POSL: A Parallel-Oriented Solver Language

Thesis for the Degree of
Doctor of Computer Science

Alejandro Reyes Amaro

Doctoral School STIM

Academic advisors:
Eric Monfroy[1], Florian Richoux[2]

[1]Department of Informatics
Faculty of Science
University of Nantes
France

[2]Department of Informatics
Faculty of Science
University of Nantes
France

Submitted: dd/mm/2016

Assessment committee:

Prof. (1)
Institution (1)

Prof. (2)
Institution (2)

Prof. (3)
Institution (3)

ISBN ??

Part I

Presentation

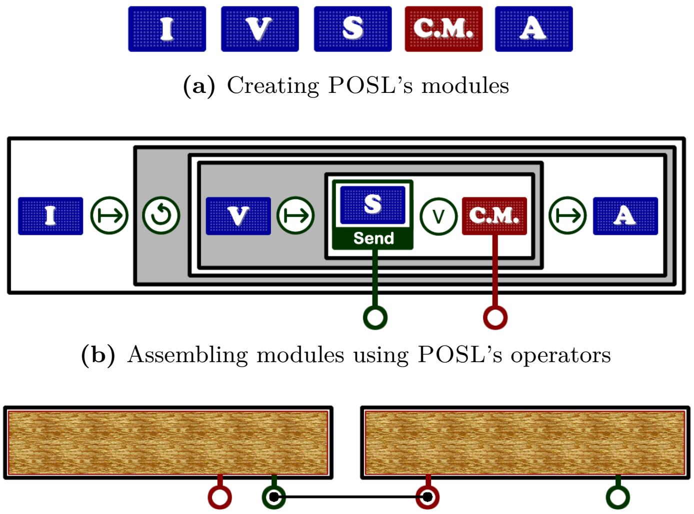
1

A Parallel-Oriented Language for Modeling Meta-Heuristic-Based Solvers

In this chapter POSL *is introduced as the main contribution, and a new way to solve CSPs. Its characteristics and advantages are summarized, and a general procedure to be followed is described, in order to build parallel solvers using* POSL, *followed by a detailed description of each of the single steps.*

Contents

In this chapter we present the different steps to build communicating parallel solvers with POSL. First of all, the algorithm we have conceived to solve the target problem is decomposed it into small modules of computation, which are implemented as separated *functions*. We name them *computation modules* (see Figure 1.1a, blue shapes). At this point it is crucial to find a good decomposition of its algorithm, because it will have a significant impact in its future re-usage and variability. The next step is to decide which information is interesting to *receive* from other solvers. This information is encapsulated into another kind of component called *communication module*, allowing data transmission between solvers (see Figure 1.1a, red shapes). A third stage is to ensemble the modules through POSL's inner language (the interested reader is referred to Appendix [...]) to create independent solvers. The parallel-oriented language based on operators provided by POSL (see Figure 1.1b, green shapes) allows not only the information exchange, but also executing components in parallel. In this stage the information that is interesting to be shared with other solvers is sent using operators. After that we can connect them using *communication operators*. We call this final entity a *solvers set* (see Figure 1.1c).



(a) Creating POSL's modules

(b) Assembling modules using POSL's operators

(c) Connecting POSL solvers to create *communication strategies*

Figure 1.1: Solver construction process using POSL

In the following sections we explain in details all these steps, but first, we explain how to model the target benchmark using POSL.

1.1 Modeling the target benchmark

Target problems are modeled in POSL using the C++ programing language, respecting some rules of the object-oriented design. First of all, the benchmark must inherit from the class **Benchmark** provided by POSL. This class does not have any method to be overridden or implemented, but receives in its constructor three objects, instances from classes that the user must create. Those classes must inherit from **SolutionCostStrategy**, **RelativeCostStrategy** and **ShowStrategy**, respectively. In these classes the most important functionalities of the benchmark model are defined.

SolutionCostStrategy: In this class the strategy to compute the *cost* of a configuration is implemented. POSL is based on improving step by step an initial configuration, taking into account a *cost function* provided by the user through the model (by implementing the function *solutionCost(dots)*). The kind of problems that POSL solves is the class of *Constraint Satisfaction Problems*, so this *cost function* must return an integer taking into account the problem constraints. Given a configuration s, the *cost function*, as a mandatory rule, must return 0 if and only if s is a solution of the problem, i.e., s fulfill all the problem constraints. An example of *cost function* is one that returns the number of violated constraints. However, the more expressive the function cost is, the better the performance of POSL leading to the solution.

The method to be implemented in this class is:

- `int solutionCost(std::vector<int> & c)` → Computes the cost of a given configuration (`c`).

RelativeCostStrategy: In this class the user implements the strategy to compute the *cost* of a given configuration with respect to another. If the cost of some configuration has been calculated, sometimes it is possible to store some information in order to compute the cost of another configuration, if the differences between them are known. If it is possible, the algorithms is defined in this class. If it is not possible, this class must have the same functionality of **SolutionCostStrategy**.

The methods to implement in this class are:

- `void initializeCostData(std::vector<int> & c)` → Initializes the information related to the cost (auxiliary data structures, the current configuration (`c`), the current cost, etc.)

- `void updateConfiguration(std::vector<int> & c)` → Updates the information related to the cost.

- `int relativeSolutionCost(std::vector<int> & c)` → Returns the relative cost of the configuration `c` with respect to the current configuration.

- `int currentCost()` → Property that returns the cost of the current configuration.

- `int costOnVariable(int variable_index)` → Returns a measure of the contribution of a variable to the total cost of a configuration.

