ensuite, les concepts servant à contraindre des paramètres dans les spécialisations du patron doivent être déclarés auprès du contexte de spécialisation, à l'aide de la macro `gnx_add_uses` qui simplifie la manipulation de la métafonction `gnx_uses_concept` conçue sur le même principe que `gnx_models_concept`.

A partir de la liste des concepts déclarés dans le contexte de spécialisation et d'une taxonomie de concepts, la métafonction `gnx_contextual_concept` détermine le "meilleur" concept pour un type lié au paramètre `T`, i.e. le concept modélisé le plus spécialisé parmi ceux utilisés pour contraindre le paramètre.

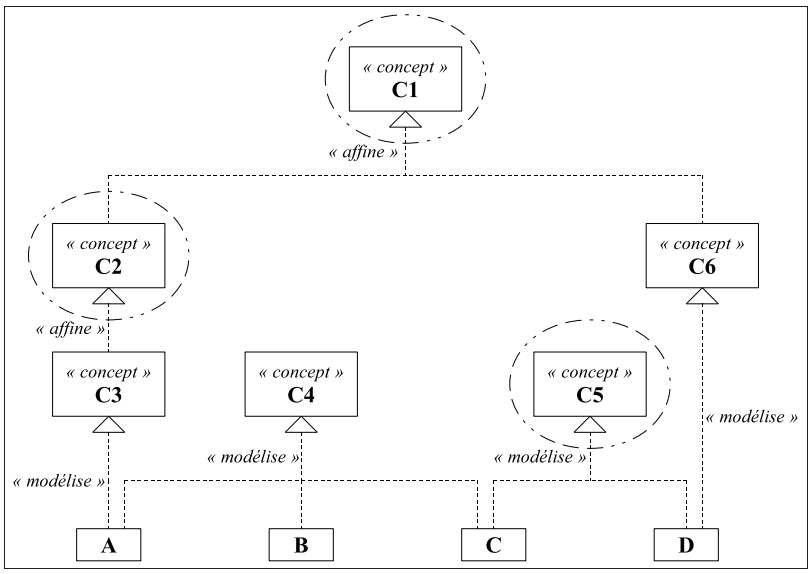

Figure 6.3: Exemple de sélection du meilleur concept.

Considérons la taxonomie de concepts de la figure 6.3 et un contexte `X` qui propose des spécialisations pour les concepts `C1`, `C2` et `C5` de la figure. Les concepts suivants doivent être sélectionnés.

- Pour le type `A`: le concept `C2`, car les candidats sont `C1` et `C2`, mais `C2` est plus spécialisé.
- Pour le type `B`: aucun concept, la valeur `gnx_nil` est retournée.
- Pour le type `C`: le concept `C5`, car c'est l'unique candidat.
- Pour le type `D`: le concept `C1` ou `C5`, car les deux sont des candidats valides, mais aucune relation entre eux n'existe pour déterminer lequel est le plus spécialisé. Celui qui est sélectionné dépend de l'implémentation de `gnx_contextual_concept` (dans notre cas, celui avec l'index le plus élevé). Mais pour éviter tout choix arbitraire, il est possible d'ajouter des relations dans la taxonomie, ou bien de spécialiser la métafonction `gnx_contextual_concept` pour le type `D` dans le contexte `X`.

La métafonction `gnx_contextual_concept<X,T>` parcourt la liste de tous les concepts modélisés directement ou indirectement par le type `T` (fournie par `gnx_all_concepts<T>`), et sélectionne celui qui n'affine pas directement ou indirectement tout autre concept de la liste (à l'aide de la métafonction `gnx_matches_concept`) et qui est déclaré dans le contexte `X`. Cette métafonction requiert $O(n^2 + rn)$ opérations: $O(n^2 + rn)$ pour construire la liste et $O(n)$ pour sélectionner le meilleur candidat (car `gnx_all_concepts` a déjà réalisé toutes les instanciations de `gnx_matches_concept` nécessaires).

6.2.5. Performances de compilation

La complexité des métafonctions établie précédemment suppose que certaines opérations de compilation sont en temps constant, nous avons donc étudié leur comportement en pratique. Des programmes C++ ont été générés aléatoirement pour manipuler différentes quantités de concepts n et de relations de modélisation et d'affinement r. Les temps de compilation présentés ici sont exprimés en secondes et représentent une moyenne pour 10 programmes différents de même taille (i.e. avec les mêmes quantités n et r). Ces résultats sont extraits de [Bach13].

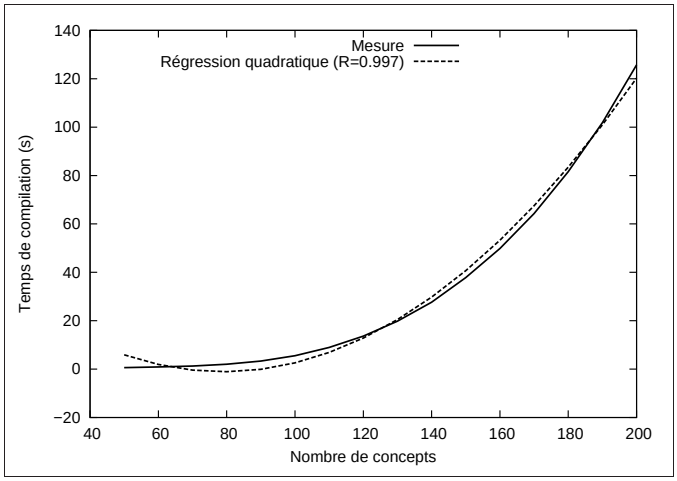

Figure 6.4: Temps de compilation pour l'indexation des concepts ($r = 100$).

La figure 6.4 montre le temps nécessaire pour indexer n concepts. Comme prévu, la dépendance à n est quadratique (confirmée par une régression quadratique avec un coefficient de corrélation [27] $R = 0.997$).

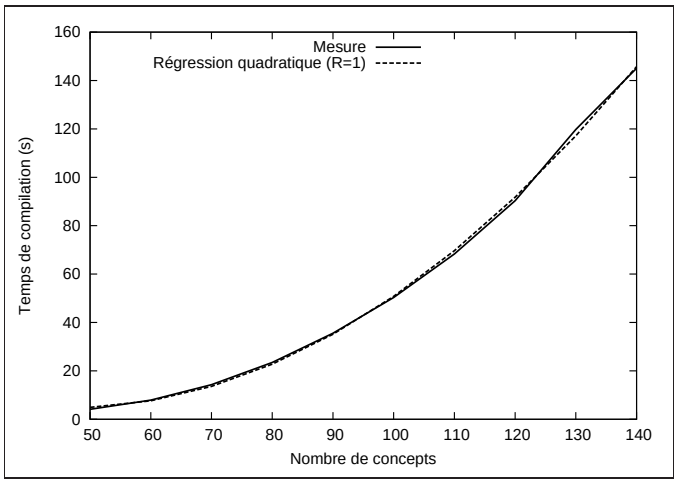

Figure 6.5: Temps de compilation pour la sélection d'une spécialisation (50 instanciations, $r = 300$).

[27] R = coefficient de corrélation de Pearson; plus il est proche de 1, plus la régression est conforme à la courbe.

La figure 6.5 montre le temps nécessaire pour sélectionner une spécialisation lors de l'instanciation d'un patron, en fonction du nombre n de concepts dans le programme. Les valeurs présentées cumulent les temps de 50 instanciations différentes, i.e. de 50 appels différents à la métafonction `gnx_contextual_concept`. Comme prévu, la dépendance à n est quadratique (confirmée avec $R = 1$). Une dépendance linéaire à r est aussi constatée (confirmée avec $R = 0.989$, résultats non montrés [Bach13]).

6.3. Patrons d'expressions avec concepts

Afin d'illustrer quelques-unes des possibilités offertes par les concepts, nous proposons d'étudier la conception de langages dédiés embarqués (*Embedded Domain Specific Languages* - EDSL [Fowl10]) à l'aide de la spécialisation orientée concept. En C++, les EDSL reposent sur la surcharge d'opérateurs pour proposer un langage adapté à un domaine spécifique, e.g. l'algèbre linéaire avec des opérations pour les matrices et les vecteurs. Outre l'aspect syntaxique, la technique sous-jacente appelée patrons d'expressions (*expression templates*) permet d'accélérer l'évaluation d'une expression (elle permet notamment d'évaluer une expression en une simple passe qui évite les objets temporaires) [Veld96b]. L'évaluation consiste à parcourir la structure de l'expression et à visiter chacune de ses opérations et opérandes pour lui appliquer une action spécifique. Nous proposons une approche générique pour l'évaluation qui repose sur une version statique du patron de conception *visiteur* [Gamm95] utilisant la spécialisation orientée concept.

6.3.1. Modélisation classique

Les patrons d'expressions ont été introduits par [Veld96b] et [Vand03] pour représenter une expression sous la forme d'un objet, en utilisant des patrons pour construire le type de l'objet. L'objectif principal est de résoudre certains problèmes de performance qui peuvent survenir lors de la surcharge d'opérateurs. La structure d'une expression est représentée par une composition récursive de types qui modélise un arbre syntaxique abstrait: une expression est une opération sur des opérandes qui sont des expressions. Avec C++11, la composition peut être modélisée par une simple classe générique `Expression` avec un paramètre qui représente l'opérateur caractérisant l'opération et un *pack* de paramètres (i.e. un ensemble indéfini de paramètres, cf. *variadic templates* [28]) qui représente les opérandes.

```
template <class OPERATOR,class... OPERANDS> struct Expression {
 std::tuple<const OPERANDS &...> operands;

 template <class... OPS> explicit Expression(const OPS &... ops)
 : operands(ops...) {}

 double evaluate(unsigned i) const
 { return OPERATOR::evaluate(operands,i); }
};
```

[28] `http://en.cppreference.com/w/cpp/language/parameter_pack`

Le code présenté ici est simplifié pour montrer uniquement les principes essentiels (notamment, la transmission des types devrait être optimisée par le *perfect forwarding* [29], et les qualificatifs des types liés aux paramètres du patron `Expression` devraient être retirés). Supposons que chaque opérateur arithmétique unaire ou binaire du C++ soit représenté par une classe définie selon le schéma suivant.

```
struct AdditionOperator {
 template <class OP1,class OP2>
 static double evaluate(const std::tuple<OP1,OP2> & tuple,unsigned i) {
  return std::get<0>(tuple).evaluate(i) + std::get<1>(tuple).evaluate(i);
 }
};
```

Considérons une classe générique `Array<N>` (similaire à `std::array`) qui représente un tableau de taille `N`, pour laquelle nous voulons surcharger les opérateurs arithmétiques, de manière à ce que les opérations exprimées sur des tableaux soient appliquées sur chaque élément, e.g. l'opération `c = a+b` signifie `c[i] = a[i]+b[i]` pour chaque élément d'indice `i`. L'expression `-a+b*c` par exemple, où les variables `a`, `b` et `c` sont des objets de la classe `Array<N>`, peut être modélisée par une composition récursive utilisant le patron `Expression`.

```
using exp_t = Expression< AdditionOperator,
                          Expression< MinusOperator,
                                      Array<N>
                                    >,
                          Expression< MultiplicationOperator,
                                      Array<N>,
                                      Array<N>
                                    >
                        >;
```

A la compilation, cette structure statique pourrait être analysée par des techniques de métaprogrammation générique pour générer un code spécifique. Supposons maintenant que les opérateurs arithmétiques ont été surchargés de la manière suivante [30].

```
template <class OP1,class OP2>
inline Expression<AdditionOperator,OP1,OP2> operator+(OP1 && op1,
                                                      OP2 && op2)
{ return Expression<AdditionOperator,OP1,OP2>(op1,op2); }
```

Cette surcharge permet de produire automatiquement, à partir du code `-a+b*c`, un objet du type `exp_t`. Avant l'arrivée des expressions lambda en C++11, de tels objets pouvaient servir à représenter des fonctions lambda [Veld96b], mais l'intérêt majeur des patrons d'expressions réside dans leur capacité à différer l'évaluation des opérations intermédiaires pour éviter des objets temporaires inutiles. Par exemple, considérons l'affectation `d = -a+b*c`, où `d` est un objet de la classe `Array<N>`, et supposons que le patron `Array` possède l'opérateur d'affectation suivant.

```
template <class EXPRESSION>
inline Array<N> & Array<N>::operator=(const EXPRESSION & expression) {
 for (unsigned i = 0; i<N; ++i) values[i] = expression.evaluate(i);
 return *this;
}
```

[29] `http://en.cppreference.com/w/cpp/utility/forward`

[30] Les arguments sont des références "universelles" ici (syntaxe `&&`), i.e. on ne se préoccupe pas de leur aspect constant.

L'arbre syntaxique abstrait de l'expression est construit et transmis comme argument à l'opérateur d'affectation. La méthode `evaluate` est alors appelée récursivement sur chaque opération et opérande de l'expression, ce qui aboutit à une simple boucle et à l'*inlining* (cf. section 5.1.1) de l'évaluation complète. Le code généré pour l'opérateur d'affectation a donc une performance équivalente à:

```
for (unsigned i = 0; i<N; ++i) d[i] = -a[i] + b[i]*c[i];
```

La surcharge classique des opérateurs aurait créé des objets temporaires lors de l'évaluation de l'expression `-a+b*c`, produisant un code d'une performance équivalente à:

```
Array<N> a1; for (unsigned i = 0; i<N; ++i) a1[i] = -a[i];
Array<N> a2; for (unsigned i = 0; i<N; ++i) a2[i] = b[i]*c[i];
Array<N> a3; for (unsigned i = 0; i<N; ++i) a3[i] = a1[i]+a2[i];
```

6.3.2. Modélisation par les concepts

Nous proposons d'intégrer les concepts dans le développement d'un *framework* pour les patrons d'expressions [31], avec l'objectif de modéliser les expressions et de surcharger les opérateurs une fois pour toutes, de manière à ce que l'utilisateur puisse se concentrer sur l'évaluation des expressions. Dans l'exemple précédent, le patron `Expression` est conçu pour un seul type d'évaluation (cf. méthode `evaluate`), alors qu'il est envisageable d'avoir besoin de plusieurs types d'évaluations (e.g. calcul, affichage, analyse sémantique...). Notre solution utilise la spécialisation orientée concept comme moyen fiable et extensible pour définir des évaluations. Elle est inspirée du double *dispatch* du patron de conception *visiteur* [Gamm95] où la spécialisation de patron remplace la redéfinition de méthode. Les concepts peuvent aussi apporter plus de contrôle sur les opérandes, e.g. des assertions statiques formulées à l'aide de concepts peuvent détecter un mauvais emploi d'opérandes pour une opération.

