A Parallel-Oriented Language for Modeling Constraint-Based Solvers

Alejandro REYES AMARO Eric MONFROY
Florian RICHOUX
{alejandro.reyes, eric.monfroy, florian.richoux}@univ-nantes.fr
Université de Nantes, France

Abstract

This paper presents the Parallel-Oriented Solver Language (POSL, pronounced "*puzzle*"): a new framework to build interconnected meta-heuristic based solvers working in parallel. The goal of this work is to obtain a framework to easily build solvers and reduce the developing effort, by providing a mechanism for reusing code from other solvers. The novelty of this approach lies in looking at solver as a set of components with specific goals, written in a parallel-oriented language based on operators.

An interesting advantage of POSL is the possibility to share not only information, but also behaviors, allowing solvers modifications. POSL allows solver's components to be transferred and executed by other solvers. It provides an additional layer to dynamically define the connectivity between the solvers.

The implementation of POSL remains a work in progress, therefore this paper will focus on POSL concepts only.

1 Introduction

Combinatorial Optimization has important applications in several fields, including machine learning, artificial intelligence, and software engineering. In some cases, the main goal is only to find a solution, like for *Constraint Satisfaction Problems (CSP)*. A solution will be an assignment of variables satisfying the constraints set. In other words: finding one feasible solution.

*CSP*s find a lot of application in the industry. For that reason, many techniques and methods are applied to the resolution of these problems. Although many of these techniques, like meta–heuristics, have shown themselves to be effective, sometimes problems we want to solve are too large, i.e. the search space is huge, and in most cases intractable.

However, the development of computer architecture is leading us toward massively *multi/many–core* computers. This evolution, closely related to the development of the super–computing, has opened a new way to find solutions for these problems in a more feasible manner, reducing the search times. Adaptive Search [3] is one of the most efficient algorithm showing very good performances scaling to hundreds or even thousands of cores. It is an example of multi-walk local search method (i.e. algorithms that explore the search space by using independent search processes). For this algorithm, an implementation of a cooperative multi-walks strategy has been published in [9]. These works have shown the efficiency of independent multi-walk, that is why we have oriented POSL towards this parallel scheme.

In the last years, a lot of efforts have been made in parallel constraint programming. In this field, the inter-process communication, useful to share information between solvers, is one of the more critical issues. In [8], an idea to include low-level reasoning components in the SAT problems resolution is proposed, dynamically adjusting the size of shared clauses to reduce the possible blow up in communication. The interaction between solvers is called *solver cooperation* and it is very popular in this field due to their good results. In [11] is presented a paradigm that enables the user to properly separate strategies to follow to combine solvers applications in order to find the desired result, from the way that the search space is explored. *Meta–S* is a practical implementation of a theoretical framework proposed in [5], which allows to tackle problems, through the cooperation of arbitrary domain–specific constraint solvers. POSL provides a mechanism of creating solver–independent communication strategies, giving the possibility of easily studying solving processes and results.

During the development of constraint programming field, much research focuses on fitting existing algorithms , trying to improve some metrics such as convergence speed and solution quality. However, it is required a deep study to find the right algorithm for the right problem. Hyperion [2] is a Java framework for meta– and hyper–heuristics built with the principles of interoperability, generality by providing generic templates for a variety of local search and evolutionary computation algorithms, and efficiency allowing rapid prototyping, with the potential use of the same source code. POSL aims to offer these advantages, but provides also a mechanism to define communication protocols between solvers.

In this paper we explain a methodology to use POSL for easily build many and different cooperating solvers based on coupling four fundamental and independent components: *operation module*s, *open channel*s, the *computation strategy* and *communication channels* or *subscription*s. Recently, the hybridization approach, leads to very good results in constraint satisfaction [4]. For that reason, since the solver's component can be combined, POSL is designed to give the possibility to obtain sets of different solvers to be executed in parallel.

POSL provides, through a simple operator-based language, a way to create a *computation strategy*, combining already defined *components* (*operation module*s and *open channel*s). A similar idea was proposed in [6] without communication, where it is investigated an evolutionary approach that uses a simple composition operator, to automatically discover new local search heuristics for SAT, visualizing them as combinations of a set of building blocks. In the last phase of the coding process using POSL, solvers can be connected each others, depending on the structure of their *open channel*s, and this way, they can share not only information, but also their behavior, giving the possibility of send/receive their *operation module*s. This approach gives to the solvers the possibility to evolve during the execution.

