Un langage orienté parallèle pour modéliser des solveurs de contraintes

Alejandro Reyes Amaro, Eric Monfroy, Florian Richoux

To cite this version:

Alejandro Reyes Amaro, Eric Monfroy, Florian Richoux. Un langage orienté parallèle pour modéliser des solveurs de contraintes. Onzièmes Journées Francophones de Programmation par Contraintes (JFPC), 2015, Bordeaux, France. hal-01248171

HAL Id: hal-01248171
https://hal.archives-ouvertes.fr/hal-01248171

Submitted on 24 Dec 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Un langage orienté parallèle pour modéliser des solveurs de contraintes

Alejandro REYES AMARO Eric MONFROY Florian RICHOUX

LINA - UMR 6241, TASC - INRIA Université de Nantes, France.

{alejandro.reyes, eric.monfroy, florian.richoux}@univ-nantes.fr

Résumé

Cet article présente Parallel-Oriented Solver Language (POSL, prononcé "*puzzle*") : un système pour construire des méta-heuristiques interconnectées travaillant en parallèle. Le but de ce travail est d'obtenir un système pour facilement construire des solveurs et réduire l'effort de leur développement en proposant un mécanisme de réutilisation de code entre les différents solveurs. La nouveauté de cette approche porte sur le fait que l'on voit un solveur comme un ensemble de composants spécifiques, écris dans un langage orienté parallèle basé sur des opérateurs.

Un avantage de POSL est la possibilité de partager non seulement des informations mais aussi des comportements, permettant ainsi de modifier à chaud les solveurs. POSL permets aux composants d'un solveur d'être transmis et exécutés par d'autres solveurs. Il propose également une couche supplémentaire permettant de définir dynamiquement des connexions entre solveurs.

L'implémentation de POSL restant un travail en cours, cet article se concentre uniquement sur ses concepts.

1 Introduction

L'optimisation combinatoire a d'importantes applications dans des champs divers, tels que l'apprentissage automatique, l'intelligence artificielle et l'ingénierie logicielle. Dans certains cas, le but principal est de simplement trouver une solution, comme pour les *problèmes de satisfaction de contraintes (CSP)*. Une solution sera une affectation de valeur à chaque variable du problème afin de satisfaire chacune de ses contraintes. En d'autres mots : il s'agit de chercher une solution possible.

Les *CSP* trouvent beaucoup d'applications dans l'industrie. Pour cette raison, plusieurs techniques et méthodes peuvent être appliquées pour résoudre ces problèmes. Bien que ces techniques, telles que les méta-heuristiques par exemple, se montrent efficaces, les problèmes que l'on souhaite résoudre en pratique sont parfois trop vastes, c'est-à-dire, avec un espace de recherche trop grand, pour pouvoir être traité en un temps raisonnable.

Cependant, le développement de l'architecture des ordinateurs nous mène vers des machines massivement *multi/many cœurs*. Cette évolution, étroitement liée aux développements des super-calculateurs, a ouvert une nouvelle manière de trouver des solutions pour les problèmes d'optimisation combinatoire, réduisant les temps de résolution. Adaptive Search [3] est l'un des algorithmes les plus efficaces, montrant de très bonnes performances et passant à l'échelle de plusieurs centaines ou même milliers de cœurs. C'est un exemple de recherche locale dit *multi-walk*, c'est-à-dire d'algorithmes explorant l'espace de recherche en lançant plusieurs processus indépendants de recherche, avec ou sans communication. Pour Adaptive Search, une implémentation de multi-walk coopératif a été publiée dans [9]. Ces travaux ont montré l'efficacité du schéma parallèle multi-walk, c'est pourquoi nous avons orienté POSL vers celui-ci.

Ces dernières années, beaucoup d'efforts ont été fait en parallélisation de la programmation par contraintes. Dans ce champ de recherche, la bonne gestion des communications inter-processus, utiles pour partager des informations entre solveurs, est absolument critique pour garantir de bonnes performances. Dans [8], une idée proposée est d'inclure des composants bas niveau de raisonnement pour la résolution de problèmes SAT, afin d'ajuster dynamiquement la taille des clauses partagées pour réduire les potentiels engorgements de communication. L'interaction entre solveurs est appelée *coopération de solveurs*

et est très populaire dans ce domaine de recherche du fait de leurs bons résultats [11]. *Meta–S* est une implémentation du système proposé dans [5], permettant de traiter des problèmes à travers la coopération de solveurs sur des domaines à spécialiser. POSL propose un mécanisme de création de stratégies de communication indépendamment des solveurs, donnant la possibilité d'étudier facilement les processus de résolution et leurs résultats. Créer des solveurs implémentant différentes stratégies parallèles peut être aussi complexe que fastidieux. En ce sens, POSL donne la possibilité de faire de rapides prototypes de solveurs parallèles.

En programmation par contraintes, les résultats publiés se focalisent souvent sur l'amélioration de certaines métriques telles la vitesse de convergence ou la qualité des solutions par des solveurs connus. Cependant, ceci requière de profondes connaissances pour trouver le bon algorithme à appliquer au bon problème. Hyperion [2] est un système Java pour méta et hyper-heuristiques basé sur le principe d'interopérabilité, fournissant des patrons génériques pour une variété d'algorithmes de recherche locale et évolutionnaires, permettant des prototypages rapides avec la possibilité de réutiliser le code source. POSL vise à offrir ces avantages, mais en fournissant également un mécanisme permettant de définir des protocoles de communications entre solveurs.

Dans cet article, nous expliquons une méthodologie pour utiliser POSL afin de construire différents solveurs coopératifs basés sur le couplage de quatre composants indépendants : *operation module*, *open channel*, *computation strategy* et les *communication channels* ou *subscription*. Récemment, l'approche hybride mène à de très bons résultats en satisfaction de contraintes [4]. Ainsi, puisque les composants de solveurs peuvent être combinés, POSL est conçu pour obtenir simplement des ensembles de solveurs différents à exécuter en parallèle.

POSL propose, à travers un simple langage basé sur des opérateurs, une manière de créer une *computation strategy*, combinant des *composants* (à savoir des *operation modules* et *open channels*) définis au préalable. On retrouve une idée similaire dans [6] sans communication, où une méthode d'algorithme évolutionnaire utilisant un simple opérateur de composition est proposée, afin d'instancier automatiquement de nouvelles heuristiques de recherche locale pour SAT, en voyant ces dernières comme une combinaison de blocs indépendants. Une autre idée intéressante est proposée dans Templar [12], un système pour générer des algorithmes changeant de composants prédéfinis via une hyper-heuristique Dans la dernière phase de programmation via POSL, les solveurs peuvent être connectés à d'autres solveurs, en fonction de la structure de leurs *open channels*, leur permettant de partager non seulement des informations mais également des comportements, donnant la possibilité d'envoyer et recevoir leur *operation module*. Cette approche donne ainsi aux solveurs la possibilité d'évoluer durant leur exécution, en fonction d'un contexte parallèle.