6.3.2.1. Représentation des expressions

Une autre version du patron `Expression` est proposée [32], sans aucune méthode d'évaluation (le processus est externalisé grâce au double *dispatch* présenté à la section 6.3.3). Dans un souci de clarté, nous considérons ici uniquement des expressions "provisoires" (cf. *transient* dans notre code), i.e. des objets temporaires (ou *rvalues* [33]) qui doivent être utilisés immédiatement. Reporter leur utilisation implique une recopie de l'expression (où chaque opération et opérande devient une *lvalue* [33]) [Bach17].

```
template <class OPERATOR,class... OPERANDS> struct Expression {
 using operator_t = OPERATOR;
 using operands_t = std::tuple<OPERANDS...>;

 std::tuple<etc_transient_operand_t<OPERANDS>...> operands;

 template <class... OPS> explicit Expression(OPS &&... ops)
 : operands(std::forward<OPS>(ops)...) {}
};
```

[31] `http://forge.clermont-universite.fr/projects/et-concepts`
[32] A noter que cette fois-ci, le *perfect forwarding* est appliqué à l'aide de la fonction `std::forward`.
[33] `http://en.cppreference.com/w/cpp/language/value_category`

Le patron `Expression` modélise le concept `cExpression` qui impose une interface pour accéder à l'opérateur et aux opérandes de l'expression (cf. figure 6.6). Notons l'utilisation de la métafonction `etc_transient_operand_t` qui permet de choisir la manière de stocker une opérande (par référence, choix par défaut pour éviter toute copie superflue, ou par copie).

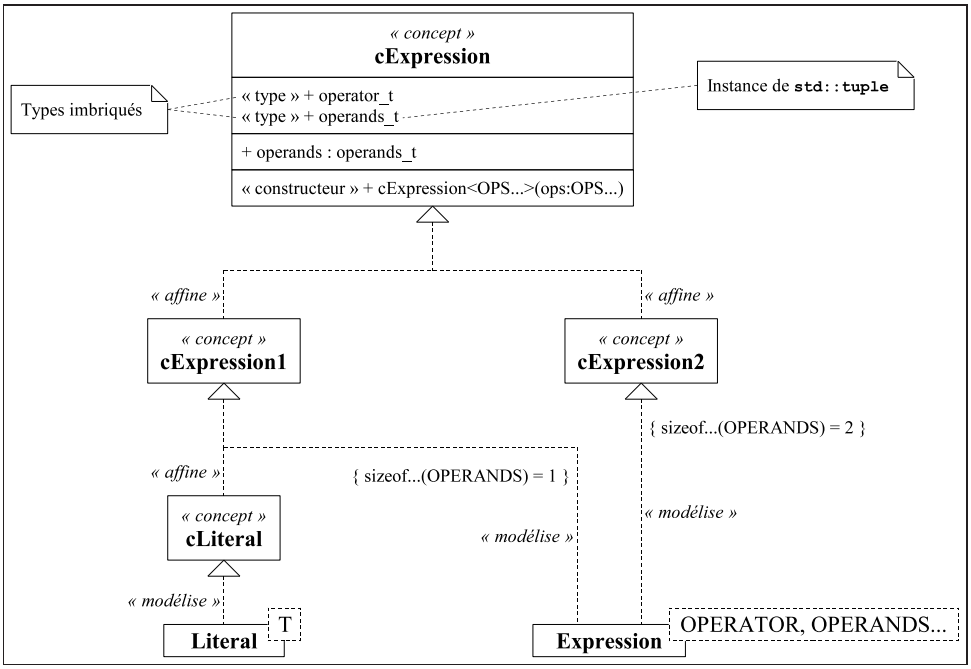

Figure 6.6: Taxonomie de concepts pour les expressions.

Un patron `Literal<T>` est nécessaire pour garder la trace de toute opérande *rvalue* de type `T` qui n'est pas un objet issu du patron `Expression`. Par exemple, dans l'expression `3*a`, la première opérande est une *rvalue* qui devra être copiée si l'on souhaite conserver l'expression [Bach17]. Toute instance du patron `Literal` est considérée comme une expression: le patron modélise le concept `cLiteral` qui affine le concept `cExpression` (cf. figure 6.6). Il fournit donc les mêmes fonctionnalités que le patron `Expression` en encapsulant simplement une référence constante sur l'opérande. La surcharge d'opérateur (détaillée par la suite) est conçue pour encapsuler automatiquement une opérande *rvalue* dans une instance du patron `Literal`. L'expression `3*a` produit alors un objet du type suivant.

```
Expression< MultiplicationOperator,Literal<int>,Array<N> >
```

6.3.2.2. Taxonomie de concepts

Comme l'analyse d'une expression repose sur les concepts, il est nécessaire de définir une taxonomie des concepts qui caractérisent les opérations et les opérandes d'une expression. La figure 6.6 montre que les patrons `Expression` et `Literal` modélisent indirectement le concept `cExpression` et que les expressions sont classées en fonction de leur arité: le concept `cExpression1` pour les expressions unaires, le concept `cExpression2` pour les expressions binaires...

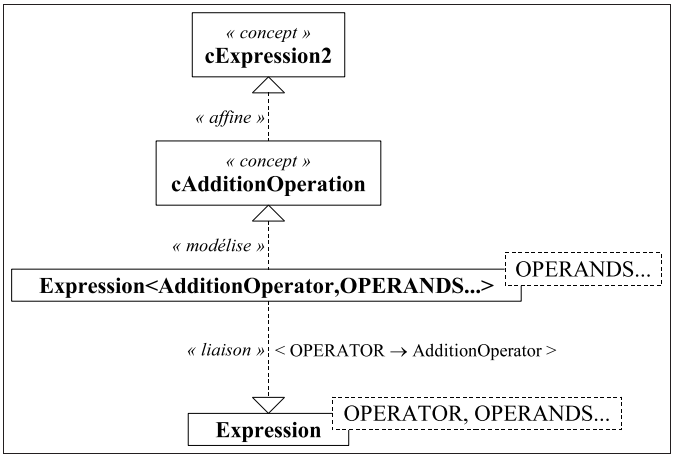

Figure 6.7: Opération d'addition dans la taxonomie de concepts.

Chaque opérateur d'intérêt pour les patrons d'expressions est représenté par une classe (comme `AdditionOperator` présentée à la section 6.3.1, mais sans méthode d'évaluation), et l'opération associée est caractérisée par un concept. La taxonomie de concepts est complétée de manière à ce que les expressions reposant sur l'opérateur modélisent le concept de l'opération. Par exemple, déclarer l'opérateur d'addition implique de définir la classe `AdditionOperator` et de déclarer le concept `cAdditionOperation` qui affine `cExpression2` (cf. figure 6.7). En outre, toute instance du patron `Expression` où `OPERATOR = AdditionOperator` modélise le concept `cAdditionOperation`. Cette déclaration peut être réalisée de la manière suivante avec C4TS++.

```
struct AdditionOperator;

gnx_declare_concept(cAdditionOperation);

template <> gnx_add_models(cAdditionOperation,cExpression2);

template <class... OPERANDS>
gnx_add_models(Expression<AdditionOperator,OPERANDS...>,
               cAdditionOperation);
```

La taxonomie peut être étendue, notamment pour détecter la sémantique originale d'un opérateur, e.g. savoir s'il est arithmétique, logique ou relationnel. Dans ce but, les concepts `cArithmeticOperation`, `cLogicalOperation` et `cRelationalOperation` ont été insérés dans la taxonomie (e.g. le concept `cAdditionOperation` affine le concept `cArithmeticOperation`).

6.3.2.3. Surcharge des opérateurs

Une fois un opérateur déclaré, la fonction associée doit être surchargée pour retourner un objet expression. Par exemple, la fonction `operator+` doit être surchargée pour retourner un objet du type `Expression<AdditionOperator,B1,B2>`, où `B1` et `B2` sont les types des deux opérandes. Cependant, une telle surcharge nécessite quelques précautions, comme le montre le code suivant (sur ce modèle, des macros sont proposées pour permettre une surcharge en une simple instruction).

```
template < class OP1,class OP2,
           class T1 = gnx_base_type_t<OP1>,
           class T2 = gnx_base_type_t<OP2>,
           class B1 = etc_operand_boxing_t<OP1 &&>,
           class B2 = etc_operand_boxing_t<OP2 &&>
         >
typename enable_if< gnx_or< etc_is_operand<T1>,
                            etc_is_operand<T2>
                          >,
                    Expression<AdditionOperator,B1,B2>
                  >::type
operator+(OP1 && op1,OP2 && op2) {
 return Expression<AdditionOperator,B1,B2>(std::forward<OP1>(op1),
                                           std::forward<OP2>(op2));
}
```

La surcharge est paramétrée sur les types `OP1` et `OP2` (déduits à l'appel de la fonction) qui sont convertis en `B1` et `B2` (opération appelée *boxing*) à l'aide de la métafonction `etc_operand_boxing_t`. Cette dernière est conçue pour convertir toute *rvalue* qui n'est pas issue du patron `Expression` en une instance du patron `Literal`. Les qualificatifs des types `OP1` et `OP2` sont retirés pour obtenir `T1` et `T2` (e.g. `const int &` devient `int`) à l'aide de la métafonction `gnx_base_type_t` (à noter qu'elle est aussi utilisée dans le *boxing* pour obtenir les types `B1` et `B2` sans qualificatifs).

Sans contraintes sur `OP1` et `OP2`, la surcharge serait valide pour n'importe quel type d'opérande. Le principe SFINAE (*Substitution Failure Is Not An Error*) du C++, à travers le patron `enable_if`[34], est appliqué pour un contrôle: la surcharge est valide seulement si `T1` ou `T2` est un type activé pour être une opérande (la métafonction `etc_is_operand<T>` indique si le type `T` est activé). Initialement, seuls les types reposant sur les patrons `Expression` et `Literal` sont activés pour être des opérandes, et la métafonction `etc_is_operand` (conçue sur le même principe que `gnx_models_concept`, cf. section 6.2.3) doit être spécialisée pour formellement activer un type. La macro `etc_activate_operand` est fournie pour faciliter la spécialisation, e.g. `Array<N>` peut être activée comme suit.

```
template <unsigned N> etc_activate_operand(Array<N>);
```

6.3.3. Evaluation statique d'une expression

6.3.3.1. Patron de conception *visiteur*

Le patron de conception *visiteur* est une solution courante pour représenter une opération à appliquer à chaque élément d'un ensemble hétérogène, où le code de l'opération dépend du type de l'élément [Gamm95]. Ce patron permet de définir de nouvelles opérations sans impact sur les classes des éléments. Dans notre cas, il peut être utilisé pour évaluer une expression (cf. figure 6.8), mais il présente quelques restrictions: il repose sur la liaison dynamique (*dynamic binding*, ce qui peut conduire à une augmentation significative du temps d'exécution), et les éléments doivent appartenir à la même classe de base (ce qui réduit les possibilités d'extension, car tout objet ne peut pas être un élément dans ce patron).

[34] `http://en.cppreference.com/w/cpp/types/enable_if`

Le principe est qu'un objet, le visiteur, est déplacé d'un élément à l'autre pour réaliser l'opération. Le code exécuté pour chaque visite dépend du type du visiteur (i.e. le genre d'opération à réaliser) et du type de l'élément. Ce *dispatch* repose sur deux méthodes virtuelles: `accept` qui est redéfinie pour chaque type d'élément (pour rediriger sur la bonne méthode `visit` du visiteur), et `visit` qui est surchargée pour chaque type d'élément et redéfinie pour chaque type d'opération (cf. figure 6.8, où les éléments visités sont les opérandes d'une expression). Une visite est réalisée en appelant la méthode `accept` de l'élément avec le visiteur comme argument.

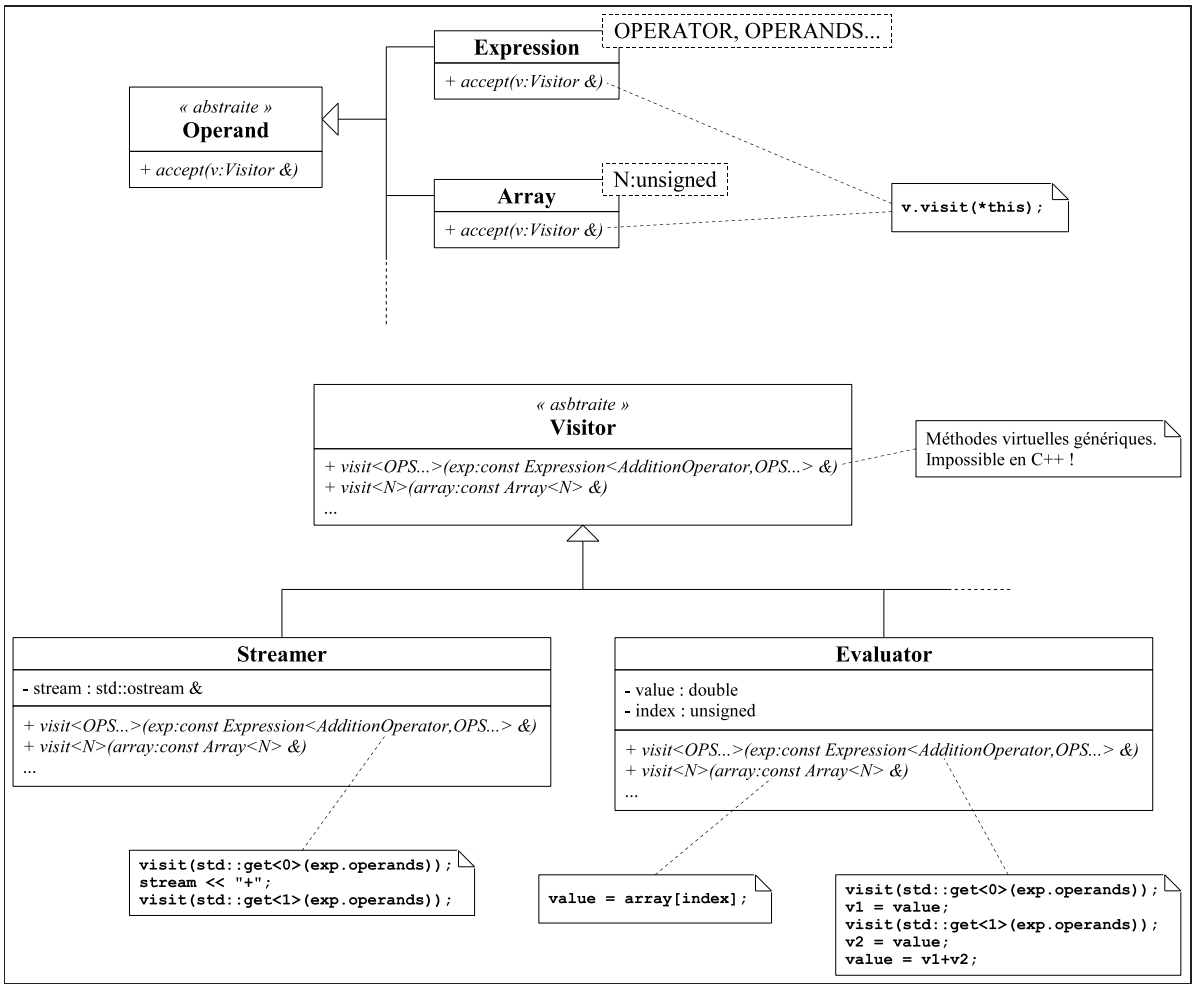

Figure 6.8: Patron de conception visiteur pour l'évaluation d'expression.

Dans cette conception, les éléments appartiennent à la même classe de base, alors qu'avec les patrons d'expressions, les éléments peuvent être de différents types sans relation de classe, comme les instances des patrons `Expression` et `Array`. La figure 6.8 présente aussi les méthodes `visit` comme des méthodes virtuelles génériques, ce qui est impossible en C++. Ainsi, la méthode `visit` devrait être explicitement surchargée pour potentiellement toute instance du patron `Expression`, ce qui est difficilement gérable.