The work that we present in this paper focuses on the concepts of POSL only. It is part of an ongoing investigation that foresees a complete implementation.

2 Target problems

POSL is a framework to tackle *Constraint Satisfaction Problems* (CSP). A CSP is defined by a triple $\langle X, D, C\rangle$ where $X = \{x_1, x_2, \ldots, x_n\}$ is a finite set of variables, $D = \{D_1, D_2, \ldots, D_n\}$, is the set of associated domains to each variable in X, and $C = \{c_1, c_2, \ldots, c_m\}$, is a set of constraints. Each constraint is defined involving a set of variables, and specifies the possible combinations of values for these variables. We denote by $\mathcal{P}$ a *CSP*.

A configuration $s \in D_1 \times D_2 \times \cdots \times D_n$, is a combinations of values for the variables of X. We say that s is a solution of $\mathcal{P}$ if and only if s mets all the constraints $c_i \in C$.

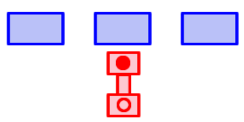
3 POSL parallel solvers

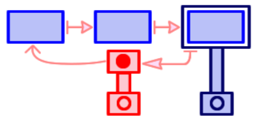
POSL proposes a solver construction platform, following different stages:

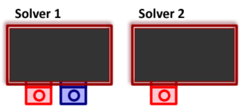
1. The solver algorithm is modeled by decomposing it into small pieces/modules of computation. After that, they are implemented as separated *functions*. We will give the name of *operation module* to these pieces/modules (Figure 1a).

2. The next step is to decide what information is interesting to receive from other solvers. This information is encapsulated into other objects called *open channel*s, allowing data transmission among solvers (Figure 1a).

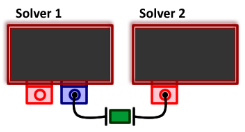
3. A generic strategy is coded through the Parallel-Oriented Solver Language based on operators it provides, using the mentioned components in the stages 1 and 2, allowing not only the information


(a) Defining *operation module*s (blue blocks) and *open channel*s (red shapes)


(b) Defining the *computation strategy*



(c) Disconected solvers



(d) Defining cooperation through *subscriptions*

Figure 1: Building parallel solvers with POSL

exchange, but also to execute the components in parallel. In this stage, the information that is interesting to share with other solvers, is sent using operators for that purpose. This will be the solver's backbone.

4. Solvers are defined by instantiating and connecting the *strategy*, *operation module*s and *open channel*s, and by connecting them each others (Figure 1d).

In this section we will explain in details each step.

3.1 Operation module

An *operation module* is the most basic and abstract way to define a piece of computation. It receives an input, executes an internal algorithm and gives an output. In this paper, we use this concept to describe and define one of the basic components of a solver, and they will be joined through *computation strategies*.

An *operation module* will represent a piece of the algorithm that can be changed or modified during the execution. They can be dynamically replaced by or combined with others *operation module*s, since the *operation module*s can be also information to be shared between solvers working in parallel. This way the solver can change/mutate its behavior during the execution, by combining its *operation module*s with *operation module*s from other solvers. They are represented by blue blocks in the Figure 4.

Definition 1 *An operation module Om is an mapping defined as follow:*

$$Om : \mathcal{D} \rightarrow \mathcal{I} \tag{1}$$

In (1), the nature $\mathcal{D}$ and $\mathcal{I}$ will depend on the type of *operation module*. They can be either a configuration, or a set of configurations, or a set of values of some data type, etc.

3.2 Open channels

The *open channel*s are the solver's components in charge of the information reception in the communication between solvers. They can interact with the *operation module*s, depending on the *computation strategy*. The *open channel*s will play the role of outlets, allowing solvers to be connected to them and receiving information. They are represented by red shapes in the Figure 4.

Definition 2 *An open channel Ch is an mapping defined as follow:*

$$Ch : \mathcal{D} \rightarrow \mathcal{D} \tag{2}$$

An *open channel* is an open door to outside to receive information. Inside this component will not exist any processing or data transformation. That is way in (2), the domain and the image are the same.