Les travaux présentés dans ce papier se concentrent sur les concepts de POSL uniquement ; son implémentation est un travail en cours.

2 Problèmes ciblés

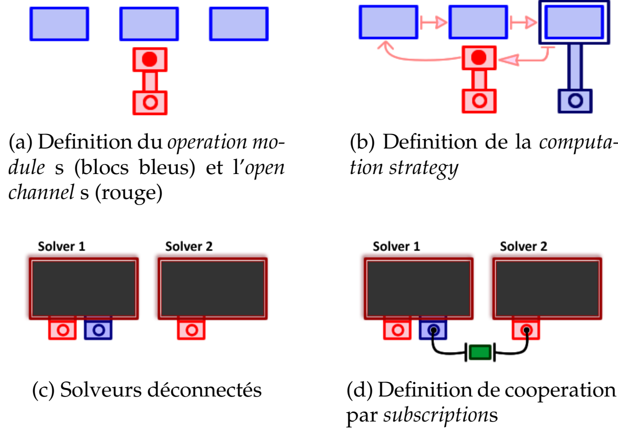
POSL est un système pour résoudre les CSP. Un CSP est défini par un triplet $\langle X, D, C\rangle$ où $X = \{x_1, x_2, \ldots, x_n\}$ est un ensemble fini de variables, $D = \{D_1, D_2, \ldots, D_n\}$, est l'ensemble des domaines de chaque variable dans X, et $C = \{c_1, c_2, \ldots, c_m\}$, est l'ensemble des contraintes. Chaque contrainte est définie selon un ensemble de variables et détermine les combinaisons possibles de leurs valeurs. Nous désignons par $\mathcal{P}$ un *CSP* donné.

Une configuration $s \in D_1 \times D_2 \times \cdots \times D_n$ est une combinaison de valeurs de chaque variable dans X. Nous écrivons que s est une solution de $\mathcal{P}$ si et seulement si s satisfait chacune des contraintes $c_i \in C$.

3 Solveurs parallèles POSL

POSL permet de construire des solveurs suivant différentes étapes :

1. L'algorithme du solveur considéré est exprimé via une décomposition en modules de calcul. Ces modules sont implémentés à la manière de *fonctions* séparées. Nous appelons *operation module* ces morceaux de calcul (figure 1a).
2. L'étape suivante consiste à décider quelles sont les types d'informations que l'on souhaite recevoir des autres solveurs. Ces informations sont encapsulées dans des composants appelés *open channel*, permettant de transmettre des données entre solveurs (figure 1a).
3. Une *stratégie générique* est codée à travers POSL, en utilisant les opérateurs fournis par le langage appliqués sur les composants donnés lors des étapes 1 et 2. Cette stratégie définie non seulement les informations échangées, mais détermine également l'exécution parallèle de composants. Lors de cette étape, les informations à partager sont transmises via les opérateurs ad-hoc. On peut voir cette étape comme la définition de la colonne vertébrale des solveurs.



(a) Definition du *operation module* s (blocs bleus) et l'*open channel* s (rouge)

(b) Definition de la *computation strategy*

(c) Solveurs déconnectés

(d) Definition de cooperation par *subscriptions*

FIGURE 1 – Construire des solveurs parallèles avec POSL

4. Les solveurs sont créés en instanciant la strategy, *operation module* et *open channel*, puis en les assemblant (figure 1d).

Les sous-sections suivantes expliquent en détail chacune des étapes ci-dessus.

3.1 Operation module

Un *operation module* est la plus basique et abstraite manière de définir un composant de calcul. Il reçoit une entrée, exécute un algorithme interne et retourne une sortie. Dans ce papier, nous utilisons ce concept afin de décrire et définir les composants de base d'un solveur, qui seront assemblés par la *computation strategy*.

Un *operation module* représente un morceau de l'algorithme du solveur qui est suceptible de changer au cours de l'exécution. Il peut être dynamiquement remplacé ou combiné avec d'autres *operation modules*, puisque les *operation modules* sont également des informations échangeables entre les solveurs. De cette manière, le solveur peut changer/adapter son comportement à chaud, en combinant ses *operation modules* avec ceux des autres solveurs. Ils sont représentés par des blocs bleus dans la figure 1.

Definition 1 ***(Operation Module)*** *Un operation module Om est une application définie par :*

$$Om : \mathcal{D} \to \mathcal{I} \qquad (1)$$

Dans (1), la nature de $\mathcal{D}$ et $\mathcal{I}$ dépend du type d'*operation module*. Ils peuvent être soit une configuration, ou un ensemble de configurations, ou un ensemble de valeurs de différents types de données, etc.

Soit une méta-heuristique de recherche locale, basée sur un algorithme bien connu, comme par exemple *Tabu Search*. Prenons l'exemple d'un *operation module* retournant le voisinage d'une configuration donnée, pour une certaine métrique de voisinage. Cet *operation module* peut être défini par la fonction suivante :

$$Om_{neighborhood} : D_1 \times D_2 \times \cdots \times D_n \to 2^{D_1 \times D_2 \times \cdots \times D_n}$$

où D_i représente la définition des domaines de chacune des variables de la configuration d'entrée.

3.2 Open channels

Les *open channel* sont les composants des solveurs en charge de la réception des informations communiquées entre solveurs. Ils peuvent interagir avec les *operation modules*, en fonction du *computation strategy*. Les *open channel* jouent le rôlé de prise, permettant aux solveurs de se brancher et de recevoir des informations. Il sont représentés en rouge dans la figure 8.

Un *open channel* peut recevoir deux types d'informations, provenant toujours d'un solveur tiers : des données et des *operation modules*. En ce qui concerne les *operation modules*, leur communication peut se faire via la transmission d'identifiants permettant à chaque solveur de les instancier.

Pour faire la distinction entre les deux différents types de *open channels*, nous appelons **Data Open Channels** les *open channels* responsables de la réception de données et **Object Open Channels** ceux s'occupant de la réception et de l'instanciation d'*operation modules*.

Definition 2 ***(Data Open Channel)*** *Un* Data Open Channel *Ch est un composant produisant une application définie comme suit :*

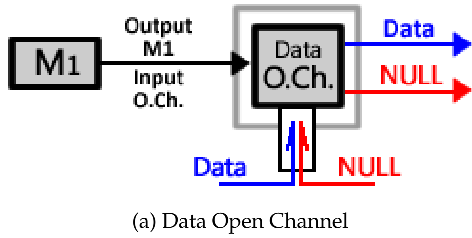
$$Ch : \mathcal{U} \to \mathcal{I} \qquad (2)$$

et retournant l'information $\mathcal{I}$ provenant d'un sovleur tiers, quelque soit l'entrée $\mathcal{U}$.