En outre, la signature de la méthode `visit` est figée: dans notre exemple, il n'y a pas de valeur retournée et un seul argument (l'opérande). Une manière de contourner le problème consiste à ajouter des attributs au visiteur pour représenter la valeur de retour et les arguments, mais cela peut conduire à un code difficilement lisible (cf. visiteur `Evaluator`).

6.3.3.2. Visiteur orienté concept

Nous proposons une solution avec les concepts qui conserve l'idée du double *dispatch* du patron de conception *visiteur*. Cette approche statique est implémentée par un patron `etc_visit` avec deux paramètres: le type du visiteur et le type de l'opérande.

```
template < class VISITOR,
           class OPERAND,
           class CONCEPT = gnx_best_concept_t<VISITOR,OPERAND>,
           class ENABLE = void
         >
struct etc_visit;
```

Le double héritage du patron de conception original est remplacé par une simple spécialisation de patron: par exemple, au lieu de spécialiser la classe `Visitor` par la classe `X` et la classe `Operand` par la classe `Y`, le patron `etc_visit` est spécialisé avec `VISITOR = X` et `OPERAND = Y`. Le processus de spécialisation doit être contrôlé par les concepts, comme présenté dans la section 6.2, ce qui nécessite un paramètre additionnel `CONCEPT`. Un dernier paramètre `ENABLE` est également ajouté pour faciliter l'application du mécanisme SFINAE si nécessaire.

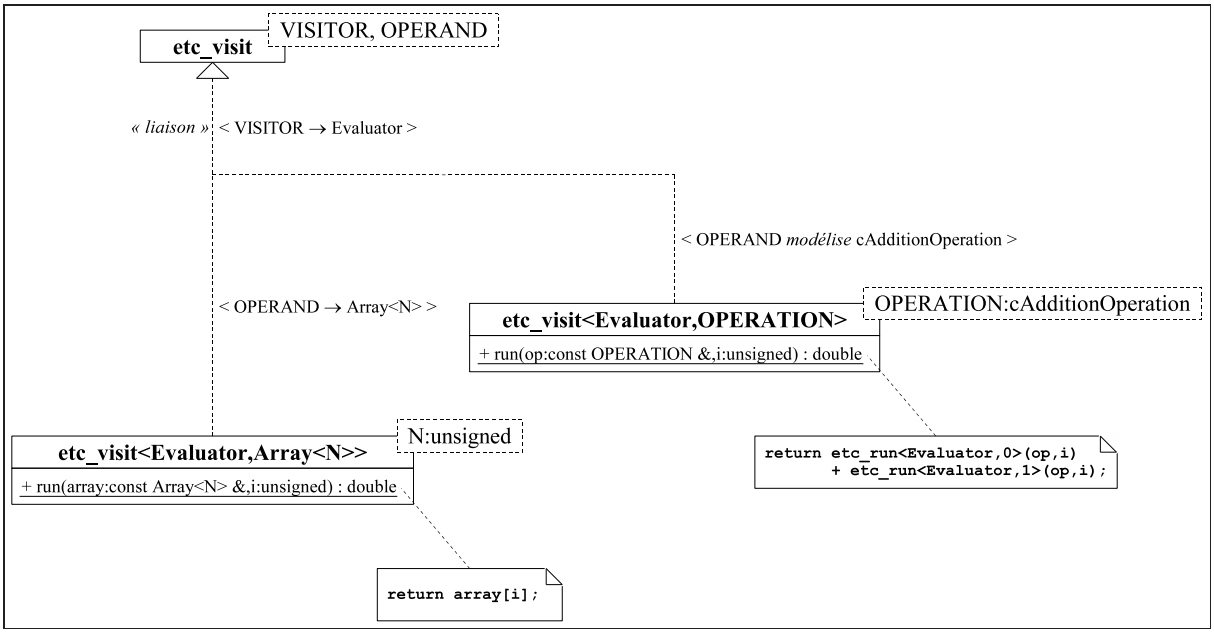

Figure 6.9: Visiteur orienté concept pour l'évaluation d'expression.

La notion de visiteur est toujours présente sous la forme d'une classe servant à identifier le type d'opération appliquée sur les éléments, mais l'instanciation d'un objet n'est pas obligatoire. Déclarons une classe vide `Evaluator` pour représenter l'opération consistant à calculer l'élément à l'indice `i` d'une expression d'objets `Array<N>`. Par exemple, pour définir la visite de l'opérateur d'addition, le patron `etc_visit` est spécialisé pour le concept `cAdditionOperation` et le visiteur `Evaluator` (cf. figure 6.9 et le code suivant). Notons que la classe du visiteur sert aussi de contexte de spécialisation (cf. section 6.2).

```
template <> gnx_add_uses(Evaluator,cAdditionOperation);

template <class OPERATION>
struct etc_visit<Evaluator,OPERATION,cAdditionOperation> {
 static double run(const OPERATION & op,unsigned i) {
  return etc_run<Evaluator,0>(op,i) + etc_run<Evaluator,1>(op,i);
 }
};
```

Le code de la visite est localisé dans une méthode statique `run` de la version spécialisée du patron `etc_visit`. Le prototype de cette méthode est quasi libre, la seule contrainte étant que le premier argument soit l'opérande à traiter. Parcourir toutes les opérandes et opérations d'une expression est récursif: par exemple, visiter une opération d'addition signifie ajouter les résultats des visites sur chaque opérande. Une visite est appelée par l'intermédiaire de la fonction `etc_run` qui facilite l'instanciation du patron `etc_visit`, comme le montre le code suivant.

```
template <class VISITOR,unsigned N,class EXPRESSION,class... ARGS>
inline auto etc_run(EXPRESSION && expression,ARGS &&... args) {
 return etc_visit< gnx_base_type_t<VISITOR>,
                   etc_operand_type_t<EXPRESSION,N>
                 >::run(etc_operand_value<N>(expression),
                        gnx_forward<ARGS>(args)...);
}
```

Les fonctions `etc_operand_type_t` et `etc_operand_value` sont fournies pour faciliter l'accès au type et à la valeur d'une opérande à partir de sa position dans une expression. Plusieurs versions de la fonction `etc_run` sont implémentées pour offrir plus de flexibilité. Notamment, la fonction peut être appelée avec l'indice de l'opérande à visiter, comme dans l'exemple précédent, ou sans indice pour visiter toute l'expression. Par exemple, l'opérateur d'affectation du patron `Array` peut être surchargé comme suit pour évaluer totalement l'expression reçue en argument.

```
template <class EXPRESSION>
inline Array<N> & Array<N>::operator=(const EXPRESSION & expression) {
 for (unsigned i = 0; i<N; ++i)
  values[i] = etc_run<Evaluator>(expression,i);

 return *this;
}
```

6.3.4. Exemple d'application

Un EDSL pour la programmation linéaire est défini ici à l'aide des patrons d'expressions orientés concept [35]. Un programme linéaire (classe `Program`) est un problème où l'on cherche à optimiser un objectif (minimiser ou maximiser une expression linéaire) sous des contraintes qui sont des expressions linéaires bornées par une valeur inférieure ou supérieure, ou égales à une valeur. Une expression linéaire (classe `Linear` [36]) est une somme pondérée de variables (classe `Variable`). Voici un exemple de programme linéaire exprimé avec notre EDSL.

$$P \begin{cases} \text{maximiser } 3x_1 - 2x_2 + 8x_3 \\ \text{avec } 5x_1 - 2x_2 + 4x_3 \leq 8 \\ \qquad x_1 + 3x_2 + 8x_3 \geq 25 \\ \qquad 9x_1 + 6x_2 - 3x_3 = 17 \end{cases}$$

```
Program p;

auto & x1 = p.newVariable();
auto & x2 = p.newVariable();
auto & x3 = p.newVariable();

p.maximize(3*x1 - 2*x2 + 8*x3);

p += 5*x1 - 2*x2 + 4*x3 <= 8;
p += x1 + 3*x2 + 8*x3 >= 25;
p += 9*x1 + 6*x2 - 3*x3 == 17;
```

La classe `Variable` est activée pour être une opérande, ce qui conduit automatiquement toute expression contenant au moins une opérande de ce type à produire un objet issu du patron `Expression`. Ce dernier peut être évalué par le visiteur `Builder` pour construire l'expression (ou la contrainte) linéaire qu'il représente (i.e. un objet `Linear`). Notamment, l'opérateur d'affectation de la classe `Linear` est surchargé de la manière suivante.

```
template <class EXPRESSION>
inline Linear & Linear::operator=(const EXPRESSION & expression) {
 clear();
 etc_run<Builder>(expression,*this);
 return *this;
}
```

La visite consiste à construire progressivement une expression (ou une contrainte) linéaire en associant un coefficient à chaque variable. Le processus commence avec une expression où tous les coefficients sont nuls, et progressivement, à chaque opération rencontrée, ces coefficients sont actualisés, ce qui conduit finalement à une expression linéaire simplifiée. Par exemple, les contraintes suivantes seront simplifiées lors de leur évaluation.

```
3*x1 + 4*(2*x2 - 3*x3) <= 13  ⟶  3*x1 + 8*x2 - 12*x3 <= 13
2*x1 - 3*x2 >= 5*x1 + 2*x3    ⟶  -3*x1 - 3*x2 - 2*x3 >= 0
```

La visite de `Builder` est définie à partir des spécialisations du patron `etc_visit` décrites dans le tableau 6.1. Certaines sont contraintes par le type de l'opérande (e.g. `Variable` et `Linear`) alors que d'autres sont contraintes par les concepts modélisés par l'opérande (e.g. les opérations d'addition et de multiplication).

[35] Code complet: `http://forge.clermont-universite.fr/projects/et-concepts`
[36] Pour éviter toute confusion avec le patron `Expression`, la classe `lin::Expression` du code est appelée ici `Linear`.

Contraintes de spécialisation	Assertions
`OPERAND` = `Variable`	
`OPERAND` = `Linear`	
`OPERAND` modélise `cLiteral`	Le littéral est arithmétique.
`OPERAND` modélise `cMultiplicationOperation`	L'une des opérandes est un littéral arithmétique et l'autre est une expression linéaire.
`OPERAND` modélise `cDivisionOperation`	L'opérande gauche est une expression linéaire et l'opérande droite est un littéral arithmétique.
`OPERAND` modélise `cPlusOperation` ou `cMinusOperation`	L'opérande est une expression linéaire.
`OPERAND` modélise `cAdditionOperation` ou `cSubtractionOperation`	Les deux opérandes sont des expressions linéaires.
`OPERAND` modélise `cRelationalOperation`	Les deux opérandes sont des expressions linéaires et la relation est une infériorité, une supériorité ou une égalité.

Tableau 6.1: Spécialisations pour la visite d'une expression ou d'une contrainte linéaire.

Pour chaque visite, des assertions sont établies afin de vérifier la syntaxe de l'expression et s'assurer qu'elle est bien linéaire. Ces assertions ne sont que superficielles car la récursivité de la visite permet une vérification en profondeur. Voici l'exemple de la visite d'une opération de soustraction.

```
template <> gnx_add_uses(Builder,cSubtractionOperation);

template <class TYPE>
struct etc_visit<Builder,TYPE,cSubtractionOperation> {
 static_assert(is_linear<etc_operand_type_t<TYPE,0>>::value,
               "Left operand must be a linear expression.");

 static_assert(is_linear<etc_operand_type_t<TYPE,1>>::value,
               "Right operand must be a linear expression.");

 static void run(const TYPE & operation,Linear & linear,
                 double coef = 1.0) {
  etc_run<Builder,0>(operation,linear,coef);
  etc_run<Builder,1>(operation,linear,-coef);
 }
};
```

Des assertions garantissent que les deux opérandes sont des expressions linéaires: la métafonction `is_linear` indique si une opérande est linéaire en s'assurant que son type est `Linear` ou `Variable`, ou bien qu'il modélise les concepts d'opérations *a priori* linéaires comme `cAdditionOperation`, `cMultiplicationOperation`... Ce test repose sur la métafonction `gnx_matches`[37] fournie par C4TS++.

```
template <class TYPE>
struct is_linear : gnx_matches<Builder,TYPE,
                               Linear,Variable,
                               cAdditionOperation,
                               cMultiplicationOperation,
                               [...]
                              > {};
```

[37] Le second paramètre est comparé aux types et concepts qui le suivent, le premier est l'observateur (cf. section 6.2.2).

A partir de seulement 8 spécialisations [38], la vérification, la construction et la simplification d'une expression ou d'une contrainte linéaire sont définies. Grâce aux concepts, la visite de l'expression est totalement contrôlée: la sélection d'une version spécialisée du patron `etc_visit` n'est possible que si l'opérande modélise le concept spécifié, et des assertions permettent des contrôles supplémentaires pour valider la syntaxe.

6.4. Conclusion

Ce chapitre montre l'intérêt des concepts pour la programmation générique, et en particulier pour le contrôle du processus de spécialisation des patrons. Les concepts semblent plus pertinents pour caractériser les paramètres d'un patron que leur motif de type, ce qui permet une meilleure discrimination des versions du composant générique. Les concepts n'étant pas encore intégrés au langage C++, nous avons proposé une bibliothèque portable reposant sur des techniques de métaprogrammation générique pour représenter partiellement les concepts et permettre la spécialisation de patron orientée concept.

Il est alors possible de définir une taxonomie de concepts à l'aide de relations de modélisation et d'affinement. Des versions spécialisées d'un patron peuvent ensuite être spécifiées par des contraintes liées aux concepts modélisés par ses paramètres. Lors d'une instanciation du patron, sa version la plus appropriée est automatiquement sélectionnée par un métaprogramme sur la base des concepts modélisés par les types liés aux paramètres. Une analyse théorique ainsi que des expériences pratiques ont montré que notre implémentation est adaptée à une utilisation à grande échelle des concepts dans le développement d'applications.

Afin d'illustrer les nouvelles possibilités offertes par la spécialisation de patron orientée concept, nous avons proposé une nouvelle conception des patrons d'expressions reposant sur les concepts. Une taxonomie extensible des concepts qui caractérisent les opérations et les opérandes d'une expression est d'abord définie. L'évaluation d'une expression est ensuite décrite à partir de spécialisations orientées concept d'un patron qui représente l'action à effectuer sur chaque opérande et opération de l'expression. Ces dernières sont discriminées par rapport aux concepts qu'elles modélisent, ce qui sert à définir les versions spécialisées du patron.

Les patrons d'expressions orientés concept ont permis de définir un EDSL pour la programmation linéaire. Avec seulement quelques spécialisations de patron, un processus d'évaluation pour vérifier, construire et simplifier des expressions et des contraintes linéaires a été élaboré. Cette expérience montre que les concepts offrent des possibilités pour contrôler la spécialisation de patron qui renforcent sa fiabilité et sa sélectivité. La spécialisation de patron orientée concept doit être envisagée en remplacement de l'héritage lorsque la liaison dynamique n'est pas nécessaire, comme c'est le cas pour le parcours de la structure statique d'une expression.

[38] C4TS++ permet l'utilisation de combinaisons logiques de concepts pour contrôler la spécialisation de patron [Bach12].

CHAPITRE 7
PROJET DE RECHERCHE

Les concepts offrent de nouvelles possibilités de conception logicielle en permettant notamment de développer des métaprogrammes avec plus de simplicité et de fiabilité. D'une manière générale, il est tout à fait envisageable d'exploiter des informations sur la structure d'un programme au moment de la compilation afin de l'optimiser pour ses exécutions futures. La métaprogrammation orientée concept peut alors intervenir pour capter les informations pertinentes du programme, souvent à l'aide des patrons d'expressions, et ensuite les analyser pour produire un code optimisé.