3.3 Computation strategy

The *computation strategy* is the solver's backbone. It joins the *operation module*s and the *open channel*s coherently. It is independent from the *operation module*s and *open channel*s used in the solver. It means they can be changed or modified during the execution, without altering the general algorithm, but still respecting the main structure.

Definition 3 *We call module to (and it is denoted by the letter $\mathcal{M}$):*

1. *An operation module or*
2. *An open channel or*
3. *$[\mathcal{M}_1$ OP $\mathcal{M}_2]$: The composition of two modules $\mathcal{M}_1$ and $\mathcal{M}_2$ to be executed sequentially, one after the other, and returning an output depending on the nature of the operator* OP*; or*
4. *$[\![\mathcal{M}_1$ OP $\mathcal{M}_2]\!]$: The composition of two modules $\mathcal{M}_1$ and $\mathcal{M}_2$ to be executed and returning an output depending on the nature of the operator* OP*. These two modules will be executed in parallel if and only if* OP *supports parallelism, (i.e. some modules will be executed sequentially although they were grouped this way); or sequentially otherwise.*

We denote the space of the modules by $\mathbb{M}$. We will call compound modules to the composition of modules described in 3 and 4.

Following we will define some operators included in POSL framework. In order to group modules, like in Definition 3(3) and 3(4), we will use $|.|$ as generic grouper.

The following operator allows us to execute two modules sequentially one after the other.

Definition 4 *Suppose two different modules: i) $\mathcal{M}_1 : \mathcal{D}_1 \to \mathcal{I}_1$ and ii) $\mathcal{M}_2 : \mathcal{D}_2 \to \mathcal{I}_2$. Then, we can obtain a compound module through the following parametric operator:*

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

The operation $|\mathcal{M}_1 \longmapsto \mathcal{M}_2|$ can be performed if and only if $\mathcal{I}_1 \subseteq \mathcal{D}_2$.

The operation $|\mathcal{M}_1 \longmapsto \mathcal{M}_2|$ returns a compound module as result of the sequential execution of $\mathcal{M}_1$ followed by $\mathcal{M}_2$.

The following operator is useful to execute modules sequentially creating bifurcations, subject to some boolean condition:

Definition 5 *Suppose three different modules: i) $\mathcal{M}_1 : \mathcal{D}_1 \to \mathcal{I}_1$, ii) $\mathcal{M}_2 : \mathcal{D}_2 \to \mathcal{I}_2$ and iii) $\mathcal{M}_3 : \mathcal{D}_3 \to \mathcal{I}_3$. Then, we can obtain a compound module through the following parametric operator:*

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

The operation $\left|\mathcal{M}_1 \underset{<.>}{\longmapsto} \{\mathcal{M}_2, \mathcal{M}_3\}\right|$ can be performed if and only if: $\mathcal{I}_1 \subseteq \mathcal{D}_2 \cap \mathcal{D}_3$ and $\mathcal{I}_2 \cup \mathcal{I}_3 \subseteq \mathcal{I}_o$.

The operation $\left|\mathcal{M}_1 \underset{<cond>}{\longmapsto} \{\mathcal{M}_2, \mathcal{M}_3\}\right|$ returns a compound module as result of the sequential execution of $\mathcal{M}_1$ followed by $\mathcal{M}_2$ if $< cond >$ is **true** *or by $\mathcal{M}_3$ otherwise.*

We can execute modules sequentially creating also cycles, by defining *compound modules* with another conditional operator:

Definition 6 *Suppose a module $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$. Then, we can obtain a cyclic compound module through the following parametric operator:*

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

The operation $\circlearrowleft (< cond >) \{\mathcal{M}_1\}$ can be performed if and only if $\mathcal{I}_1 \subseteq \mathcal{D}_1$.

The operation $\circlearrowleft (< cond >) \{\mathcal{M}_1\}$ returns a compound module as result of the sequential execution of $\mathcal{M}_1$ while $< cond >$ remains **true**.

In Figure 2 we present a simple example of how combining *module*s using POSL operators introduced above. Algorithm 1 shows the corresponding code.