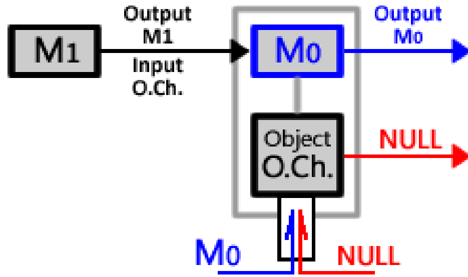
Definition 3 ***(Object Open Channel)*** *Si nous notons $\mathbb{M}$ l'espace de tous les operation modules de la définition 1, alors un* Object Open Channel *Ch est un composant produisant un operation module venant d'un solveur tiers défini ainsi :*

$$Ch : \mathbb{M} \to \mathbb{M} \qquad (3)$$

Puisque les *open channel* reçoivent des informations provenant d'autres solveurs sans pour autant avoir de contrôle sur celles-ci, il est nécessaire de définir l'information *NULL*, signifiant l'absence d'information. La figure 2 montre le mécanisme interne d'un *open channel*. Si un **Data Open Channel** reçoit une information, celle-ci est automatiquement retournée



(a) Data Open Channel



(b) Object Open Channel

FIGURE 2 – Mécanisme interne du *open channel*

(figure 2a, lignes bleues). Si un **Object Open Channel** reçoit un *operation module*, ce dernier est instancié et exécuté avec l'entrée de l'*open channel*, et le résultat est retourné (figure 2b, lignes bleues). Dans les deux cas, si aucune information n'est reçue, l'*open channel* retourne l'objet *NULL* (figure 2, lignes rouges).

3.3 Computation strategy

La *computation strategy* est le cœur du solver. Elle joint les *operation modules* et les *open channels* de manière cohérente, tout en leur restant indépendant. Ceci signifie qu'elle peut changer ou être modifiée durant l'exécution, sans altérer l'algorithme général et en respectant la structure du solveur. À travers la *computation strategy*, on peut décider également des informations à envoyer aux autres solveurs.

La *computation strategy* est définie par des opérations paramétriques. Tout d'abord, nous allons définir ce que sont ces opérations et les illustrées dans différents scénarios.

Definition 4 ***(Opérateurs de*** **POSL)** *Nous appelons* opérateur de POSL *un opérateur paramétrique défini comme suit :*

$$OP\{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_n\} : \mathcal{D}_{OP} \rightarrow \mathcal{I}_{OP} \qquad (4)$$

Nous appelons Compound Module *l'opération définie par (4), où chaque* $\mathcal{M}_i$ *est : i) Soit un operation module, ii) Soit un open channel, iii) Soit un compound module.*

Par soucis de simplification, nous utiliserons à partir de maintenant le terme de Module *pour désigner soit un operation module, un open channel ou un compound module.*

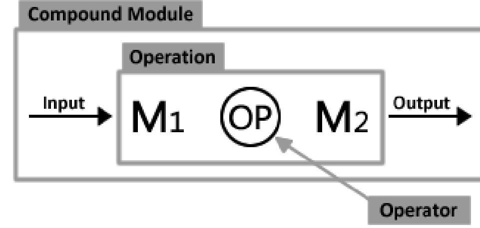


FIGURE 3 – Un *compound module*

Pour illustrer la définition 4, la figure 3 montre graphiquement les concepts d'*Operator*, *Operation* et *Compound Module*.

Chaque fois qu'une opération (4) est exécutée, l'opérateur *évalue* chaque *module* séparément, en affectant ses entrées à des *module* composant cette opération, c'est-à-dire que l'entrée de l'opérateur sera l'entrée de certains *modules* impliqués, voir de tous les *module* en fonction de l'opérateur. Après avoir évalué les *modules*, l'opérateur traite les données et retourne une sortie.

Le *processus d'évaluation* d'un *module* fonctionne de la manière suivante :

1. **Évaluation d'un Compound Module :** L'entrée de l'opérateur est passée au *compound module* et est évaluée.
2. **Évaluation d'un Operation Module :** Tout comme pour le *compound module*, l'entrée de l'operateur est passée à l'*operation module* et est évaluée.
3. **Évaluation d'un Open Channel :**
   — Data *open channel* : Le résultat de l'évaluation est la donnée réceptionnée, quelque soit l'entrée.
   — Object *open channel* : Le résultat de l'évaluation correspond à celui de l'évaluation de l'*operation module* réceptionné.

Comme nous le voyons par la définition 4, dans chaque opérateur fourni dans POSL, un ou plusieurs *compound modules* sont impliqués. En fonction de la nature de l'opérateur, ils peuvent être exécutés de manière séquentielle uniquement ou peuvent être exécutés en parallèle. POSL propose également des groupages d'opérations et laisse la possibilité de définir comment les *compound modules* impliqués vont être exécutés :

1. $[\mathrm{OP}\{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_n\}]$: Les *compound modules* $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_n$ seront exécutés séquentiellement.
2. $[\![\mathrm{OP}\{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_n\}]\!]_p$: Les *compound module* $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_n$ seront exécutés en parallèle si et seulement si *OP* supporte le parallélisme.

Dans le cas particulier où un des *compound modules* impliqués est un *open channel*, chaque opérateur gère l'information *NULL* à sa manière.

Afin de grouper des modules, nous utiliserons la notation $|.|$ comme un groupe générique qui pourra être indifféremment interprété comme $[.]$ ou comme $[\![\cdot]\!]_p$.

L'opérateur suivant nous permet d'exécuter deux modules séquentiellement, l'un après l'autre.

Definition 5 ***(Operator Sequential Execution)*** *Soient i)* $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *et ii)* $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$*. deux modules différents. L'*Operator Sequential Execution *est défini par l'opérateur paramétrique :*

$$\longmapsto (\mathcal{M}_2, \mathcal{M}_1) : \mathcal{D}_1 \rightarrow \mathcal{I}_2$$

et l'opération $|\mathcal{M}_1 \longmapsto \mathcal{M}_2|$ *retourne un compound module comme résultat de l'exécution de* $\mathcal{M}_1$ *suivi de* $\mathcal{M}_2$*.*

L'opération $|\mathcal{M}_1 \longmapsto \mathcal{M}_2|$ *peut être effectué si et seulement si* $\mathcal{I}_1 \subseteq \mathcal{D}_2$*.*

L'opérateur présenté dans la définition 5 est un exemple d'opérateur ne supportant pas une exécution parallèle de ses *compound modules* impliqués, puisque l'entrée du second *compound module* est la sortie du premier.