Dans ce contexte, nous proposons d'étudier l'apport des concepts à deux problématiques en particulier où un processus d'optimisation est souhaitable en amont de l'exécution du programme. La première étude concerne le couplage de modèles de simulation par intégration numérique qui nécessite d'analyser les dépendances entre les équations des modèles et d'ordonner les calculs afin d'en garantir l'exactitude. La seconde étude concerne le développement de métaheuristiques dont les structures algorithmiques se prêtent généralement à la parallélisation, l'objectif étant d'analyser ces structures à la compilation afin de produire automatiquement un plan de parallélisation qui soit efficace à l'exécution.

7.1. Patrons d'expressions avec concepts

Nous avons montré que les concepts peuvent améliorer la qualité d'un code exploitant les patrons d'expressions, notamment en simplifiant et en sécurisant l'évaluation d'une expression avec une variante statique du patron de conception *visiteur*. Un de nos objectifs est de proposer un EDSL pour la définition d'équations différentielles de modèles de simulation et éventuellement pour la description de la structure algorithmique d'une métaheuristique. Néanmoins, il est nécessaire d'examiner plus en profondeur certaines limitations de performance identifiées notamment dans [Bass98] et [Iglb12], pour lesquelles des variantes des patrons d'expressions ont déjà été proposées, et d'étudier l'apport d'une approche orientée concept à ces problématiques.

7.1.1. Problème d'*aliasing*

Tout d'abord, les patrons d'expressions peuvent masquer des informations cruciales pour le compilateur: par exemple, l'expression `a = b + c*b` où la variable `b` apparaît deux fois est vue comme l'expression `a = b1 + c*b2` où les deux occurrences de `b` sont considérées comme des variables différentes, ce qui empêche le compilateur de réaliser certaines optimisations. Les *fast expression templates* sont une solution proposée pour éviter ce phénomène d'*aliasing* [Hard05]. L'idée est d'obtenir un arbre syntaxique abstrait de l'expression (sous forme de type, cf. section 6.3) où chaque variable est identifiée par un type

différent. Par exemple, une classe générique `Variable` paramétrée sur un nombre statique incrémenté manuellement permet de représenter et de distinguer les variables d'une expression.

```
Variable<1> a;
Variable<2> b;
Variable<3> c;

c = b + c*b;
```

En associant statiquement à chaque variable un numéro unique que l'on retrouve dans l'arbre syntaxique abstrait de l'expression, on peut permettre au compilateur de distinguer plusieurs occurrences d'une même variable. [Hard05] propose d'accéder à une variable via un membre statique dans l'instance associée de la classe `Variable`. Nous pensons que cette approche n'est pas *thread-safe* (car chaque variable est considérée unique dans le programme, alors qu'elle devrait se situer sur la pile), et que la numérotation manuelle n'est pas acceptable; l'approche doit être revue, notamment à l'aide d'un compteur statique comme celui de notre bibliothèque C4TS++ (cf. section 6.2.2) pour automatiser la numérotation, et d'un visiteur orienté concept (cf. section 6.3.3) pour définir une transformation de l'expression permettant d'éviter le phénomène d'*aliasing*.

7.1.2. Optimisation de boucle

Généralement, les boucles peuvent être optimisées, soit par le compilateur, soit par le développeur, via des techniques de *loop unrolling* (généralement effectuées par le compilateur) qui visent à accélérer l'exécution du code, et de *loop blocking* qui visent à optimiser l'accès mémoire grâce à une meilleure utilisation du cache. Les patrons d'expressions doivent permettre ces optimisations. Prenons l'exemple du calcul `a=b*c`, où `b` est une matrice de taille $N \times N$, et `a` et `c` sont des vecteurs de taille N. Une implémentation classique des patrons d'expressions conduit à un code équivalent au suivant.

```
for (unsigned i = 0; i<N; ++i) {
 a[i] = 0;

 for (unsigned j = 0; j<N; ++j)
  a[i] += b[i][j]*c[j];
}
```

Ce code va accéder plusieurs fois aux mêmes éléments des vecteurs `a` et `c`, mais à des itérations différentes des boucles, ce qui va certainement provoquer plusieurs chargements des mêmes éléments de la mémoire vers le cache. Pour limiter cet effet, la technique de *loop blocking* consiste à réorganiser les boucles afin d'exploiter autant que possible les données en cache avant de les remplacer. Au lieu de traiter les éléments des vecteurs et de la matrice l'un après l'autre dans l'ordre des indices, ils vont être regroupés et traités par bloc. En supposant K la taille d'un bloc, une version améliorée de la boucle précédente est la suivante.

```
for (unsigned i = 0; i<N; i+=K) {
 for (unsigned x = i; x < min(i+K,N); ++x) a[x] = 0;

 for (unsigned j = 0; j<N; j+=K)
  for (unsigned x = i; x < min(i+K,N); ++x)
   for (unsigned y = j; y < min(j+K,N); ++y)
    a[x] += b[x][y]*c[y];
}
```

Connaissant le nombre d'opérandes dans l'expression, il est possible de déterminer la valeur K qui permet de remplir au maximum le cache, notamment par métaprogrammation, en analysant l'arbre syntaxique abstrait de l'expression (e.g. [Hard06b]). Une évaluation à l'aide d'un visiteur orienté concept doit permettre d'effectuer cette tâche simplement. D'autres techniques d'optimisation peuvent aussi être envisagées, comme la parallélisation des boucles [Hard10].

7.1.3. Optimisation d'expression

La technique des patrons d'expressions de supprimer les objets temporaires dans l'évaluation d'une expression n'est pas toujours efficace. Ces objets temporaires, qui représentent des calculs intermédiaires, peuvent dans certains cas permettre d'accélérer significativement l'évaluation globale de l'expression. Considérons une somme de matrices `A=B+C+D`, l'évaluation classique par les patrons d'expressions est efficace puisque les éléments sont calculés directement: `A[i][j]=B[i][j]+C[i][j]+D[i][j]`. Prenons maintenant l'exemple du produit matriciel `A=B*C*D`, le calcul direct de chaque élément n'est pas approprié, il est effectivement plus judicieux de stocker le résultat `B*C` dans un objet temporaire avant de le multiplier à la matrice `D`.

Dans le cas du calcul matriciel, il faut donc traiter différemment les évaluations d'une addition et d'une multiplication. Il peut aussi être nécessaire de restructurer une expression, le calcul `A*B*v` par exemple, où `A` et `B` sont des matrices et `v` un vecteur, est évalué de gauche à droite, i.e. équivalent à `(A*B)*v`, alors que le calcul `A*(B*v)` serait plus efficace puisqu'il s'agit alors de deux produits matrice-vecteur au lieu d'un produit matrice-matrice suivi d'un produit matrice-vecteur.

[Iglb12] propose une variante des patrons d'expressions appelée *smart expression templates*, qui repose sur le schéma *Barton-Nackman Trick* [Bart96b] pour permettre une surcharge dédiée des opérateurs en fonction de leur nature (notamment pour permettre la création ou non d'un objet temporaire). Nous pensons que cette approche n'est pas suffisamment flexible et impose un volume de code important, alors que les concepts peuvent apporter une sémantique aux opérateurs (comme [Gott08] qui attribue des propriétés algébriques aux types via les concepts), et à l'aide d'un visiteur orienté concept, peuvent permettre d'adapter l'évaluation en fonction de la nature des opérateurs, voire de transformer l'expression si nécessaire.

7.2. Couplage de simulations par intégration numérique

Les modèles de simulation par intégration numérique tels qu'ils sont décrits à la section 4.1 (i.e. composés d'équations différentielles ordinaires du premier ordre) soulèvent des difficultés de couplage. L'apparition de dépendances entre les équations de plusieurs modèles peut nécessiter une réorganisation des calculs des dérivées des variables d'intégration par la méthode d'approximation du simulateur. Comme illustration, nous étudions le modèle classique proie-prédateur auquel nous ajoutons un phénomène de décomposition des carcasses des animaux morts qui alimente le stock de matière organique du sol et augmente ainsi la production des plantes consommées par les herbivores.

7.2.1. Dépendances entre équations

Nous considérons un modèle *Prédateur* qui représente une population de prédateurs se nourrissant uniquement d'herbivores, dont la population est représentée par le modèle *Herbivore* (cf. figure 7.1). Ces deux modèles décrivent l'évolution de trois variables d'intégration (la population d'animaux P_p ou P_h, le nombre cumulé de naissances B_p ou B_h, et le nombre cumulé de morts naturelles D_p ou D_h) qui dépendent, pour les prédateurs d'une population P_h d'herbivores (leurs proies), et pour les herbivores d'une population de prédateurs P_p. Dans le modèle *Herbivore*, une variable H_h supplémentaire décrit l'évolution du nombre de morts par prédation.

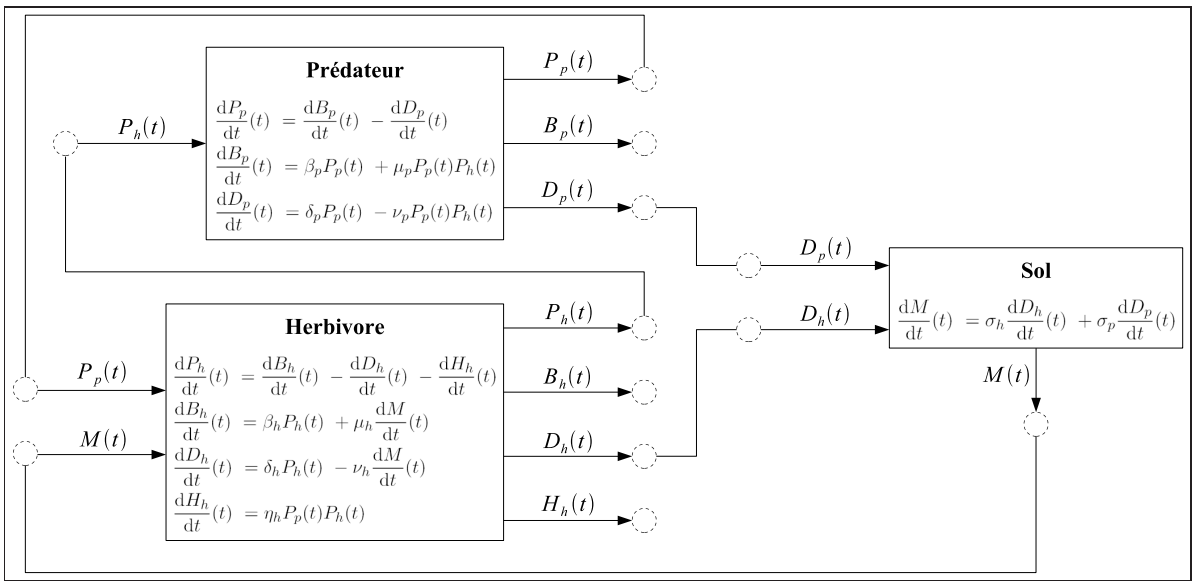

Figure 7.1: Modèle proie-prédateur.

Le schéma classique proie-prédateur est complété ici pour tenir compte du stock M de matière organique dans le sol qui a un impact sur la quantité de nourriture disponible pour les herbivores, et donc sur leurs nombres de morts naturelles et de naissances (on suppose que la variation du stock influence directement

ces quantités, d'où l'expression de la dérivée de M dans les équations décrivant B_h et D_h). Le modèle *Sol* est également ajouté pour décrire l'évolution du stock de matière organique qui dépend du nombre de morts naturelles d'animaux (proies et prédateurs).

La simulation de tels modèles consiste à calculer, à chaque pas de temps, les dérivées des variables d'intégration pour ensuite actualiser les valeurs des variables d'intégration. Avec ces deux étapes distinctes, que le calcul d'une dérivée soit dépendant d'une variable d'intégration n'est aucunement problématique. En revanche, une dépendance entre deux dérivées implique un ordre de leur calcul précis. La figure 7.2 exprime les dépendances entre dérivées sous forme de graphe pour les trois modèles du schéma proie-prédateur, d'abord séparément, puis couplés.

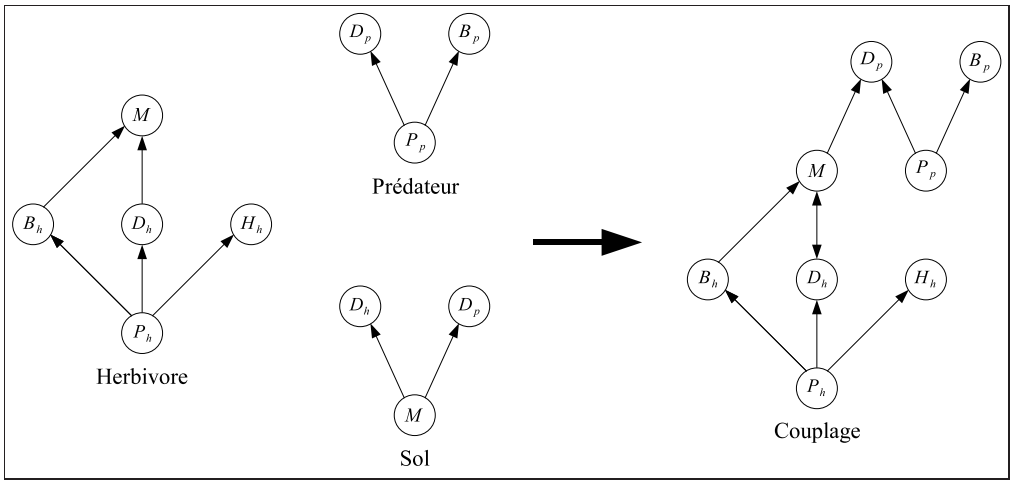

Figure 7.2: Dépendance des dérivées des variables d'intégration du modèle proie-prédateur.

Un tri topologique du graphe de dépendance [Kahn62], s'il est sans circuit, permet de déterminer l'ordre de calcul des dérivées. Dans notre cas, l'interdépendance entre les dérivées de M et D_h empêche de trouver un ordre topologique. En analysant leurs équations, la substitution de D_h dans l'équation de M permet d'éviter la dépendance:

$$\frac{\mathrm{d}M}{\mathrm{d}t}(t) = \frac{\sigma_h \delta_h P_h(t) + \sigma_p \frac{\mathrm{d}D_p}{\mathrm{d}t}(t)}{1 + \sigma_h \nu_h}$$

7.2.2. Modélisation par patrons d'expressions

Dans le cas où le couplage est statique, on peut envisager de traiter les problèmes de dépendance par métaprogrammation. Un EDSL pourrait être développé pour capter les équations des modèles, et pour un couplage, analyser ensuite l'ensemble d'équations et construire un graphe de dépendance des dérivées des variables d'intégration. Néanmoins, les patrons d'expressions tels que nous les avons présentés se limitent à capter et manipuler de manière statique une seule expression à la fois.