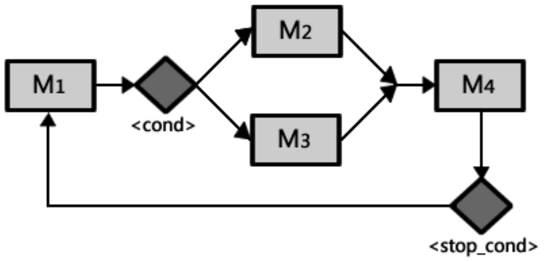


Figure 2

Algorithm 1: POSL code for Figure 2

```
[$\circlearrowleft < stop\_cond > \{$
    $\left[ \mathcal{M}_1 \xmapsto[<cond>]{} \{\mathcal{M}_2; \mathcal{M}_3\} \right] \longmapsto \mathcal{M}_4$
$\}$]
```

So far, we have assumed that we can only execute two modules, by using the output of the first one as input of the second one; but sometimes some modules need some other information as input coming from *module*(s) executed not right before them. In that case, it is necessary to give additional inputs:

Definition 7 *Suppose two different modules: i) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ and ii) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$. Then, we can obtain a compound module through the following parametric operator:*

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

The operation $\left| \mathcal{M}_1 \textcircled{\times} \mathcal{M}_2 \right|$ can be performed if and only if: $\mathcal{D}_o \subseteq \mathcal{D}_1 \cap \mathcal{D}_2$.

The operation $\left| \mathcal{M}_1 \textcircled{\times} \mathcal{M}_2 \right|$ returns a compound module as result of the execution of $\mathcal{M}_1$ and $\mathcal{M}_2$, and its image will be the cartesian product of its operand's images.

Figure 3 shows a scenario where it is necessary to use this operator.

POSL offers the possibility to give variability to the solvers. Depending on the operator, one or both operand *module* will be executed, but only the output of one of them will be returned by the *compound module*. We present these operators in two definitions, grouping those which execute only one *module* operand (Definition 8) and those which execute both (Definition 9).

Definition 8 *Suppose two different modules: $\mathcal{M}_1, \mathcal{M}_2 : \mathcal{D} \rightarrow \mathcal{I}$ and a probability ρ. Then, we can obtain compound modules through the following parametric operators:*

1. $\textcircled{\rho} (\rho, \mathcal{M}_1, \mathcal{M}_2) : \mathcal{D}_o \rightarrow \mathcal{I}_o$ *Where:* $\left| \mathcal{M}_1 \textcircled{\rho} \mathcal{M}_2 \right|$ *executes* $\begin{cases} \mathcal{M}_1 & \textit{with probability } \rho \\ \mathcal{M}_2 & \textit{with probability } (1 - \rho) \end{cases}$

2. $\textcircled{\vee} (\mathcal{M}_1, \mathcal{M}_2) : \mathcal{D}_o \rightarrow \mathcal{I}_o$ *Where:* $\left| \mathcal{M}_1 \textcircled{\vee} \mathcal{M}_2 \right|$ *executes* $\begin{cases} \mathcal{M}_1 & \textit{if } \mathcal{M}_1 \textit{ is not } \textbf{null} \\ \mathcal{M}_2 & \textit{otherwise} \end{cases}$

In the following Definition, the concepts of *cooperative parallelism* and *competitive parallelism* are implicitly included. We say that cooperative parallelism exists when two or more processes are running separately, they are independent, and the general result will be some combination of the results of all the involved processes (e.g. Definitions 9.1 and 9.2). On the other hand, competitive parallelism arise when the general result is the solution of the first process terminates (e.g. Definition 9.3).

Definition 9 *Suppose two different modules: $\mathcal{M}_1, \mathcal{M}_2 : \mathcal{D} \to \mathcal{I}$. Suppose also that o_1 and o_2 are the outputs of $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively, and that there exist some order criteria between them. Then, we can obtain a compound module through the following parametric operators:*