L'opérateur suivant est utile pour exécuter des modules séquentiels créant des branchements de calcul selon une condition booléenne :

Definition 6 ***(Operator Conditional Sequential Execution)*** *Soient i)* $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$*, ii)* $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$ *et iii)* $\mathcal{M}_3 : \mathcal{D}_3 \rightarrow \mathcal{I}_3$ *et trois modules différents. L'*Operator Conditional Sequential Execution *est défini par l'opérateur paramétrique :*

$$\underset{<.>}{\longmapsto} (\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3) : \{0,1\} \times \mathcal{D}_1 \rightarrow \mathcal{I}_o$$

et l'opération $\left|\mathcal{M}_1 \underset{<cond>}{\longmapsto} \{\mathcal{M}_2, \mathcal{M}_3\}\right|$ *retourne un compound module comme résultat de l'exécution séquentiel de* $\mathcal{M}_1$ *suivi de* $\mathcal{M}_2$ *si* $< cond >$ *est* **vrai***, ou de* $\mathcal{M}_3$ *le cas échéant.*

L'opération $\left|\mathcal{M}_1 \underset{<.>}{\longmapsto} \{\mathcal{M}_2, \mathcal{M}_3\}\right|$ *peut être appliquée si et seulement si* $\mathcal{I}_1 \subseteq \mathcal{D}_2 \cap \mathcal{D}_3$ *et* $\mathcal{I}_2 \cup \mathcal{I}_3 \subseteq \mathcal{I}_o$*.*

Nous pouvons exécuter séquentiellement des modules créant des boucles de calcul, en définissant les *compound modules* avec un autre opérateur conditionnel :

Definition 7 ***(Operator Cyclic Execution)*** *Soit* $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *un module. L'*Operator Cyclic Execution *est défini par :*

$$\circlearrowleft (\mathcal{M}_1) : \{0,1\} \times \mathcal{D}_1 \rightarrow \mathcal{I}_1$$

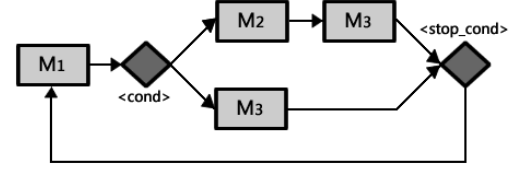


FIGURE 4

Algorithm 1: Code POSL de la figure 4

$$[\circlearrowleft < stop_cond > \{$$
$$\left[\mathcal{M}_1 \underset{<cond>}{\longmapsto} \{\mathcal{M}_2; [\mathcal{M}_2 \longmapsto \mathcal{M}_3]\}\right]$$
$$\}]$$

et l'opération $\circlearrowleft (< cond >) \{\mathcal{M}_1\}$ *retourne un compound module comme résultat de l'exécution séquentielle de* $\mathcal{M}_1$ *tant que* $< cond >$ *reste* **vrai***.*

L'opération $\circlearrowleft (< cond >) \{\mathcal{M}_1\}$ *peut être exécutée si et seulement* $\mathcal{I}_1 \subseteq \mathcal{D}_1$*.*

Dans la figure 4, on présente un exemple simple combinant des *modules* utilisant les opérateurs de POSL introduis ci-dessus. L'algorithme 1 montre le code correspondant. Cet exemple montre trois *operation modules* faisant partie d'un *compound module* avec pour but de générer une configuration initiale à partir de laquelle débutera un solveur de recherche locale. Nous avons ainsi :

— $\mathcal{M}_1$, générant une configuration aléatoire.
— $\mathcal{M}_2$, sélectionnant une variable aléatoire d'une configuration donnée, et recopiant sa valeur dans une autre variable.
— $\mathcal{M}_3$, stockant la meilleure configuration trouvée.

Dans cet exemple, l'*operation module* $\mathcal{M}_2$ est d'abord exécuté, suivi de $\mathcal{M}_3$ si tous les éléments de la configuration générée sont différents (< `cond` >). Sinon, seul l'*operation module* $\mathcal{M}_3$ est exécuté. Cette opération est répétée un certain nombre de fois (< `stop_cond` >).

Jusque là, nous avons supposé que nous pouvons seulement exécuter deux modules, en utilisant la sortie du premier comme entrée du second. Seulement parfois, des modules peuvent avoir besoin d'autres informations en entrée provenant de *modules* exécutés bien avant eux. Dans ce cas, il est nécessaire de donner des entrées supplémentaires :

Definition 8 ***(Operator Cartesian Product)*** *Soient i)* $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *et ii)* $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$*. deux modules différents. L'*Operator Cartesian Product *est défini par l'opérateur paramétrique :*

$$\bigotimes (\mathcal{M}_1, \mathcal{M}_2) : \mathcal{D}_o \rightarrow \mathcal{I}_1 \times \mathcal{I}_2$$

et l'opération $\left|\mathcal{M}_1 \otimes \mathcal{M}_2\right|$ *retourne un compound module comme résultat de l'exécution de* $\mathcal{M}_1$ *et de* $\mathcal{M}_2$*. Son image est le produit cartésien de l'image des opérandes.*

L'opération $\left|\mathcal{M}_1 \otimes \mathcal{M}_2\right|$ *peut être appliquée si et seulement si* $\mathcal{D}_o \subseteq \mathcal{D}_1 \cap \mathcal{D}_2$.

La figure 5 montre un scénario où il est nécessaire d'utiliser cet opérateur.

POSL offre la possibilité de faire muter les solveurs. En fonction de l'opération, un ou plusieurs *module* opérande(s) sera exécutée(s), mais seule la sortie de l'un d'entre eux sera retournée par le *compound module*. Nous présentons ces opérateurs dans deux définitions, groupant ceux qui exécutent uniquement un opérande de *module* (définition 9) et ceux exécutant les deux opérandes (définition 10).