En effet, la technique consiste à construire un objet temporaire représentant la structure d'une expression qui est immédiatement exploité au moment de l'affectation. Dans notre cas, il est nécessaire de pouvoir manipuler plusieurs expressions (les équations des modèles) et de les regrouper à volonté (pour réaliser un couplage de modèles). Dans [Bach17], nous proposons la possibilité de copier l'expression temporaire dans un objet persistant qui puisse être utilisé à tout moment dans le programme. Cependant, avec cette approche, la construction d'un graphe de dépendance serait dynamique et empêcherait des transformations par métaprogrammation.

Il est possible de capter la structure d'un ensemble d'expressions en surchargeant l'opérateur d'affectation pour qu'il renvoie un objet `Expression` (cf. section 6.3) plutôt que le résultat de l'évaluation de l'expression, et l'opérateur virgule, par exemple, qui servira de séparateur entre les expressions. Ainsi, considérons les équations du modèle *Prédateur* en supposant que les variables d'intégration sont des instances d'une classe `Variable` avec un attribut public `d` (classe `Derivative`) modélisant la dérivée.

```
Model predateur = Pp.d = Bp.d - Dp.d,
                  Bp.d = bp * Pp + up * Pp * Ph,
                  Dp.d = dp * Pp - vp * Pp * Ph;
```

L'opérateur d'affectation de l'objet `predateur` reçoit alors un objet dont le type correspond à:

```
using predateur_t = Expression< CommaOperator,
                                Pp_t,
                                Expression< CommaOperator,
                                            Bp_t,
                                            Dp_t
                                          >
                              >;
```

où `Pp_t`, `Bp_t` et `Dp_t` symbolisent les types de chacune des équations séparées par une virgule. Par exemple, le type `Pp_t` correspond à:

```
using Pp_t = Expression< AssignmentOperator,
                         Derivative,
                         Expression< SubtractionOperator,
                                     Derivative,
                                     Derivative
                                   >
                       >;
```

La structure de l'expression obtenue ne permet pas de distinguer les dérivées, ce qui empêche l'analyse des dépendances. Une solution consisterait alors à attribuer un type différent à chaque dérivée, à l'aide par exemple de la technique employée par les *fast expression templates* (cf. section précédente) pour éviter l'*aliasing* d'opérandes. Cependant, une solution plus simple peut être employée ici, permettant d'éviter l'utilisation d'un compteur statique. Il s'agit de paramétrer les classes `Variable` et `Derivative` sur un type qui identifie de manière unique une variable, plutôt que sur un numéro.

```
template <class ID> class Derivative { [...] };

template <class ID> class Variable {
 [...]
 public: Derivative<ID> d;
 [...]
};
```

Une astuce permet de déclarer les variables simplement à l'aide d'une déclaration anticipée à la volée [39], où la valeur du paramètre de la classe `Variable` à l'instanciation est un type avec le même identifiant que la variable représentée. Il reste néanmoins un problème de portée pour ces types déclarés à la volée qu'il faudra traiter (peut-être via l'utilisation d'espaces de noms).

```
#define variable(NAME) Variable<class NAME> NAME

[...]

variable(Pp);
variable(Bp);
variable(Dp);
```

De cette manière, les types `Pp_t`, `Bp_t` et `Dp_t` deviennent plus explicites.

```
using Pp_t = Expression< AssignmentOperator,
                         Derivative<Pp>,
                         Expression< SubtractionOperator,
                                     Derivative<Bp>,
                                     Derivative<Dp>
                                   >
                       >;
```

Il reste maintenant à pouvoir déclarer plusieurs ensembles d'équations séparément et les assembler à volonté. Pour cela, il est possible avec C++11 de capter le type du résultat d'une expression grâce à l'instruction `decltype`. Ainsi, avec la technique de surcharge des opérateurs utilisée par les patrons d'expressions, il est possible d'obtenir le type représentant l'arbre syntaxique abstrait d'une expression et d'assembler alors les arbres de plusieurs expressions. Les trois modèles de notre schéma proie-prédateur peuvent être décrits de la manière suivante.

```
using Herbivore = decltype (
 Ph.d = Bh.d - Dh.d - Hh.d,
 Bh.d = bh * Ph + uh * M.d,
 Dh.d = dh * Ph - vh * M.d,
 Hh.d = nh * Pp * Ph
);

using Predateur = decltype (
 Pp.d = Bp.d - Dp.d,
 Bp.d = bp * Pp + up * Pp * Ph,
 Dp.d = dp * Pp - vp * Pp * Ph
);

using Sol = decltype (
 M.d = sh * Dh.d + sp * Dp.d
);
```

Le type `Predateur` est ainsi équivalent au type `predateur_t` présenté précédemment. Il est ensuite possible d'assembler les modèles comme suit.

```
using ProiePredateur = decltype ( Herbivore(), Predateur(), Sol() );
```

[39] `http://boost-spirit.com/home/2013/02/23/spirit-x3-on-github/`

7.2.3. Ordonnancement des calculs

La structure de toutes les équations des modèles couplés est finalement décrite par le type `ProiePredateur` qu'il est possible de transformer par métaprogrammation générique pour construire un graphe de dépendance des variables d'intégration. A partir de ce graphe, des circuits peuvent être détectés pour identifier des interdépendances et la séquence de calcul peut être construite dans un ordre approprié. On peut aussi envisager des transformations des expressions pour réécrire des équations et résoudre ainsi certains problèmes d'interdépendance. Toutes ces analyses et transformations seraient réalisées par métaprogrammation générique, ce qui nécessite d'avoir une représentation statique d'un graphe (à l'image des *typelists* par exemple) et des algorithmes statiques sous forme de métafonctions pour cette structure.

Dans notre exemple, la description du couplage a été volontairement simplifiée en utilisant les mêmes identifiants de variables dans plusieurs modèles (e.g. variable P_p utilisée comme sortie de *Prédateur* et entrée de *Herbivore*), ce qui permet une association implicite entre variables lors du couplage. Une solution décrivant les entrées et les sorties de chaque modèle, ainsi que les connexions à réaliser à chaque couplage, devra être trouvée, en s'inspirant notamment de l'approche proposée dans [Tour10] pour le couplage de modèles de simulation DEVS (*Discrete EVent System specification*).

7.3. Parallélisation automatique de métaheuristiques

Les métaheuristiques nécessitent généralement l'évaluation d'un nombre important de solutions qui peuvent, à certaines étapes du processus, être traitées indépendamment. Les stratégies employées pour parcourir l'espace des solutions combinent souvent des déplacements de différentes portées. La structure algorithmique d'une métaheuristique peut donc offrir un potentiel de parallélisation sur plusieurs niveaux qui permet d'envisager des stratégies sophistiquées sur des architectures à mémoire partagée: exploiter au maximum les coeurs d'un ou plusieurs processeurs, déléguer les opérations parallélisables les plus fines à des accélérateurs (e.g. GPU, Xeon Phi)...

Algorithme 7.1: **GRASP(problème P,solution x).**

```
pour i = 1..n faire
 x_i ← nouvelleSolution(P);
 heuristiqueConstructive(x_i);
 rechercheLocale(x_i);
fin pour;

x ← meilleureSolution({x_1, ..., x_n});
```

Algorithme 7.2: **ELS(solution x).**

```
x' ← x;
pour i = 1..m faire
 pour j = 1..p faire
  x_j ← x';
  mutation(x_j);
  rechercheLocale(x_j);
 fin pour;

 x'' ← meilleureSolution({x_1, ..., x_p});
 x ← meilleureSolution({x'', x});
 x' ← acceptation(x'', x');
fin pour;
```

Comme illustration, nous étudions la structure de la métaheuristique GRASP $\times$ ELS qui est un couplage de la métaheuristique GRASP (*Greedy Randomized Adaptive Search Procedure*, cf. section 2.2.3) avec la stratégie de recherche locale ELS (*Evolutionary Local Search* [Wolf07]). L'algorithme 7.1 présente le schéma général de la métaheuristique GRASP: plusieurs solutions sont produites par une heuristique constructive non déterministe et sont améliorées ensuite par une recherche locale. Chaque solution est traitée indépendamment, ce qui rend la boucle de l'algorithme parallélisable. La meilleure solution trouvée est finalement retenue.

L'algorithme 7.2 présente la stratégie de recherche locale de la métaheuristique ELS qui, de manière répétée (cf. boucle principale), produit des solutions dérivées d'une solution de référence à l'aide d'une procédure non déterministe de mutation suivie d'une amélioration par recherche locale (cf. boucle interne). Les solutions dérivées sont traitées indépendamment, ce qui permet d'envisager la parallélisation de la boucle interne. A l'issue de celle-ci, la meilleure solution dérivée est retenue et une procédure d'acceptation détermine si cette solution devient la solution de référence pour la génération de solutions dérivées à l'itération suivante de la boucle principale. Cette dernière doit donc être exécutée séquentiellement, car chaque itération dépend de l'état de la précédente.

7.3.1. Squelettes algorithmiques

La structure d'un algorithme est statique, ce qui permet d'envisager son analyse par métaprogrammation générique pour produire automatiquement un code parallèle exploitant les ressources d'une architecture matérielle donnée. Pour cela, il est nécessaire d'obtenir une représentation de l'algorithme permettant d'identifier les potentielles parallélisations, notamment à partir des caractéristiques des boucles: indépendance des itérations, opération de réduction (e.g. retenir la meilleure solution)... Nous proposons de décrire une métaheuristique par l'assemblage de squelettes algorithmiques [Cole04], un squelette représentant de manière générique une structure algorithmique singulière dont l'implémentation dépend de l'architecture matérielle ciblée. Dans notre exemple, nous identifions trois squelettes élémentaires pouvant être combinés pour former les squelettes des algorithmes GRASP et ELS (cf. figure 7.3).

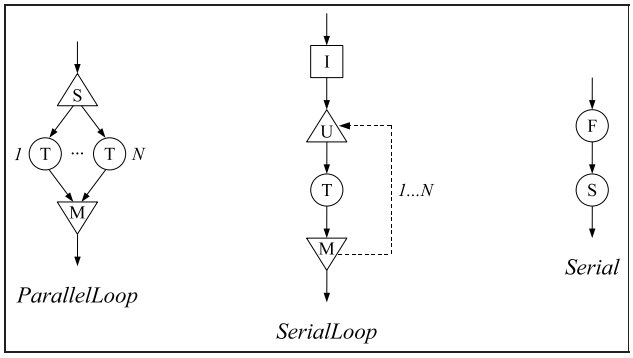

Figure 7.3: Proposition de squelettes algorithmiques.

Tout d'abord, le squelette *ParallelLoop* représente une boucle d'itérations indépendantes (e.g. la boucle principale de GRASP) dont l'algorithme (séquentiel) est présenté ci-dessous. L'opération *split* (*S*) distribue des données à chaque itération, *task* (*T*) réalise un traitement sur ces données, et *merge* (*M*) récupère le résultat de chaque itération pour produire le résultat final.

```
ParallelLoop(Input input,Output output).
data ← split(input,1);
output ← task(data);

pour k = 2..n faire
 data ← split(input,k);
 result ← task(data);
 output ← merge(result,output);
fin pour;
```

```
SerialLoop(Input input,Output output).
data ← init(input);
output ← task(data);

pour k = 2..n faire
 result ← task(data);
 output ← merge(result,output);
 data ← update(result,data);
fin pour;
```

Deux autres squelettes élémentaires sont nécessaires: *SerialLoop* qui représente une boucle où chaque itération dépend de l'état de l'itération précédente, et *Serial* qui représente la succession de deux traitements, le résultat du premier étant transmis au second.

```
Serial(Input input,Output output).
result ← first(input);
output ← second(result);
```

7.3.2. Modélisation par métaprogrammation générique

Les squelettes peuvent être modélisés à l'aide de classes génériques (e.g. [Falc06b]), afin d'obtenir une représentation de la structure d'un algorithme sous la forme d'une composition de types, à l'image des patrons d'expressions. Ainsi, il est possible par métaprogrammation de parcourir la structure, de l'analyser et de produire une implémentation adaptée à une architecture cible. Par exemple, le squelette *ParallelLoop* est modélisé par une classe générique `ParallelLoop` paramétrée sur les opérations *split*, *task* et *merge* (i.e. les types paramètres `SPLIT`, `TASK` et `MERGE`). Chaque opération peut être un squelette ou une portion de code, à condition de respecter certaines spécifications (e.g. le concept `cAlgorithm`).

```
template <class SPLIT,class TASK,class MERGE>
struct ParallelLoop : skl_arity<1> {
 using skl_input  = typename SPLIT::skl_input;
 using skl_output = typename MERGE::skl_output;

 static_assert(gnx_matches<ParallelLoop,SPLIT,cSplit>::value);
 static_assert(gnx_matches<ParallelLoop,TASK,cAlgorithm>::value);
 static_assert(gnx_matches<ParallelLoop,MERGE,cMerge>::value);

 static_assert(skl_connect<SPLIT,TASK,1>::value);
 static_assert(skl_connect<TASK,MERGE,1>::value);
 static_assert(skl_connect<MERGE,MERGE,2>::value);
};
```

Cette classe générique ne contient pas le code du squelette, puisqu'il dépend de l'architecture sur laquelle le squelette sera projeté. Néanmoins, la classe peut contenir des renseignements comme l'arité (e.g. `skl_arity`) ou la signature de l'algorithme (i.e. les types des arguments et du retour, e.g. `skl_input` et `skl_output`). On peut envisager aussi de définir des concepts pour spécifier les caractéristiques des trois opérations et s'assurer que les types fournis à l'instanciation modélisent bien ces concepts (cf. assertions avec `gnx_matches`). D'autres contraintes liées aux interactions entre les opérations peuvent être définies: par exemple, le type du retour de *split* doit correspondre au type du premier argument de *task* (cf. assertions avec `skl_connect`).

Pour des raisons de clarté, cette modélisation est simplifiée ici et devra être affinée par la suite, en particulier concernant la description des entrées-sorties des squelettes et des interactions entre les opérations internes (flot des données, dépendance...). Néanmoins, les squelettes des algorithmes GRASP et ELS (cf. code suivant) peuvent être exprimés par la combinaison de squelettes élémentaires et de portions de code séquentiel (e.g. `NouvelleSolution`, `MeilleureSolution`...), tous modélisés par des types, pour obtenir un squelette élaboré sous la forme d'un type paramétré.

```
template <class PROBLEME,class SOLUTION,
          class H_CONSTRUCTIVE,class RECHERCHE_LOCALE>
using GRASP = ParallelLoop< NouvelleSolution<PROBLEME,SOLUTION>,
                            Serial<H_CONSTRUCTIVE,RECHERCHE_LOCALE>,
                            MeilleureSolution<PROBLEME,SOLUTION>,
                          >;


template <class PROBLEME,class SOLUTION,
          class MUTATION,class RECHERCHE_LOCALE,class ACCEPTATION>
using ELS = SerialLoop< CopieSolution<PROBLEME,SOLUTION>,
                        ParallelLoop< CopieSolution<PROBLEME,SOLUTION>,
                                      Serial<MUTATION,RECHERCHE_LOCALE>,
                                      MeilleureSolution<PROBLEME,SOLUTION>
                                    >,
                        ACCEPTATION
                      >;
```

A l'instar des patrons d'expressions, ces squelettes peuvent être exploités par métaprogrammation, notamment à l'aide d'un visiteur orienté concept. Afin de faciliter leur description, un EDSL pourrait être développé (e.g. [Said09]), un squelette étant alors représenté sous la forme d'un arbre syntaxique abstrait. D'autres modélisations des squelettes ont été proposées [Gonz10], la plupart reposant sur des classes abstraites dont la spécialisation est nécessaire pour définir une implémentation sur une architecture donnée (e.g. [Gosw02]), ce qui induit un surcoût à l'exécution dont l'impact est plus ou moins important suivant la granularité des opérations parallélisées. L'approche par métaprogrammation vise à réduire ce surcoût en préparant au mieux le code du squelette pour l'architecture ciblée.