1. $\textcircled{M}(\mathcal{M}_1, \mathcal{M}_2) : \mathcal{D}_o \to \mathcal{I}_o$ *Where:* $\left|\mathcal{M}_1 \textcircled{M} \mathcal{M}_2\right|$ *returns* $\max\{o_1, o_2\}$
2. $\textcircled{m}(\mathcal{M}_1, \mathcal{M}_2) : \mathcal{D}_o \to \mathcal{I}_o$ *Where:* $\left|\mathcal{M}_1 \textcircled{m} \mathcal{M}_2\right|$ *returns* $\min\{o_1, o_2\}$
3. $\textcircled{\downarrow}(\mathcal{M}_1, \mathcal{M}_2) : \mathcal{D}_o \to \mathcal{I}_o$ *Where:* $\left|\mathcal{M}_1 \textcircled{\downarrow} \mathcal{M}_2\right|$ *returns* $\begin{cases} o_1 & \textit{if } \mathcal{M}_1 \textit{ ends first} \\ o_2 & \textit{otherwise} \end{cases}$

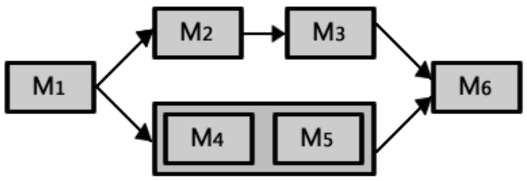


Figure 3

Algorithm 2: POSL code for Figure 3

$\mathcal{M}_1 \longmapsto \left[\left[\mathcal{M}_4 \textcircled{\downarrow} \mathcal{M}_5\right] \textcircled{\times} \left[\mathcal{M}_2 \longmapsto \mathcal{M}_3\right]\right]$
$\longmapsto \mathcal{M}_6$

The operators presented in Definitions 8 and 9 are very useful in terms of sharing not only information between solvers, but also *behaviors*. If one of the operands is an *open channel* it can receive an external solver's *operation module*, providing the opportunity to instantiate it in the current solver. The operator will either instantiate that module if it is not null, and execute it; or it will execute the other operand module otherwise.

Some others operators can be useful dealing with *sets*, especially for population–based solvers:

Definition 10 *Suppose two different modules: $\mathcal{M}_1, \mathcal{M}_2 : \mathcal{D} \to 2^{\mathcal{I}}$. Where $2^{\mathcal{I}}$ is to denote sets of some data type. Suppose also that the sets V_1 and V_2 are the outputs of the execution of $\mathcal{M}_1$ and $\mathcal{M}_2$. Then we can obtain a compound module through the following parametric operators:*

1. $\textcircled{\cup}(\mathcal{M}_1, \mathcal{M}_2) : \mathcal{D}_o \to \mathcal{I}_o$ *Where:* $\left|\mathcal{M}_1 \textcircled{\cup} \mathcal{M}_2\right|$ *returns* $V_1 \cup V_2$
2. $\textcircled{\cap}(\mathcal{M}_1, \mathcal{M}_2) : \mathcal{D}_o \to \mathcal{I}_o$ *Where:* $\left|\mathcal{M}_1 \textcircled{\cap} \mathcal{M}_2\right|$ *returns* $V_1 \cap V_2$
3. $\textcircled{-}(\mathcal{M}_1, \mathcal{M}_2) : \mathcal{D}_o \to \mathcal{I}_o$ *Where:* $\left|\mathcal{M}_1 \textcircled{-} \mathcal{M}_2\right|$ *returns* $V_1 \backslash V_2$

Now, we define the operators that allow us to send information to outside, i.e. others solvers. We can send two types of information: i) we can execute the *operation module* and send its output, or ii) we can send the *operation module* itself. . This utility is very useful in terms of sharing behaviors between solvers.

Definition 11 *Given an operation module $\mathcal{M} : \mathcal{D} \to \mathcal{I}$, we can define the following parametric operators:*

1. $(\!|.|\!)^o(\mathcal{M}) : \mathcal{D} \to \mathcal{I}$ *Where:* $(\!|\mathcal{M}|\!)^o$ *executes* $\mathcal{M}$ *and sends its output to outside*
2. $(\!|.|\!)^m(\mathcal{M}) : \mathcal{D} \to \mathcal{I}$ *Where:* $(\!|\mathcal{M}|\!)^m$ *executes* $\mathcal{M}$ *and also sends the operation module itself to outside*

In this stage, and using these operators, we can create the algorithm responsible for managing the components to find the solution for a given problem. These algorithms are fixed, but generic w.r.t. their components (*operation modules* and *open channels*). It means that we can build different solvers using the same strategy, but instantiating it with different components, as long as they have the right structure (i.e., input/output signature).