Definition 9 *Soient i)* $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *et ii)* $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$*. deux module différents et une probabilité* ρ*. Nous pouvons alors définir l'opérateur paramétrique suivant :*

1. $\textcircled{\rho}(\rho, \mathcal{M}_1, \mathcal{M}_2) : \mathcal{D}_o \rightarrow \mathcal{I}_o$ *Où :*

$$\left|\mathcal{M}_1 \textcircled{\rho} \mathcal{M}_2\right| \text{ exécute } \begin{cases} \mathcal{M}_1 & \text{avec probabilité } \rho \\ \mathcal{M}_2 & \text{avec probabilité } (1-\rho) \end{cases}$$

2. $\textcircled{\vee}(\mathcal{M}_1, \mathcal{M}_2) : \mathcal{D}_o \rightarrow \mathcal{I}_o$ *Où :*

$$\left|\mathcal{M}_1 \textcircled{\vee} \mathcal{M}_2\right| \text{ exécute } \begin{cases} \mathcal{M}_1 & \text{si } \mathcal{M}_1 \text{ n'est pas } \textbf{null} \\ \mathcal{M}_2 & \text{sinon} \end{cases}$$

Ces opérations peuvent être appliquée si et seulement si $\mathcal{D}_o \subseteq \mathcal{D}_1 \cap \mathcal{D}_2$ *et* $\mathcal{I}_1 \cap \mathcal{I}_2 \subseteq \mathcal{I}_o$ *sont vérifiés.*

La définition suivante fait appelle aux notions de *parallélisme coopératif* et de *parallélisme compétitif*. Nous disons qu'il y a parallélisme coopératif quand deux unités de calcul ou plus s'exécutent simultanément, et que le résultat obtenu provient de la combinaison des résultats calculés par chaque unité de calcul (voir définitions 10.1 et 10.2). À l'opposé, nous considérons qu'il y a parallélisme compétitif lorsque le résultat obtenu est une solution ne provenant que d'un seul processus exécuté en parallèle ; en général le premier processus à terminer (voir définition 10.3).

Definition 10 *Soient i)* $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *et ii)* $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$*. . deux module différents. Soient également* o_1 *et* o_2 *les sorties de* $\mathcal{M}_1$ *et* $\mathcal{M}_2$ *respectivement, telles qu'il existe une relation d'ordre totale entre elles. Nous définissons alors les opérateurs paramétriques suivants :*

1. $\textcircled{M}(\mathcal{M}_1, \mathcal{M}_2) : \mathcal{D}_o \rightarrow \mathcal{I}_o$ *où :*

$$\left|\mathcal{M}_1 \textcircled{M} \mathcal{M}_2\right| \text{ retourne } \max\{o_1, o_2\}$$

2. $\textcircled{m}(\mathcal{M}_1, \mathcal{M}_2) : \mathcal{D}_o \rightarrow \mathcal{I}_o$ *où :*

$$\left|\mathcal{M}_1 \textcircled{m} \mathcal{M}_2\right| \text{ retourne } \min\{o_1, o_2\}$$

3. $\textcircled{\downarrow}(\mathcal{M}_1, \mathcal{M}_2) : \mathcal{D}_o \rightarrow \mathcal{I}_o$ *où :*

$$\left|\mathcal{M}_1 \textcircled{\downarrow} \mathcal{M}_2\right| \text{ retourne } \begin{cases} o_1 & \text{si } \mathcal{M}_1 \text{ termine en premier} \\ o_2 & \text{sinon} \end{cases}$$

Ces trois opérations peuvent être appliquées si et seulement si $\mathcal{D}_o \subseteq \mathcal{D}_1 \cap \mathcal{D}_2$ *et* $\mathcal{I}_1 \cap \mathcal{I}_2 \subseteq \mathcal{I}_o$.

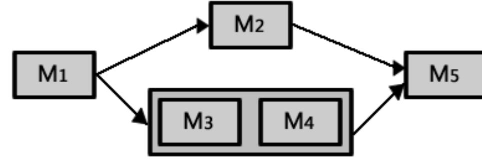


FIGURE 5

Algorithm 2: Code POSL de la figure 5

$$\mathcal{M}_1 \longmapsto \left[\!\left[\mathcal{M}_2 \otimes \left[\!\left[\mathcal{M}_3 \textcircled{\downarrow} \mathcal{M}_4 \right]\!\right]_p \right]\!\right]_p \longmapsto \mathcal{M}_5$$

Nous illustrons un de ces trois opérateurs dans l'exemple de la figure 5. Les modules rentrant en jeu sont :

— $\mathcal{M}_1$, retournant une configuration
— $\mathcal{M}_2$, calculant un paramètre de tolérance ϵ
— $\mathcal{M}_3$ et $\mathcal{M}_4$, calculant le voisinage $\mathcal{N}$ de la configuration provenant de $\mathcal{M}_1$. Ces deux *operation modules* calcule $\mathcal{N}$ de manière différente, mais ils sont combinés de façon à ce que seul la sortie du premier module terminant son calcul soit retournée.
— $\mathcal{M}_5$, sélectionnant une configuration dans $\mathcal{N}$ améliorant le coût global avec un tolérance ϵ, à la manière du *Threshold Accepting Method* [1]

Dans l'algorithme 2, les *modules* $\mathcal{M}_3$ et $\mathcal{M}_4$ sont exécutés en parallèle, mais seule la sortie du premier module à terminer son calcul sera retournée.

De la même manière, le *compound module* composé de $\mathcal{M}_3$ et $\mathcal{M}_4$ peut être exécuté en parallèle avec $\mathcal{M}_2$, parce qu'ils sont indépendants l'un de l'autre et que l'opérateur le permet. Il est important de souligner qu'ils peuvent être exécutés par l'opérateur $\otimes$, puisqu'ils reçoivent la même entrée.

Les opérateurs introduits par les définitions 9 et 10 sont très utiles en terme de partage d'informations entre solveurs, mais également en terme de partage de comportements. Si un des opérandes est un *open channel* alors l'opérateur peut recevoir l'*operation module* d'un autre solveur, donnant la possibilité d'instancier ce module dans le solveur le réceptionnant.

L'opérateur va soit instancier le module s'il n'est pas *null* et l'exécuter, soit exécuter le module donné par le second opérande.

POSL contient également des opérateurs pour manipuler des *ensembles*, ce qui est particulièrement utile pour les solveurs basés sur des populations.

Definition 11 *Soient i) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow 2^{\mathcal{I}_1}$ et ii) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow 2^{\mathcal{I}_2}$. . deux modules différents, où $2^{\mathcal{I}}$ désigne l'ensemble des sous-ensembles d'un certain type de données $\mathcal{I}$. Soient les ensembles V_1 et V_2 les sorties respectives de $\mathcal{M}_1$ et $\mathcal{M}_2$. Nous définissons les opérateurs paramétriques suivants :*

1. $\bigcirc\!\!\!\!\cup\,(\mathcal{M}_1, \mathcal{M}_2) : \mathcal{D}_o \rightarrow \mathcal{I}_o$ *où :*

   $\left|\mathcal{M}_1 \bigcirc\!\!\!\!\cup\, \mathcal{M}_2\right|$ *retourne* $V_1 \cup V_2$
2. $\bigcirc\!\!\!\!\cap\,(\mathcal{M}_1, \mathcal{M}_2) : \mathcal{D}_o \rightarrow \mathcal{I}_o$ *où :*

   $\left|\mathcal{M}_1 \bigcirc\!\!\!\!\cap\, \mathcal{M}_2\right|$ *retourne* $V_1 \cap V_2$
3. $\ominus(\mathcal{M}_1, \mathcal{M}_2) : \mathcal{D}_o \rightarrow \mathcal{I}_o$ *où :*

   $\left|\mathcal{M}_1 \ominus \mathcal{M}_2\right|$ *retourne* $V_1 \backslash V_2$

Les opérateurs peuvent s'appliquer si et seulement on a $\mathcal{D}_o \subseteq \mathcal{D}_1 \cap \mathcal{D}_2$ et $\mathcal{I}_1 \cap \mathcal{I}_2 \subseteq \mathcal{I}_o$.