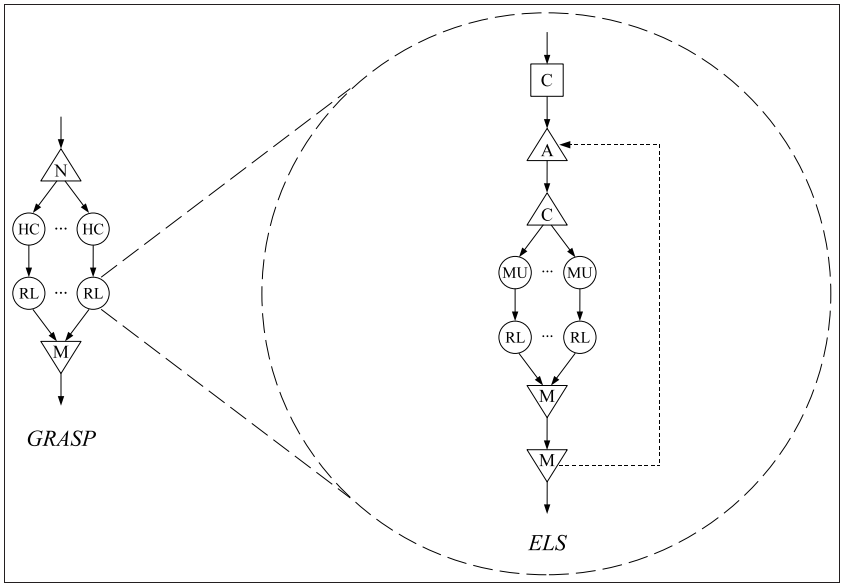

Figure 7.4: Squelette GRASP × ELS.

Les squelettes élaborés *GRASP* et *ELS* peuvent à leur tour être combinés pour former la structure de l'algorithme GRASP × ELS. Cette association révèle une structure algorithmique avec deux boucles imbriquées qui peuvent potentiellement être parallélisées (cf. figure 7.4). Néanmoins, il est nécessaire de s'interroger sur la pertinence de paralléliser l'une ou l'autre des boucles, ou bien les deux, en fonction de l'architecture ciblée (nombre de coeurs, hybride CPU-accélérateur...).

7.3.3. Graphes de tâches

A la compilation, un squelette peut être analysé pour produire un graphe, dit de tâches, qui identifie notamment les dépendances entre les opérations (ou tâches) de l'algorithme. Considérons le squelette *GRASP* × *ELS* où $n = 6$ itérations pour GRASP et $p = 2$ itérations pour ELS (sa boucle interne), sa structure peut être modélisée par le graphe de tâches de la figure 7.5. En supposant une architecture multiprocesseur (i.e. une architecture à mémoire partagée constituée de plusieurs processeurs avec éventuellement plusieurs coeurs), il faut décider de l'attribution des 12 tâches (supposées avoir des temps d'exécution similaires) aux coeurs.

Si l'on dispose de suffisamment de coeurs, la solution consiste à paralléliser les deux boucles, i.e. à définir 12 *threads* de calcul, chacun affecté à un coeur différent et chargé de traiter une tâche seulement. Si le nombre de coeurs est limité, il faut s'interroger sur la pertinence de paralléliser la boucle interne (i.e. celle de l'ELS). En effet, il semble *a priori* plus efficace de paralléliser la boucle externe (i.e. celle de GRASP) et de maintenir la boucle interne séquentielle: si l'on dispose de 3 coeurs, on définit 3 *threads* chargés chacun de traiter 2 itérations de la boucle externe.

En revanche, dans le cas où le nombre d'itérations de la boucle externe n'est pas un multiple du nombre de coeurs disponibles (e.g. 4 coeurs), il n'est pas possible de répartir la charge de travail de façon homogène sans paralléliser la boucle interne. Dans notre exemple, une solution consiste à définir 4 *threads* (cf. figure 7.5) chargés chacun de traiter d'abord une itération de la boucle externe (les itérations 1 à 4), puis de traiter une itération de la boucle interne (les itérations 5 et 6, divisées en tâches 9 à 12).

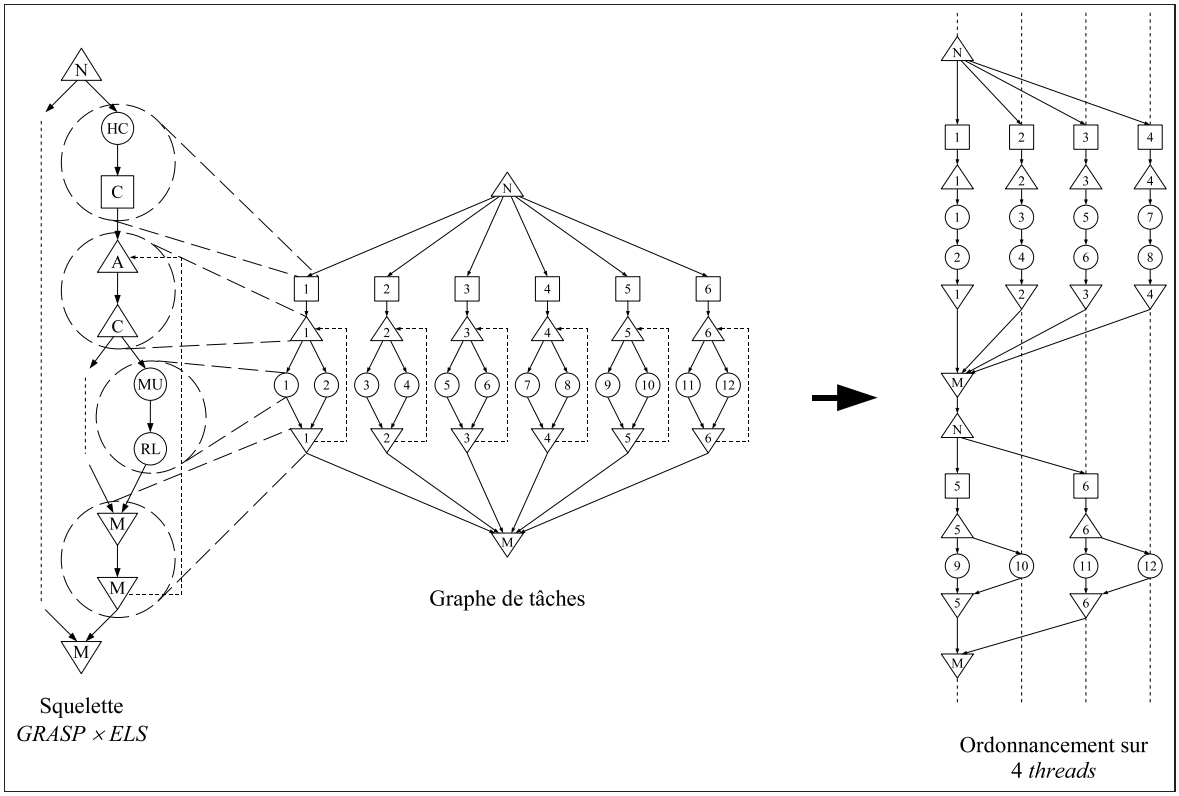

Figure 7.5: Graphe de tâches.

Le raisonnement mené ici suppose de connaître à la fois le nombre d'itérations des boucles et la quantité de coeurs disponibles. Ces informations ne sont généralement pas toutes statiques, ce qui empêche de préparer précisément le nombre de *threads* et leur travail à la compilation. Une approche purement dynamique consiste à créer, au début de chaque boucle parallélisée, la quantité de *threads* adéquate et de leur attribuer une charge de travail. La création d'un *thread* étant coûteuse, cette solution n'est pas adaptée à un parallélisme multiniveau.

Une autre approche consiste à créer autant de *threads* que de coeurs et à les recycler (cf. *thread pool* [Ling00]): une fois sa tâche terminée, un *thread* se place en attente pour un autre travail, les demandes d'exécution de tâches sont alors centralisées pour être affectées aux *threads* disponibles. Cette approche peut aussi s'avérer coûteuse car elle nécessite d'abstraire la notion de tâche, généralement sous la forme d'une classe abstraite ou d'un pointeur de fonction.

La stratégie de parallélisation est souvent déployée intégralement soit à la compilation, soit à l'exécution. Avec la métaprogrammation, il est possible d'élaborer une approche en deux étapes (e.g. [Said09]). Dans notre exemple, une forme de régularité de la solution est identifiable, quels que soient les nombres d'itérations et de coeurs: chaque *thread* effectue potentiellement une ou plusieurs itérations de la boucle externe, puis éventuellement une ou plusieurs itérations de la boucle interne. Un code pour ce travail peut être généré par métaprogrammation générique, sans abstraction et donc sans surcoût significatif à l'exécution. La séquence de travail d'un *thread* est donc planifiée à l'avance, il suffit de déterminer, une fois les informations manquantes connues, le nombre de *threads* adéquat, de leur communiquer les données à traiter et de les synchroniser pour les guider dans les différentes étapes de leur travail.

Plus généralement, la métaprogrammation générique devrait permettre de transformer un squelette en un graphe, afin d'exploiter les nombreux algorithmes existants pour cette structure et produire un code parallèle adapté. Plusieurs formes de graphe peuvent représenter un squelette, en fonction des caractéristiques de sa structure et de l'architecture ciblée [Sinn00]: graphe de dépendance, graphe de tâches, graphe de flot de données... Ces différents modèles permettent de décrire les tâches avec éventuellement leur temps d'exécution, les dépendances entre tâches (notion de précédence), les temps de communication entre tâches (incontournables dans le cas d'une architecture à mémoire distribuée), les "distances" de dépendance (pour représenter les itérations et en particulier les dépendances inter-itération [Yang97a], cf. squelette *SerialLoop*). Il peut être nécessaire aussi d'y ajouter une représentation des accès mémoire (e.g. dans le cas d'une architecture à mémoire partagée non uniforme NUMA [Wolk93] ou hybride CPU-GPU [Kirs09]).

Des transformations peuvent être appliquées sur ces graphes, notamment pour adapter le degré de parallélisme (e.g. réorganiser des tâches parallèles en série, cf. notre exemple), ou bien changer de type de graphe (e.g. transformer un graphe de flot de données en un graphe de tâches [Sinn00]) en fonction de l'architecture ciblée. Dans le cas d'un graphe de dépendance, des interblocages peuvent être détectés en recherchant des circuits (à l'instar des interdépendances entre variables d'intégration, cf. section précédente). Enfin, des algorithmes d'optimisation peuvent être envisagés pour l'affectation et l'ordonnancement des tâches sur les unités de calcul (e.g. [Kwok99, Hend00]).

7.4. Conclusion

Deux applications en optimisation et simulation ont été présentées pour lesquelles il est possible d'exploiter des informations sur la structure du programme au moment de la compilation afin de produire un code plus efficace. La métaprogrammation orientée concept, et notamment les patrons d'expressions orientés concept, peuvent faciliter le développement des structures statiques et des métaprogrammes nécessaires à cette opération.

Ces applications montrent la nécessité de disposer de représentations statiques sous forme de graphes et d'outils algorithmiques accessibles à la compilation pour les manipuler: vérification (e.g. détection de circuit), transformation (e.g. opérations de réduction du graphe), optimisation (e.g. ordonnancement de tâches)... [Wood11] propose déjà une modélisation statique d'un graphe reposant sur des structures de données de la bibliothèque MPL (*Boost Metaprogramming Library*) qui est considérée comme une version statique de la STL.

Néanmoins, nous pensons que les concepts peuvent faciliter l'élaboration de métaprogrammes par leur capacité à introduire de la sémantique et à structurer ainsi la spécialisation de patron. Il semble donc pertinent d'étudier la conception de structures statiques de graphe avec une approche orientée concept. L'objectif est de fournir une bibliothèque d'algorithmes d'optimisation pour les graphes sous la forme de métaprogrammes équivalents aux algorithmes dynamiques traditionnels (plus court chemin, flot de coût minimum...), avec la flexibilité d'une bibliothèque comme *Boost Graph Library*.

Conclusion

CONCLUSION

Bilan

Les travaux présentés dans ce mémoire visent avant tout à proposer des solutions performantes pour l'optimisation et la simulation. Des algorithmes d'optimisation rapides, pour un usage proche du temps réel, ont ainsi été conçus pour la synchronisation de documents hypermédia. Des métaheuristiques ont été développées pour maximiser les performances d'une base de données sous des contraintes de budget, avec une vitesse d'exécution permettant d'envisager un réajustement périodique de l'utilisation des ressources de l'application en fonction de la charge du nuage qui l'héberge.

Un couplage optimisation-simulation original a été proposé pour réduire significativement le nombre de simulations par rapport à un couplage classique, et permettre aussi d'améliorer la qualité pratique des solutions relativement à une optimisation théorique seule. Une méthodologie pour l'analyse et la minimisation de la vulnérabilité d'un système a été introduite afin de réduire le nombre de simulations nécessaires à l'optimisation, à partir d'une approximation du modèle de simulation du système étudié par des surfaces de réponse.

Concernant la conception logicielle, des solutions à la fois flexibles et performantes ont été recherchées. Des *frameworks* génériques (au sens large) de simulation à événements discrets pour des flux discrets dans un réseau, et de simulation par intégration numérique permettant la composition de modèles, ont été développés. Ces solutions reposent principalement sur la programmation objet et en partie sur la programmation générique. Si la programmation objet permet une grande abstraction et beaucoup de flexibilité dans le code, il s'avère dans certains cas que ses mécanismes introduisent un surcoût significatif à l'exécution. L'analyse des raisons de cette inefficacité et la proposition de patrons de conception exploitant la programmation générique ont permis de concevoir une bibliothèque générique pour la recherche opérationnelle.

A partir d'informations statiques (i.e. connues à la compilation et constantes à l'exécution) sur la structure d'un code, il est aussi possible d'exploiter la programmation générique via des techniques de métaprogrammation afin d'effectuer à la compilation des tâches élaborées et d'éviter ainsi certains traitements à l'exécution. Pour faciliter un tel développement, la programmation générique doit être améliorée, notamment à travers les "concepts". Cette notion n'étant pas encore présente en C++, une implémentation partielle des concepts (principalement le mécanisme de spécialisation de patron) a été réalisée sous la forme d'une bibliothèque générique. Les patrons d'expressions, une technique de métaprogrammation souvent utilisée pour capter des informations statiques, a été revisitée à l'aide des concepts et a permis de montrer le potentiel de ces derniers pour la production de code générique plus concis et plus fiable.

Perspectives

De ces expériences émergent deux besoins majeurs qui vont guider nos futurs travaux: l'utilisation de la phase de compilation pour préparer au mieux un code en fonction des informations statiques dont on dispose, et la conception d'abstractions logicielles permettant de masquer la complexité du développement d'applications parallèles et de s'affranchir de l'architecture matérielle et logicielle sous-jacente. La métaprogrammation par les génériques semble être un outil adapté à ces objectifs, mais à condition de pouvoir faciliter à la fois le développement de métaprogrammes, ce que nous pensons possible notamment grâce aux concepts, et la captation d'informations statiques sur la structure du code, ce qui est souvent réalisé à l'aide de patrons d'expressions dont l'exploitation à la compilation doit encore être améliorée.