To define a *computation strategy* we will use the following environment:

```
St := cStrategy
 oModule: < list of operation modules types > ;
 oChannel: < list of open channels types > ;
{
   < ...computation strategy... >
}
```

Before coding the *computation strategy*, it is necessary to declare which types of *operation modules* and *open channels* will be used. They are placed in `< list of operation modules types >` and `< list of open channels types >` respectively. Inside the brackets, the field `< ...computation strategy... >` corresponds to the Parallel-Oriented Solver Language based on operators combining the *modules*. A clear example is provided in Algorithm 3.

3.4 Solver definition

With *operation modules*, *open channels* and *computation strategy* already defined, we can create the solvers by instantiating the mentioned components. POSL provides an environment to do that:

```
solver_k := solver
{
    cStrategy: < strategy > ;
    oModule: < list of operation modules instances > ;
    oChannel: < list of open channels instances > ;
}
```

3.5 Communication definition

Once we have defined the strategy to our solver, the next step is to declare the communication channels, i.e. connecting the solvers each others. Up to here, the solvers are disconnected, but they have everything to establish the communication (Figure 1c). In this last stage, POSL provides to the user a platform to easily define the cooperation *meta–strategy* to be followed by the set of solvers declared before.

The communication is established by following the next rules guideline:

1. Each time a solver sends any kind of information by using the operator $(\!|.|\!)^o$ or $(\!|.|\!)^m$, it creates a *communication jack*
2. Each time a solver places an *open channel* into its definition, it creates a *communication outlet*
3. Solvers can be connected each others by creating *subscriptions*, connecting *communication jacks* with *communication outlet* (see Figure 4).

With the operator $(\cdot)$ we can have access to the *operation modules* sending information and to the *open channel*'s names in a solver. For example: $Solver_1 \cdot \mathcal{M}_1$ provides access to the *operation module* $\mathcal{M}_1$ in $Solver_1$ if and only if it is affected by the operator $(\!|.|\!)^o$ (or $(\!|.|\!)^m$), and $Solver_2 \cdot Ch_2$ provides access to the *open channel* Ch_2 in $Solver_2$. Tacking this into account, we can define the *subscriptions*.

Definition 12 *Suppose two different solvers: $Solver_1$ and $Solver_2$. Then, we can connect them through the following operation:*

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

The connection can be defined if and only if:

1. *$Solver_1$ has an operation module called $\mathcal{M}_1$ encapsulated into an operator $(\!|.|\!)^o$ or $(\!|.|\!)^m$.*
2. *$Solver_2$ has an open channel called Ch_2 receiving the same type of information sent by $\mathcal{M}_1$.*

4 An POSL solver

In this section we explain the structure of a solver created by POSL through an example. We choose one of the more classic solution methods for combinatorial problems: local search meta-heuristics algorithms. These algorithms have a common structure: they start by initializing some data structures (e.g. a *tabu list* for *Tabu Search* [7], a *temperature* for *Simulated Annealing* [10], etc.). Then, an initial configuration s is generated (either randomly or by using an heuristic). After that, a new configuration s^* is selected from the neighborhood $\mathcal{V}(s)$. If s^* is a solution for the problem $\mathcal{P}$, then the process stops, and s^* is returned. If not, the data structured are updated, and s^* is accepted or not for the next iteration, depending on some criterion (e.g. penalizing features of local optimums, like in *Guided Local Search* [1]).

Restarts are classic mechanisms to avoid becoming trapped in local minimum. They are trigged by reaching no improvements or a timeout.

The *operation module*s composing *local search meta–heuristics* are described bellow:

$\boxed{Operation\ Module - 1:}$ Generating a configuration s

$\boxed{Operation\ Module - 2:}$ Defining the neighborhood $\mathcal{V}(S)$

$\boxed{Operation\ Module - 3:}$ Selecting $s^* \in \mathcal{V}(s)$

$\boxed{Operation\ Module - 4:}$ Evaluating an acceptance criteria for s^*

We can combine modules to create more complex ones. For example, if we want a *operation module* generating a random configuration but able to give sometimes a fixed configuration (one passed by parameter) following a given probability, our *operation module* 1 would be the combination of this two modules through the operator ($\bigcirc\!\!\!\!\rho$).