Un exemple classique d'*operation module* retournant des ensembles est l'algorithme calculant l'ensemble $\mathcal{N}_S$ des voisins d'une configuration donnée S dans une méthode de recherche locale. Prenons les *operation modules* suivants :

1. $\mathcal{M}_1$ qui, étant donnée une configuration S, calcule l'ensemble des configurations $\mathcal{N}_S^1 = \{S_i'\}$ où

   $$S_i' \in \mathcal{N}_S \Longleftrightarrow g(S_i, S) < \epsilon$$

   avec $\epsilon > 0$, et g une fonction de proximité.
2. $\mathcal{M}_2$ qui, étant donnée une configuration S, calcule l'ensemble des configurations $\mathcal{N}_S^2 = \{S_i'\}$ où chaque S_i' est obtenue en faisant varier la valeur de quelques variables de S choisies aléatoirement.

En partant des deux *operation modules* ci-dessus et en utilisant un des opérateurs présentés jusque-là, on peut créer un *compound module* pour calculer un voisinage de S étant orienté à la fois exploitation et exploration :

$$\ldots \longmapsto \left[\!\left[\mathcal{M}_1 \bigcirc\!\!\!\!\cup\, \mathcal{M}_2\right]\!\right]_p \longmapsto \ldots$$

Dans ce cas, nous avons $\mathcal{N}_S = \mathcal{N}_S^1 \cup \mathcal{N}_S^2$.

Maintenant, nous définissons les opérateurs nous permettant d'envoyer de l'information vers d'autres solveurs. Deux types d'envois sont possibles :

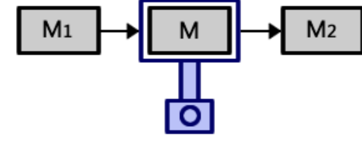


FIGURE 6

Algorithm 3: Code POSL de la figure 6 cas (a)
$\mathcal{M}_1 \longmapsto (\!|\mathcal{M}|\!)^o \longmapsto \mathcal{M}_2$

(a) on exécute un *operation module* et on envoie sa sortie,

(b) ou on envoie l'*operation module* lui-même.

Definition 12 *Soit $\mathcal{M} : \mathcal{D} \rightarrow \mathcal{I}$ un operation module. Nous avons les opérateurs paramétriques suivant :*

1. $(\!|.|\!)^o\,(\mathcal{M}) : \mathcal{D} \rightarrow \mathcal{I}$ *où :*

   $(\!|\mathcal{M}|\!)^o$ *exécute $\mathcal{M}$ et envoie la sortie*
2. $(\!|.|\!)^m\,(\mathcal{M}) : \mathcal{D} \rightarrow \mathcal{I}$ *où :*

   $(\!|\mathcal{M}|\!)^m$ *exécute $\mathcal{M}$ mais envoie l'operation module plutôt que sa sortie.*

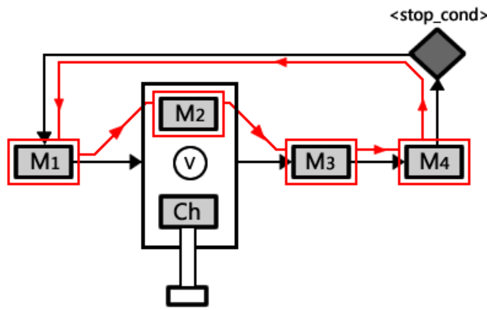
Algorithm 4: Code POSL de la figure 6 cas (b)
$\mathcal{M}_1 \longmapsto (\!|\mathcal{M}|\!)^m \longmapsto \mathcal{M}_2$

Les algorithmes 3 et 4 montrent du code utilisant POSL illustrant deux situations possibles de la configuration de la figure 6 : a) envoyer le résultat de l'exécution de l'*operation module* $\mathcal{M}$ b) envoyer l'*operation module* $\mathcal{M}$ lui-même.

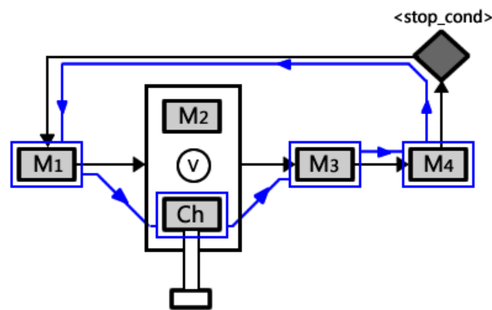
La figure 7 montre un autre exemple où l'on peut combiner un *open channel* avec l'*operation module* $\mathcal{M}_2$ à travers l'opérateur $\bigcirc\!\!\!\!\vee$. L'*operation module* $\mathcal{M}_2$ sera exécuté tant que l'*open channel* reste **null** (figure 7a, lignes rouges). Si un *operation module* a été reçu par l'*open channel*, il sera exécuté à la place de l'*operation module* $\mathcal{M}_2$ (figure 7b lignes bleues).

Avec les opérateurs présentés jusqu'ici, nous sommes en mesure de concevoir des stratégies (ou algorithme) de résolution d'un problème d'optimisation combinatoire donné. Une fois une telle stratégie définie, on peut changer les composants (*operation module* et *open channel*) auxquels elle fait appel, permettant ainsi d'implémenter différents solveurs à partir de la même stratégie mais composés de différents modules, du moment que ces derniers respectent la signature attendue, à savoir le typage des entrées et sorties.

Pour définir une *computation strategy*, nous utiliserons l'environnement suivant :



(a) Le solveur exécute son propre *operation module*



(b) Le solveur exécute l'*operation module* provenant d'un autre solveur

FIGURE 7 – Deux comportements différents dans le même solveur

```
1 St := cStrategy
2 oModule: < liste des types d'operation modules > ;
3 oChannel: < liste des types d'open channels > ;
4 {
5         < ...computation strategy... >
6 }
```

Avant de programmer la *computation strategy*, il est nécessaire de déclarer les types d'*operation modules* et d'*open channels* qui seront utilisés. Ceci est spécifié par `< liste des types d'operation modules >` et `< liste des types d'open channels>` respectivement. Après ces déclarations, le corps du code contient des opérateurs de POSL combinant des *module*. Un exemple simple est illustré par l'algorithme 5.