Dans la continuité des travaux déjà menés, nous pensons étudier deux types d'applications: le couplage de modèles de simulation par intégration numérique où la problématique est d'identifier les dépendances entre variables, de détecter les éventuels circuits et d'ordonner correctement la séquence de calcul; et la parallélisation de métaheuristiques qui nécessitent souvent le traitement (évaluation, résolution d'un sous-problème d'optimisation...) d'un nombre important de solutions, avec une structure algorithmique adaptée à un parallélisme sur plusieurs niveaux.

Les patrons d'expressions orientés concept doivent être améliorés afin de répondre à certaines problématiques comme l'*aliasing* de variables, l'optimisation de boucles et la réécriture d'expressions. Pour le couplage de modèles de simulation par intégration numérique, une représentation des équations reposant sur les patrons d'expressions semble adaptée, avec la particularité de modéliser et manipuler un ensemble d'expressions. Concernant la parallélisation automatique de métaheuristiques, une approche par squelette algorithmique est envisagée, avec une modélisation par programmation générique similaire aux patrons d'expressions, ce qui permet d'envisager des outils communs aux deux applications. Il semble en effet pertinent de disposer d'algorithmes de manipulation de graphes à la compilation: les dépendances entre variables des modèles de simulation, comme la structure d'une métaheuristique, peuvent être modélisées par un graphe statique qui sera analysé et transformé par métaprogrammation générique.

BIBLIOGRAPHIE

BIBLIOGRAPHIE

[Abra04] David Abrahams et Aleksey Gurtovoy. **C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond**. Addison-Wesley, 2004.

[Ahuj03] Ravindra K. Ahuja, Dorit S. Hochbaum et James B. Orlin. **Solving the Convex Cost Integer Dual Network Flow Problem**. Dans *Management Science*, volume 49, pages 950–964, 2003.

[Ahuj93] Ravindra K. Ahuja, Thomas L. Magnanti et James B. Orlin. **Network Flows - Theory, Algorithms, and Applications**. Prentice Hall, 1993.

[Alex01] Andrei Alexandrescu. **Modern C++ Design: Generic Programming and Design Patterns Applied**. Addison-Wesley, 2001.

[Aust99] Matthew H. Austern. **Generic Programming and the STL: Using and Extending the C++ Standard Template Library**. Addison-Wesley, 1999.

[Azad92] F. Azadivar. **A Tutorial on Simulation Optimization**. Dans *Proceedings of the Winter Simulation Conference*, pages 198–204, 1992.

[Bach03a] Bruno Bachelet. **Modélisation et optimisation de problèmes de synchronisation dans les documents hypermédia**. Thèse de Doctorat, Université Blaise Pascal, Clermont-Ferrand, France, 2003.

[Bach03b] Bruno Bachelet et Philippe Mahey. **Minimum Convex-Cost Tension Problems on Series-Parallel Graphs**. Dans *RAIRO Operations Research*, volume 37-4, pages 221–234. EDP Sciences, 2003.

[Bach04a] Bruno Bachelet, Christophe Duhamel, Philippe Mahey et Luiz Fernando Soares. **Hypermedia Synchronization: Modeling and Optimization with Graphs**. Dans *Information Processing: Recent Mathematical Advances in Optimization and Control*, pages 49–62. Presses de l'Ecole des Mines de Paris, 2004.

[Bach04b] Bruno Bachelet et Philippe Mahey. **Minimum Convex Piecewise Linear Cost Tension Problem on Quasi-k Series-Parallel Graphs**. Dans *4OR: a Quarterly Journal of Operations Research*, volume 2-4, pages 275–291. Springer-Verlag, 2004.

[Bach06a] Bruno Bachelet, Antoine Mahul et Loïc Yon. **Designing Generic Algorithms for Operations Research**. Dans *Software: Practice and Experience*, volume 36-1, pages 73–93. John Wiley & Sons, 2006.

[Bach07a] Bruno Bachelet, Philippe Mahey, Rogério Rodrigues et Luiz Fernando Soares. **Elastic Time Computation in QoS-Driven Hypermedia Presentations**. Dans *ACM Multimedia Systems Journal*, volume 12, pages 461–478. Springer-Verlag, 2007.

[Bach07b] Bruno Bachelet et Loïc Yon. **Model Enhancement: Improving Theoretical Optimization with Simulation**. Dans *Simulation Modelling Practice and Theory*, volume 15-6, pages 703–715. Elsevier Science, 2007.

[Bach09] Bruno Bachelet et Christophe Duhamel. **Aggregation Approach for the Minimum Binary Cost Tension Problem**. Dans *European Journal of Operations Research*, volume 197-2, pages 837–841. Elsevier Science, 2009.

[Bach12] Bruno Bachelet. **Logical Operations on Concepts in the C4TS++ Library**. Rapport Technique, LIMOS, Université Blaise Pascal, Clermont-Ferrand, France, 2012.

[Bach13] Bruno Bachelet, Antoine Mahul et Loïc Yon. **Template Metaprogramming Techniques for Concept-Based Specialization**. Dans *Scientific Programming*, volume 21, pages 43–61. IOS Press, 2013.

[Bach15] Bruno Bachelet et Loïc Yon. **Schrödinger Effect of Templates**. Rapport Technique, LIMOS, Université Blaise Pascal, Clermont-Ferrand, France, 2015.

[Bach17] Bruno Bachelet et Loïc Yon. **Designing Expression Templates with Concepts**. Dans *Software: Practice and Experience*. John Wiley & Sons, 2017. A paraître.

[Bari03] Xavier Baril et Zohra Bellahsène. **Selection of Materialized Views: a Cost-Based Approach**. Dans *Lecture Notes in Computer Science*, volume 2681, pages 665–680. Springer-Verlag, 2003.

[Bart96b] John J. Barton et Lee R. Nackman. **Algebra for C++ Operators**. Dans *C++ Gems*, pages 501–514. SIGS Books, 1996.

[Bass98] Federico Bassetti, Kei Davis et Dan Quinlan. **C++ Expression Templates Performance Issues in Scientific Computing**. Dans *Parallel Processing Symposium*, pages 635–639, 1998.

[Berg62] C. Berge et A. Ghoula-Houri. **Programmes, jeux et réseaux de transport**. Dunod, 1962.

[Bracha04] Gilad Bracha. **Generics in the Java Programming Language**. Rapport Technique, Sun Microsystems, 2004.

[Béru09] Jean-François Bérubé, Michel Gendreau et Jean-Yves Potvin. **An Exact Epsilon-Constraint Method for Bi-objective Combinatorial Optimization Problems: Application to the Traveling Salesman Problem with Profits**. Dans *European Journal of Operational Research*, volume 194, pages 39–50. Elsevier Science, 2009.

[Buch92] M. Cecelia Buchanan et Polle T. Zellweger. **Specifying Temporal Behavior in Hypermedia Documents**. Dans *European Conference on Hypertext '92*, pages 262–271, 1992.

[Burk05] Edmund K. Burke et Graham Kendall. **Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques**. Springer-Verlag, 2005.

[Catt92] Dirk G. Cattrysse et Luk N. Van Wassenhove. **A Survey of Algorithms for the Generalized Assignment Problem**. Dans *European Journal of Operational Research*, volume 60-3, pages 260–272. Elsevier Science, 1992.

[Cell05] François E. Cellier et Ernesto Kofman. **Continuous System Simulation**. Springer-Verlag, 2005.

[Cole04] Murray Cole. **Bringing Skeletons Out of the Closet: a Pragmatic Manifesto for Skeletal Parallel Programming**. Dans *Parallel Computing*, volume 30, pages 389–406. Elsevier Science, 2004.

[Copl96] James O. Coplien. **Curiously Recurring Template Patterns**. Dans *C++ Gems*, pages 135–144. SIGS Books, 1996.

[Czar00a] Krzysztof Czarnecki et Ulrich Eisenecker. **Generative Programming: Methods, Tools, and Applications**. Addison-Wesley, 2000.

[Datt99] Alak Kumar Datta et Ranjan Kumar Sen. **An Efficient Scheme to Solve Two Problems for Two-Terminal Series Parallel Graphs**. Dans *Information Processing Letters*, volume 71, pages 9–15. Elsevier Science, 1999.

[Deb02] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal et T. Meyarivan. **A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II**. Dans *IEEE Transactions on Evolutionary Computation*, volume 6-2, pages 182–197. IEEE, 2002.

[Duha03] Christophe Duhamel, Antoine Mahul et Alexandre Aussem. **Routing with Neural-Based QoS Constraints**. Dans *INOC Conference*, pages 201–206, 2003.

[Ehrg05] Matthias Ehrgott. **Multicriteria Optimization**. Springer-Verlag, 2005.

[Epps92] David Eppstein. **Parallel Recognition of Series-Parallel Graphs**. Dans *Information and Computation*, volume 98-1, pages 41–55, 1992.

[Falc06b] Joel Falcou, Jocelyn Sérot, Thierry Chateau et Jean-Thierry Lapresté. **QUAFF: Efficient C++ Design for Parallel Skeletons**. Dans *Parallel Computing*, volume 32-7, pages 604–615. Elsevier Science, 2006.

[Favr04] Jean-Marie Favre. **Towards a Basic Theory to Model Model Driven Engineering**. Dans *3rd Workshop in Software Model Engineering*, pages 262–271, 2004.

[Feo95] Thomas A. Feo et Mauricio G.C. Resende. **Greedy Randomized Adaptative Search Procedures**. Dans *Journal of Global Optimization*, volume 6, pages 109–134. Kluwer Academic Publishers, 1995.

[Font05] Sébastien Fontaine et Sébastien Barot. **Size and Functional Diversity of Microbe Populations Control Plant Persistence and Long-Term Soil Carbon Accumulation**. Dans *Ecology Letters*, volume 8, pages 1075–1087. Blackwell Publishing Ltd, 2005.

[Fost84] James Foster, Joel Greer et Erik Thorbecke. **A Class of Decomposable Poverty Measures**. Dans *Econometrica*, volume 52, pages 761–766. The Econometric Society, 1984.

[Fowl10] Martin Fowler. **Domain-Specific Languages**. Addison-Wesley, 2010.

[Gamm95] Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides. **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley, 1995.

[Garc03] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy G. Siek et Jeremiah Willcock. **A Comparative Study of Language Support for Generic Programming**. Dans *Proceedings of the 18th ACM SIGPLAN Conference OOSPLA*, pages 115–134, 2003.

[Gay06] Jean-Christophe Gay, Bruno Bachelet et Vincent Maire. **Coupling Numerical Integration Models: Granularity and Computational Sequence**. Rapport de Recherche RR06-11, INRA / FGEP, Clermont-Ferrand, France, 2006.

[Glov97] F. Glover et M. Laguna. **Tabu Search**. Kluwer Academic Publishers, 1997.

[Gonz10] Horacio González-Vélez et Mario Leyton. **A Survey of Algorithmic Skeleton Frameworks: High-Level Structured Parallel Programming Enablers**. Dans *Software: Practice and Experience*, volume 40-12, pages 1135–1160. John Wiley & Sons, 2010.

[Gosw02] Dhrubajyoti Goswami, Ajit Singh et Bruno R. Preiss. **Building Parallel Applications Using Design Patterns**. Dans *Advances in Software Engineering*, pages 243–265. Springer-Verlag, 2002.

[Gott08] Peter Gottschling et Andrew Lumsdaine. **Integrating Semantics and Compilation: Using C++ Concepts to Develop Robust and Efficient Reusable Libraries**. Dans *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, pages 67–75, 2008.

[Grau11] Anne-Isabelle Graux. **Modélisation des impacts du changement climatique sur les écosystèmes prairiaux. Voies d'adaptation des systèmes fourragers**. Thèse de Doctorat, Université Blaise Pascal, Clermont-Ferrand, France, 2011.

[Greg06b] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis et Andrew Lumsdaine. **Concepts: Linguistic Support for Generic Programming in C++**. Dans *Proceedings of OOPSLA'06*, pages 291–310, 2006.

[Greg07] Douglas Gregor, Bjarne Stroustrup, Jeremy Siek et James Widman. **Proposed Wording for Concepts (Revision 3)**. Rapport Technique, N2421=07-0281, ISO/IEC JTC 1, 2007.

[Hadj97] Malika Hadjiat et Jean François Maurras. **A Strongly Polynomial Algorithm for the Minimum Cost Tension Problem**. Dans *Discrete Mathematics*, volume 165/166, pages 377–394. Elsevier Science, 1997.

[Hard05] Jochen Härdtlein, Alexander Linke et Christoph Pflaum. **Fast Expression Templates: Object-Oriented High Performance Computing**. Dans *Lecture Notes in Computer Science*, volume 3515, pages 1055–1063. Springer-Verlag, 2005.

[Hard06b] Jochen Härdtlein, Alexander Linke et Christoph Pflaum. **Blocking Techniques with Fast Expression Templates**. Rapport Technique, Department Informatik, Universität Erlangen-Nürnberg, Erlangen, Germany, 2006.

[Hard10] Jochen Härdtlein, Christoph Pflaum, Alexander Linke et Carsten H. Wolters. **Advanced Expression Templates Programming**. Dans *Computing and Visualization in Science*, volume 13-2, pages 59–68. Springer-Verlag, 2010.

[Hend00] Bruce Hendrickson et Tamara G. Kolda. **Graph Partitioning Models for Parallel Computing**. Dans *Parallel Computing*, volume 26, pages 1519–1534. Elsevier Science, 2000.

[Herz97] A. Herz, E. Taillard et D. de Werra. **Tabu Search**. Dans *Local Search in Combinatorial Optimization*, pages 121–136. Princeton University Press, 1997.

[Hoch97] Dorit S. Hochbaum. **Approximation Algorithms for NP-Hard Problems**. PWS Publishing Company, 1997.

[Iglb12] Klaus Iglberger, Georg Hager, Jan Treibig et Ulrich Rüde. **Expression Templates Revisited: a Performance Analysis of Current Methodologies**. Dans *SIAM Journal on Scientific Computing*, volume 34-2, pages C42–C69, 2012.

[Jarv03] Jaakko Järvi, Jeremiah Willcock et Andrew Lumsdaine. **Concept-Controlled Polymorphism**. Dans *Lecture Notes in Computer Science*, volume 2830, pages 228–244. Springer-Verlag, 2003.

[Jaza98] Mehdi Jazayeri, Rüdiger Loos, David Musser et Alexander Stepanov. **Generic Programming**. Dans *Report of the Dagstuhl Seminar on Generic Programming*, 1998.

[Kahn62] Arthur B. Kahn. **Topological Sorting of Large Networks**. Dans *Communications of the ACM*, volume 5-11, pages 558–562, 1962.

[Kim95] Michelle Y. Kim et Junehwa Song. **Multimedia Documents with Elastic Time**. Dans *Multimedia '95*, pages 143–154, 1995.

[Kirs09] Wilfried Kirschenmann, Laurent Plagne et Stéphane Vialle. **Multi-Target C++ Implementation of Parallel Skeletons**. Dans *Proceedings of the 8th Workshop on Parallel Object-Oriented Scientific Computing*, pages 7:1–7:10, 2009.