Let's make this solver a little bit more complex: suppose that we have a neighborhood function which is exploitation-oriented, but sometimes we will ask other solvers to send their neighborhood *operation module*s to be executed in the current solver, as a mechanism to scape from local minimum. This behavior is modeled by the following *open channel*:

$\boxed{Open\ Channel - 1:}$ Asking for $\mathcal{V}(s)$ *operation module*.

Let's suppose also that we would like to always broadcast our current solution. It is possible by applying the operator $(\!|.|\!)^o$ to the *operation module* 4.

Figure 4 presents the example above. The *Solver–1*'s *open channel* is represented by a red shapes. The *Solver–1* has an *open channel* **asking for** additional neighborhood function, so it means that it can be connected with *Solver–2*, because *Solver–2* has applied the operator $(\!|.|\!)^o$ to **send** (somehow/at some point) its neighborhood function *operation module*. In the solver definition layer it is created a *subscription* (represented by a green block) to define the *communication channel*. As we can see in the diagram, there is nobody attending to the *operation module* sending the best configuration. In that case, there are no links outside the channel, and no information will be sent.

Algorithm 3 shows POSL code for the *computation strategy* of the solver described above (*Solver–1*), using predefined operators. POSL provides information regarding the execution process, such as number of Iterations, solver execution Time, Best found solution, among others.

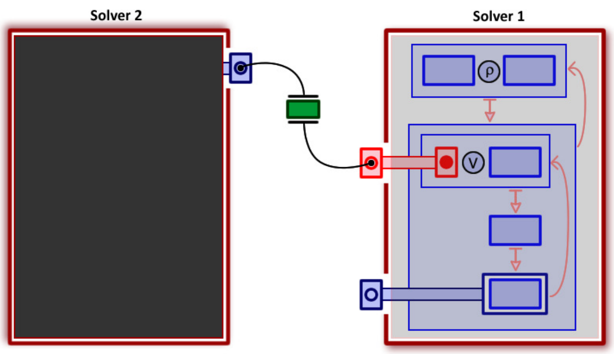


Figure 4: Diagram of a solver cooperation strategy

Algorithm 3: Local search meta–heuristic general strategy

```
$St \leftarrow$ **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) {
        $\left[Ch_1 \textcircled{\vee} \mathcal{M}_2\right] \longmapsto \mathcal{M}_3 \longmapsto (\!|\mathcal{M}_4|\!)^o$
      }]
    }]
  }
```

Algorithm 4: Local search meta–heuristic solver definition

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

Algorithm 4 shows the solver definition. We place here the *computation strategy*, followed by instances of the *module*s (*operation module*s and *open channel*s). The instances have to respect the type of the *module*s declared in the *computation strategy*.

Algorithm 5: Inter–solvers communication definition

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

Supposing that there exist another solver Σ_2 with an *operation module* called $\mathcal{M}_{\mathcal{V}}$ sharing its neighborhood function, we can connect it with the solver Σ_1 as the Algorithm 5 shows.

5 Conclusions

In this paper we have presented POSL, a framework for building cooperating solvers. It provides an effective way to build solvers ables to exchange any kind of information, even other solver's behavior, sharing their *operation module*s. Using POSL, many different solvers can be created and ran in parallel, using only one generic strategy, but instantiating different *operation module*s and *open channel*s for each other.

It is possible to implement different communication strategies, since POSL provides a layer to define *communication channels* connecting solvers dynamically using *subscription*s.

At this point, the implementation of POSL remains in progress, in which our principal task is creating a design as general as possible, allowing to add new features in the near future. Our goal is obtaining a rich library of *operation module*s and *open channel*s to be used by the user, based on a deep study of the classical meta-heuristics algorithms for solving combinatorial problems, in order to cover them as much as possible. In such a way, building new algorithms by using POSL will be easier.

At the same time we pretend to develop new operators, depending on the new needs and requirements. It is necessary, for example, to improve the *solver definition* language, allowing the process to build sets of many new solvers to be faster and easier. Furthermore, we are aiming to expand the communication definition language, in order to create versatile and more complex communication strategies, useful to study the solvers behavior.

As a medium term future work, it would be interesting also to include learning techniques, allowing the solver to change automatically, depending for instance on the results of their neighbor solvers.

References

[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.