3.4 Définition de solveur

Une fois des *operation modules*, *open channels* et *computation strategy* définis, nous pouvons créer des solveurs en choisissant et instanciant les composants que nous allons intégrer au solveur. POSL fournit un environnement pour cela :

```
1 solver_k := solver
2 {
3 cStrategy: < strategy > ;
4 oModule: < liste des instances d'operation modules > ;
5 oChannel: < liste des instances d'open channels > ;
6 }
```

3.5 Définition des communications

Une fois que nous avons défini la stratégie de nos solveurs, l'étape suivante est de déclarer les canaux de communication reliant certains solveurs entre eux. Jusque là, les solveurs sont effectivement déconnectés, mais ont tout pour établir des communications (voir figure 1c). POSL propose à l'utilisateur une plate-forme pour facilement définir une *méta-strategie* à suivre par l'ensemble des solveurs.

Les communications sont établies en respectant les règles suivantes :

1. À chaque fois qu'un solveur envoie une information via les opérateurs $(\!|.|\!)^o$ ou $(\!|.|\!)^m$, il créé une *prise mâle de communication*
2. À chaque fois qu'un solveur contient un *open channel*, il créé une *prise femelle de communication*
3. Les solveurs peuvent être connectés entre eux en créant des *subscriptions*, reliant *prises mâles* et *femelles* (voir figure 8).

Avec l'opérateur $(\cdot)$, nous pouvons avoir accès aux *operation modules* envoyant une information et aux noms des *open channels* d'un solveur. Par exemple : $Solver_1 \cdot \mathcal{M}_1$ fournit un accès à l'*operation module* $\mathcal{M}_1$ du $Solver_1$ si et seulement si il est utilisé par l'opérateur $(\!|.|\!)^o$ (ou $(\!|.|\!)^m$), et $Solver_2 \cdot Ch_2$ fournit un accès à l'*open channel* Ch_2 de $Solver_2$. Nous définissons à présent les *subscriptions*.

Definition 13 *Supposons deux solveurs différents $Solver_1$ et $Solver_2$. Nous pouvons les connecter grâce à l'opération suivante :*

$$Solver_1 \cdot \mathcal{M}_1 \rightsquigarrow Solver_2 \cdot Ch_2$$

La connexion peut être définie si et seulement si :

1. *$Solver_1$ possède un operation module appelé $\mathcal{M}_1$ encapsulé dans l'opérateur $(\!|.|\!)^o$ ou $(\!|.|\!)^m$.*
2. *$Solver_2$ contient un open channel appelé Ch_2 recevant le même type d'informations envoyées par $\mathcal{M}_1$.*

La définition 13 n'exprime que la possibilité de définir statiquement des stratégies de communication. À terme, nous aimerions inclure dans POSL des opérateurs permettant plus de souplesse et d'expressivité en terme de communication entre solveurs, notamment à travers des stratégies dynamiques de communication.

4 Un solveur POSL

Dans cette section, nous allons expliquer la structure d'un solveur créé par POSL en nous appuyant sur

un exemple. Nous choisissons un des algorithmes les plus classiques : une méta-heuristique de recherche locale. Les méta-heuristiques ont une structure commune : ils commencent par initialiser quelques structures de données qui leur sont propres (une liste tabou pour le *Tabu Search* [7], une température pour le *Simulated Annealing* [10], etc). Puis, une configuration initiale s est générée, soit aléatoirement soit à travers une certaine heuristique. Viennent ensuite les étapes de résolution à proprement parlé, à savoir la sélection d'une nouvelle configuration s^* extraite parmi le voisinage $\mathcal{V}(s)$ de la configuration actuelle. Si s^* est une solution au problème $\mathcal{P}$ alors le processus s'arrête et s^* est retournée, sinon les structures de données sont actualisées, s^* est prise en compte ou non pour une nouvelle itération, en fonction de certains critères (comme par exemple la pénalisation d'optimum locaux du *Guided Local Search* [1]).

Les *restarts* constituent un mécanisme classique pour éviter de rester piégé dans un minimum local. Ils sont déclenchés lorsque l'on n'arrive plus à trouver une meilleure configuration, ou parfois après un *timeout*.

Les *operation modules* composant un algorithme de recherche locale sont décrits ci-dessous :

O. M. – 1 :	Génére une configuration s
O. M. – 2 :	Défini le voisinage $\mathcal{V}(S)$
O. M. – 3 :	Sélectionne $s^* \in \mathcal{V}(s)$
O. M. – 4 :	Évalue le critère d'acceptation pour s^*

Nous pouvons combiner les modules pour en créer de plus complexes. Par exemple, si l'on souhaite un *operation module* générant une configuration aléatoire mais capable de donner parfois une configuration prédéfinie (par exemple passée en paramètre) suivant une certaine probabilité, notre *operation module* pourrait être la combinaison via l'opérateur $\textcircled{\rho}$ de deux modules basiques s'occupant respectivement de générer une configuration aléatoire et de renvoyer une configuration donnée en paramètre.

Complexifions encore un peu notre solveur : supposons que nous avons une fonction de voisinage qui est orientée exploitation, mais que parfois nous souhaitons demander à d'autres solveurs de nous envoyer leur *operation module* calculant le voisignage, afin d'exécuter l'un de ces derniers dans notre solveur. Ceci est modélisable par l'*open channel* suivant :

O. Ch. – 1 :	Demande de l'*operation module* $\mathcal{V}(s)$.

Supposons que nous aimerions toujours communiquer à tout le monde notre configuration courante. Il est possible d'appliquer l'opérateur $(\!|.|\!)^o$ à l'*operation module* 4.