[Kwok99] Yu-Kwong Kwok et Ishfaq Ahmad. **Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors**. Dans *ACM Computing Surveys*, volume 31-4, pages 406–471, 1999.

[Lard12a] Romain Lardy, Raphaël Martin, Bruno Bachelet, David Hill et Gianni Bellocchi. **Ecosystem Climate Change Vulnerability Assessment Framework**. Dans *International Congress on Environmental Modelling and Software*, pages 777–784, 2012.

[Lard12b] Romain Lardy, Anne-Isabelle Graux, Bruno Bachelet, David Hill et Gianni Bellocchi. **Steady-State Soil Organic Matter Approximation Model: Application to the Pasture Simulation Model**. Dans *International Congress on Environmental Modelling and Software*, pages 769–776, 2012.

[Lard13] Romain Lardy. **Calcul intensif pour l'évaluation de la vulnérabilité en utilisant une approche d'ingénierie dirigée par les modèles - Application à la vulnérabilité des prairies au changement climatique sous contraintes de plans d'expériences**. Thèse de Doctorat, Université Blaise Pascal, Clermont-Ferrand, France, 2013.

[Lard14] Romain Lardy, Bruno Bachelet, Gianni Bellocchi et David Hill. **Towards Vulnerability Minimization of Grassland Soil Organic Matter using Metamodels**. Dans *Environmental Modelling and Software*, volume 52, pages 38–50. Elsevier Science, 2014.

[Lasd70] Leon S. Lasdon. **Optimization Theory for Large Systems**. MacMillan, 1970.

[Lian99] Sheng Liang. **The Java Native Interface**. Addison-Wesley, 1999.

[Ling00] Yibei Ling, Tracy Mullen et Xiaola Lin. **Analysis of Optimal Thread Pool Size**. Dans *ACM SIGOPS Operating Systems Review*, volume 34-2, pages 42–55, 2000.

[Lipp96a] Stanley B. Lippman. **Inside the C++ Object Model**. Addison-Wesley, 1996.

[Lois98] Pierre Loiseau et Yannick Bergia. **Modélisation et simulation des flux d'azote et de carbone dans les sols prairiaux**. Rapport Technique, INRA / FGEP, Clermont-Ferrand, France, 1998.

[Luer03] Amy L. Luers, David B. Lobell, Leonard S. Sklar, C. Lee Addams et Pamela A. Matson. **A Method for Quantifying Vulnerability, Applied to the Agricultural System of the Yaqui Valley, Mexico**. Dans *Global Environmental Change*, volume 13, pages 255–267. Elsevier Science, 2003.

[Magn09] Alessandro Magnani et Stephen P. Boyd. **Convex Piecewise-Linear Fitting**. Dans *Optimization and Engineering*, volume 10-1, pages 1–17. Springer-Verlag, 2009.

[Mair13] Vincent Maire, Jean-François Soussana, Nicolas Gross, Bruno Bachelet, Loic Pagès, Raphaël Martin, Tanja Reinhold, Christian Wirth et David Hill. **Plasticity of Plant Form and Function Sustains Productivity and Dominance along Environment and Competition Gradients. A Modeling Experiment with GEMINI**. Dans *Ecological Modelling*, volume 254, pages 80–91. Elsevier Science, 2013.

[Mami12] Imene Mami et Zohra Bellahsène. **A Survey of View Selection Methods**. Dans *SIGMOD Record*, volume 41-1, pages 20–29. ACM, 2012.

[Mart00] Silvano Martello, David Pisinger et Paolo Toth. **New Trends in Exact Algorithms for the 0-1 Knapsack Problem**. Dans *European Journal of Operational Research*, volume 123, pages 325–332. Elsevier Science, 2000.

[Mart15] Rafael Martí, Vicente Campos, Mauricio G.C. Resende et Abraham Duarte. **Multiobjective GRASP with Path Relinking**. Dans *European Journal of Operational Research*, volume 240, pages 54–71. Elsevier Science, 2015.

[McKa79] Michael D. McKay, Richard J. Beckman et William J. Conover. **A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code**. Dans *Technometrics*, volume 21, pages 239–245. American Society for Quality, 1979.

[McNa00] Brian McNamara et Yannis Smaragdakis. **Static Interfaces in C++**. Dans *First Workshop on C++ Template Programming*, 2000.

[Medi04] Maira T. Medina, Celso C. Ribeiro et Luiz F.G. Soares. **Automatic Scheduling of Hypermedia Documents with Elastic Times**. Dans *Parallel Processing Letters*, volume 14-1, pages 45–59, 2004.

[Merk94] Yuri A. Merkuryev et Vladimir K. Visipkov. **A Survey of Optimization Methods in Discrete Systems Simulation**. Dans *Proceedings of the First Joint Conference of International Simulation Societies*, pages 104–110, 1994.

[Mlad97] Nenad Mladenovic et Pierre Hansen. **Variable Neighborhood Search**. Dans *Computers & Operations Research*, volume 24-11, pages 1097–1100. Elsevier Science, 1997.

[Muss89] David R. Musser et Alexander A. Stepanov. **Generic Programming**. Dans *Lecture Notes in Computer Science*, volume 358, pages 13–25. Springer-Verlag, 1989.

[Myer96] Nathan Myers. **A New and Useful Template Technique: Traits**. Dans *C++ Gems*, pages 451–457. SIGS Books, 1996.

[Nemh99] George L. Nemhauser et Laurence A. Wolsey. **Integer and Combinatorial Optimization**. Wiley-Interscience, 1999.

[Nguy12] Thi-Van-Anh Nguyen, Laurent d'Orazio, Sandro Bimonte et Jérôme Darmont. **Cost Models for View Materialization in the Cloud**. Dans *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, pages 47–54, 2012.

[ORio02] Martin J. O'Riordan. **Technical Report on C++ Performance**. Rapport Technique, International Standardization Working Group ISO/IEC JTC1/SC22/WG21, 2002.

[Pass11] Jonathan Passerat-Palmbach, Claude Mazel, Bruno Bachelet et David Hill. **ShoveRand: a Model-Driven Framework to Easily Generate Random Numbers on GP-GPU**. Dans *International Conference on High Performance Computing and Simulation*, pages 41–48, 2011.

[Perr13] Romain Perriot, Jérémy Pfeifer, Laurent d'Orazio, Bruno Bachelet, Sandro Bimonte et Jérôme Darmont. **Modèles de coût pour la sélection de vues matérialisées dans le nuage, application aux services Amazon EC2 et S3**. Dans *9èmes Journées Francophones sur les Entrepôts de Données et l'Analyse en Ligne*, volume B-9, pages 53–68. RNTI, Editions Hermann, 2013.

[Perr14] Romain Perriot, Jérémy Pfeifer, Laurent d'Orazio, Bruno Bachelet, Sandro Bimonte et Jérôme Darmont. **Cost Models for Selecting Materialized Views in Public Clouds**. Dans *International Journal of Data Warehousing and Mining*, volume 10-4, pages 1–25. IGI Global, 2014.

[Pla71] Jean-Marie Pla. **An Out-of-Kilter Algorithm for Solving Minimum Cost Potential Problems**. Dans *Mathematical Programming*, volume 1, pages 275–290, 1971.

[Pres92] William H. Press, Saul A. Teukolsky, William T. Vetterling et Brian P. Flannery. **Numerical Recipes in C - The Art of Scientific Computing, 2nd Edition**. Cambridge University Press, 1992.

[Reis15] Marco Túlio Reis Rodrigues, Rui Sá Shibasaki, Bruno Bachelet et Christophe Duhamel. **Estratégia bi-critério para um problema de escalabilidade em computação nas nuvens**. Dans *XLVII Brazilian Symposium of Operational Research*, pages 360–371, 2015.

[Ribe02] Celso C. Ribeiro et Eric Sanlaville. **On the Complexity of Scheduling with Elastic Times**. Rapport Technique, Computer Science Department, PUC-Rio, Rio de Janeiro, Brazil, 2002.

[Ried98] Marcel Riedo, Anton Grub, Marc Rosset et Jürg Fuhrer. **A Pasture Simulation Model for Dry Matter Production, and Fluxes of Carbon, Nitrogen, Water and Energy**. Dans *Ecological Modelling*, volume 105, pages 141–183. Elsevier Science, 1998.

[Said09] Tarik Saidani, Joel Falcou, Claude Tadonki, Lionel Lacassagne et Daniel Etiemble. **Algorithmic Skeletons within an Embedded Domain Specific Language for the CELL Processor**. Dans *18th International Conference on Parallel Architectures and Compilation Techniques*, pages 67–76, 2009.

[Scho95] Berry Schoenmakers. **A New Algorithm for the Recognition of Series Parallel Graphs**. Rapport Technique, No CS-59504, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1995.

[Siek00] Jeremy G. Siek et Andrew Lumsdaine. **Concept Checking: Binding Parametric Polymorphism in C++**. Dans *First Workshop on C++ Template Programming*, 2000.

[Siek02] Jeremy G. Siek, Lie-Quan Lee et Andrew Lumsdaine. **The Boost Graph Library: User Guide and Reference Manual**. Addison-Wesley, 2002.

[Sinn00] Oliver Sinnen et Leonel Sousa. **A Comparative Analysis of Graph Models to Develop Parallelising Tools**. Dans *Proceedings of the 8th IASTED International Conference on Applied Informatics*, page 832Ű838, 2000.

[Sous00a] J.F. Soussana, F. Teyssonneyre et J. Thiéry. **Un modèle dynamique d'allocation basé sur l'hypothèse d'une co-limitation de la croissance végétale par les absorptions de lumière et d'azote**. Dans *Fonctionnement des peuplements végétaux sous contraintes environnementales*, pages 87–116. INRA, France, 2000.

[Sous00b] J.F. Soussana, F. Teyssonneyre et J. Thiéry. **Un modèle simulant les compétitions pour la lumière et pour l'azote entre espèces herbacées à croissance clonale**. Dans *Fonction-*

nement des peuplements végétaux sous contraintes environnementales, pages 325–350. INRA, France, 2000.

[Sous12] Jean-François Soussana, Vincent Maire, Nicolas Gross, Bruno Bachelet, Loic Pagès, Raphaël Martin, David Hill et Christian Wirth. **GEMINI: a Grassland Model Simulating the Role of Plant Traits for Community Dynamics and Ecosystem Functioning. Parameterization and Evaluation**. Dans *Ecological Modelling*, volume 231, pages 134–145. Elsevier Science, 2012.

[Stei90] Ralf Steinmetz. **Synchronization Properties in Multimedia Systems**. Dans *IEEE Journal on Selected Areas of Communication*, volume 8-3, pages 401–412, 1990.

[Sutt13a] Andrew Sutton, Bjarne Stroustrup et Gabriel Dos Reis. **Concepts Lite: Constraining Templates with Predicates**. Rapport Technique, N3580, ISO/IEC JTC 1, 2013.

[Sutt15] Andrew Sutton. **Working Draft, C++ Extensions for Concepts**. Rapport Technique, N4361, ISO/IEC JTC 1, 2015.

[Tour10] Luc Touraille, Mamadou K. Traoré et David R.C. Hill. **Enhancing DEVS Simulation through Template Metaprogramming: DEVS-MetaSimulator**. Dans *Proceedings of the 2010 Summer Computer Simulation Conference*, pages 394–402, 2010.

[Turn03] B. L. Turner, Roger E. Kasperson, Pamela A. Matson, James J. McCarthy, Robert W. Corell, Lindsey Christensen, Noelle Eckley, Jeanne X. Kasperson, Amy Luers, Marybeth L. Martello, Colin Polsky, Alexander Pulsipher et Andrew Schiller. **A Framework for Vulnerability Analysis in Sustainability Science**. Dans *Proceedings of the National Academy of Sciences of the USA*, volume 100-14, pages 8074–8079, 2003.

[Vald82] Jacobo Valdes, Robert E. Tarjan et Eugène L. Lawler. **The Recognition of Series Parallel Digraphs**. Dans *SIAM Journal on Computing*, volume 11-2, pages 298–313, 1982.

[Vand03] David Vandevoorde et Nicolai M. Josuttis. **C++ Templates: the Complete Guide**. Addison-Wesley, 2003.

[Veld03] Todd L. Veldhuizen. **C++ Templates are Turing Complete**. Rapport Technique, Indiana University Computer Science, 2003.

[Veld96a] Todd L. Veldhuizen. **Using C++ Template Metaprograms**. Dans *C++ Gems*, pages 459–473. SIGS Books, 1996.

[Veld96b] Todd L. Veldhuizen. **Expression Templates**. Dans *C++ Gems*, pages 475–487. SIGS Books, 1996.

[Veld99] Todd L. Veldhuizen. **C++ Templates as Partial Evaluation**. Dans *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 13–18. ACM Press, 1999.

[Visé98] M. Visée, J. Teghem, M. Pirlot et E.L. Ulungu. **Two-Phases Method and Branch and Bound Procedures to Solve the Bi-objective Knapsack Problem**. Dans *Journal of Global Optimization*, volume 12, pages 139–155. Kluwer Academic Publishers, 1998.

[Vuic07] Nicolas Vuichard, Jean-François Soussana, Philippe Ciais, Nicolas Viovy, Christof Ammann, Pierluigi Calanca, John Clifton-Brown, Jürg Fuhrer, Mike Jones et Cécile Martin. **Estimating the Greenhouse Gas Fluxes of European Grasslands with a Process-Based Model: Model Evaluation from in Situ Measurements**. Dans *Global Biogeochemical Cycles*, volume 21-1, page GB1004, 2007.

[Wei01] Roland Weiss et Volker Simonis. **Exploring Template Template Parameters**. Dans *Lecture Notes in Computer Science*, volume 2244, pages 500–510. Springer-Verlag, 2001.

[Wini05] Wilfried Winiwarter. **The Interface Problem in Model Coupling: Examples from Atmospheric Science**. Dans *1st Open International Conference on Modeling and Simulation*, pages 43–47, 2005.

[Wolf07] Steffen Wolf et Peter Merz. **Evolutionary Local Search for the Super-Peer Selection Problem and the p-Hub Median Problem**. Dans *Lecture Notes in Computer Science*, volume 4771, pages 1–15. Springer-Verlag, 2007.

[Wolk93] Richard M. Wolski et John T. Feo. **Program Partitioning for NUMA Multiprocessor Computer Systems**. Dans *Journal of Parallel and Distributed Computing*, volume 19, pages 203–218. Elsevier Science, 1993.

[Wood11] Gordon Woodhull. **A Library for Graph Metaprogramming: Introducing MPL.Graph**. Dans *BoostCon'11*, 2011.

[Yang97a] Tao Yang et Cong Fu. **Heuristic Algorithms for Scheduling Iterative Task Computations on Distributed Memory Machines**. Dans *IEEE Transactions on Parallel and Distributed Systems*, volume 8-6, pages 608–622, 1997.

[Yon04a] Loïc Yon, Alain Quilliot et Christophe Duhamel. **Distance Minimization in Public Transportation Networks with Elastic Demands: Exact Model and Approached Methods**. Dans *Information Processing: Recent Mathematical Advances in Optimization and Control*, pages 259–268. Presses de l'Ecole des Mines de Paris, 2004.

[Yon05] Loïc Yon. **Modèles et outils pour la conception stratégique de réseaux de transports publics**. Thèse de Doctorat, Université Blaise Pascal, Clermont-Ferrand, France, 2005.