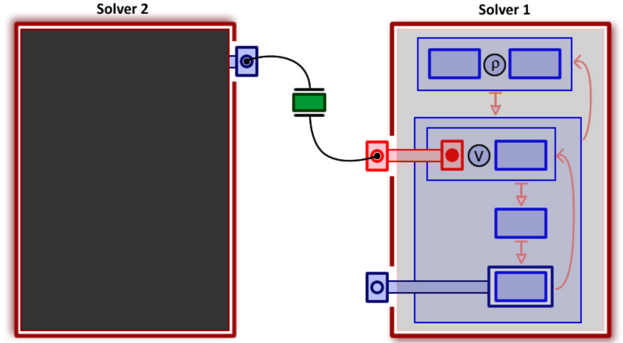


Figure 8 – Stratégie de coopération entre solveurs

Algorithm 5: Stratégie générale d'une méta–heuristique de recherche locale

$St \leftarrow$ **strategy**
oModule : $\mathcal{M}_1^a, \mathcal{M}_1^b, \mathcal{M}_2, \mathcal{M}_2, \mathcal{M}_4$;
oChannel : Ch_1 ;
{
[$\circlearrowleft$ (Iterations % 10000 != 0) {
$\left[\mathcal{M}_1^a \textcircled{\rho} \mathcal{M}_1^b\right] \longmapsto$
[$\circlearrowleft$ (Iterations % 1000 != 0) {
$\left[Ch_1 \textcircled{\vee} \mathcal{M}_2\right] \longmapsto \mathcal{M}_3 \longmapsto (\!|\mathcal{M}_4|\!)^o$
}]
}]
}

La figure 8 présente l'exemple ci-dessus. L'*open channel* de *Solver-1* est représenté en rouge. *Solver-1* contient un *open channel* **demandant** une fonction de voisinage. Il peut donc être connecté à *Solver-2* puisque ce dernier applique l'opérateur $(\!|.|\!)^o$ pour **envoyer** son *operation module* de voisinage. Une *subscription* (bloc vert) est alors créée définissant le canal de communication. La figure montre qu'aucun solveur ne contient d'*operation module* permettant d'envoyer une configuration, laissant ainsi le module chargé d'évaluer une configuration sans communication avec l'extérieur.

L'algorithme 5 montre un code POSL pour la *computation strategy* du solveur *Solver-1* décrit précédemment.

L'algorithme 6 montre une définition de solver. Nous mettons ici en place une *computation strategy*, suivit des instances des *modules* (*operation modules* et *open channels*).

Supposons qu'il existe un autre solver Σ_2 avec un *operation module* $\mathcal{M}_\mathcal{V}$ partageant sa fonction de voisinage. Nous pourrions le connecter avec le solveur Σ_1 comme le montre l'algorithme 7.

Algorithm 6: Definition d'un solveur utilisant une méta–heuristique de recherche locale

$\Sigma_1 \leftarrow$ **solver**
{
strategy : St
oModule : $m_1^a, m_1^b, m_2, m_3, m_4$
oChannel : ch_1
}

Algorithm 7: Définition de la communication entre solveurs

$\Sigma_2 \cdot \mathcal{M}_{\mathcal{V}} \rightsquigarrow \Sigma_1 \cdot Ch_1$

5 Conclusion

Dans ce papier, nous avons présenté POSL, un système pour construire des solveurs parallèles. Il propose une manière modulable pour créer des solveurs capables d'échanger n'importe quel type d'informations, comme par exemple leur comportement même, en partageant leurs *operation modules*. Avec POSL, de nombreux solveurs différents pourront être créés et lancés en parallèle, en utilisant une unique stratégie générique mais en instanciant différents *operation modules* et *open channels* pour chaque solveur.

Il est possible d'implémenter différentes stratégies de communication, puisque POSL fournit une couche pour définir les canaux de communication connectant les solveurs entre eux via des *subscriptions*.

L'implémentation de POSL reste un travail en cours. Notre principale tâche est de créer un système aussi général et souple que possible, permettant d'inclure de nouvelles fonctionnalités dans un futur proche. Notre but est d'obtenir une bibliothèque riche en *operation modules* et *open channels* à proposer aux utilisateurs, basés sur une étude approfondie des algorithmes classiques de résolution de problèmes combinatoires. Ainsi, nous espérons faciliter et accélérer la conception de nouveaux algorithmes.

En parallèle, nous prévoyons de développer de nouveaux opérateurs, en fonction des besoins des développeurs. Il est nécessaire, par exemple, d'améliorer la langage de définition de solveurs afin d'accélérer et simplifier encore le processus de production de solveurs. En outre, nous visons à étendre le langage de définition de communication pour permettre de créer de plus complexes et versatiles stratégies de communication, utiles pour étudier le comportement des solveurs parallèles.

À moyen terme, nous sommes intéressés par l'intégration de méthodes d'apprentissage automatique afin de permettre aux solveurs de s'adapter automatiquement, en fonction par exemple des résultats des solveurs qui leur sont voisins dans le réseau.

Références

[1] Ilhem Boussaïd, Julien Lepagnot, and Patrick Siarry. A survey on optimization metaheuristics. *Information Sciences*, 237 :82–117, July 2013.

[2] Alexander E.I. Brownlee, Jerry Swan, Ender Özcan, and Andrew J. Parkes. Hyperion 2. A toolkit for {meta- , hyper-} heuristic research. Technical report, 2014.

[3] Daniel Diaz, Florian Richoux, Philippe Codognet, Yves Caniou, and Salvador Abreu. Constraint-Based Local Search for the Costas Array Problem. In *Learning and Intelligent Optimization*, pages 378–383. Springer Berlin Heidelberg, 2012.

[4] Talbi El-Ghazali. Combining metaheuristics with mathematical programming, constraint programming and machine learning. *4or*, July 2013.

[5] Stephan Frank, Petra Hofstedt, and Pierre R. Mai. Meta-S : A Strategy-Oriented Meta-Solver Framework. In *FLAIRS Conference*, 2003.

[6] Alex S Fukunaga. Automated discovery of local search heuristics for satisfiability testing. *Evolutionary computation*, 16(1) :31–61, January 2008.

[7] Michel Gendreau and Jean-Yves Potvin. Tabu Search. In *Handbook of Metaheuristics*, pages 41–59. Springer US, 2010.

[8] Youssef Hamadi, Said Jaddour, and Lakhdar Sais. Control-Based Clause Sharing in Parallel SAT Solving. In *Autonomous Search*, pages 245–267. Springer Berlin Heidelberg, 2012.

[9] Danny Munera, Daniel Diaz, Salvador Abreu, and Philippe Codognet. A Parametric Framework for Cooperative Parallel Local Search. In *14th European Conference, EvoCOP*, pages 13–24, Granada, 2014.

[10] Alexander G. Nikolaev and Sheldon H. Jacobson. Simulated Annealing. In *Handbook of Metaheuristics*, pages 1–39. Springer US, 2010.

[11] Brice Pajot and Eric Monfroy. Separating Search and Strategy in Solver Cooperations. In *Perspectives of System Informatics*, pages 401–414. Springer Berlin Heidelberg, 2003.

[12] Jerry Swan and Nathan Burles. Templar - a framework for template-method hyper-heuristics. *Lecture Notes in Computer Science*, 9025 :205–216, 2015.