- `int sickestVariable()` → Returns the variable contributing the most to the cost.

SolutionCostStrategy: This class represents the way a benchmark shows a configuration, in order to provide more information about the structure. For example, a configuration of the instance 3–3–2 of the *Social Golfers Problem* (see bellow for more details about this benchmark) can be written as follows:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 3, 4, 5, 6, 7, 8, 9, 1, 2]
```

This text is, nevertheless, very difficult to be read if the instance is larger. Therefore, it is recommended that the user implements this class in order to give more details and to make it easier to interpret the configuration. For example, for the same instance of the problem, a solution could be presented as follows:

```
Golfers: players-3, groups-3, weeks-2
6       8       7
1       3       5
4       9       2
--
7       2       3
4       8       1
5       6       9
--
```

The method to be implemented in this class is:

- `std::string showSolution(std::shared_ptr<Solution> s)` → Returns a string to be written in the standard output.

Once we have modeled the target benchmark, it can be solved using POSL. In the following sections we describe how to use this parallel-oriented language to solve *Constraint Satisfaction Problems*.

1.2 First stage: creating POSL's modules

There exist two types of basic modules in POSL: *computation modules* and *communication modules*. A *computation module* is a function which received an input, then executes an internal algorithm, and returns an output. A *communication module* is also a function receiving and returning information, but in contrast, the *communication module* can receive

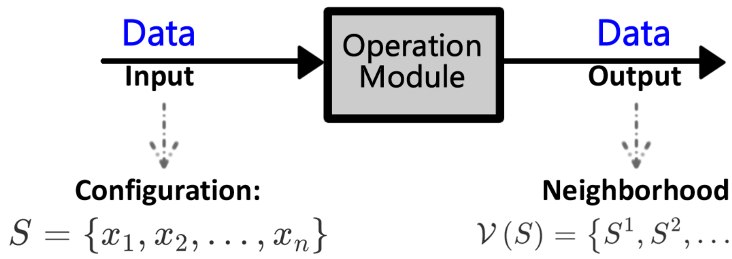


Figure 1.2: An example of a computation module computing a neighborhood

information from two different sources: through input parameters or from outside, i.e., by communicating with a module from another solver.

1.2.1 Computation Module

A *computation module* is the most basic and abstract way to define a piece of computation. It is a function which receives an instance of a POSL data type as input, then executes an internal algorithm, and returns an instance of a POSL data type as output. The input and output types will characterize the computation module signature. It can be dynamically replaced by (or combined with) other computation modules, since they can be shared among solvers working in parallel. They are joined through *abstract solvers*.

Definition 1 ***(Computation Module)*** *A computation module Cm is a mapping defined by:*

$$Cm : D \rightarrow I \tag{1.1}$$

where D and I can be either a set of configurations, a set of sets of configurations, a set of values of some data type, etc.

Consider a local search meta-heuristic solver. One of its *computation modules* can be the function returning the set of configurations composing the neighborhood of a given configuration:

$$Cm_{neighborhood} : D_1 \times D_2 \times \cdots \times D_n \rightarrow 2^{D_1 \times D_2 \times \cdots \times D_n}$$

where D_i represents the definition domains of each variable of the input configuration.

Figure 1.2 shows an example of *computation module*: which receives a configuration S and then computes the set $\mathcal{V}$ of its neighbor configurations $\{S^1, S^2, \ldots, S^m\}$.

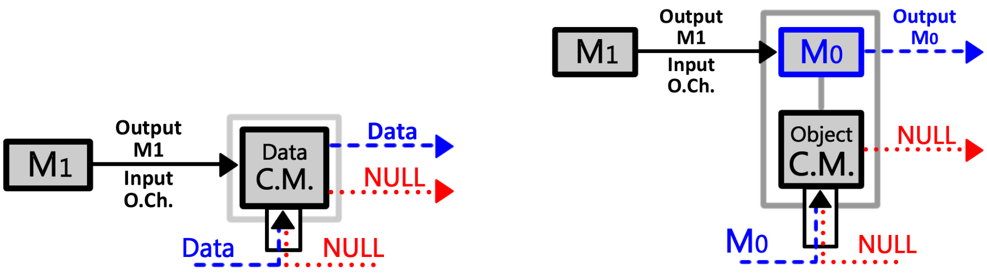
1.2.1.1 Creating new *computation modules*

To create new *computation modules* we use C++ programing language. POSL provides a hierarchy of data types to work with (See anexes) and some abstract classes to inherit from, depending on the type of *computation module* that the user wants to create. These abstract classes represent *abstract computation module* and define a type of action to be executed. In the following we present the most important ones:

- **AOM_FirstConfigurationGeneration** → Represents *computation modules* generating a first configuration. The user must implement the method *spcf_execute(**ComputationData**)* which returns a pointer to a **Solution**, that is, an object containing all the information concerning a partial solution (configuration, variable domains, etc.)
- **AOM_NeighborhoodFunction** → Represent *computation modules* creating a neighborhood of a given configuration. The user must implement the method *spcf_execute(**Solution**)* which returns a pointer to an object **Neighborhood**, containing a set of configurations which constitute the neighborhood of a given configuration, according to certain criteria. These configuration are efficiently stored.
- **AOM_SelectionFunction** → Represents *computation modules* selecting a configuration from a neighborhood. The user must implement the method *spcf_execute(**Neighborhood**)* which returns a pointer to an object **DecisionPair**, containing two solutions: the current and the selected one.
- **AOM_DecisionFunction** → Represents *computation modules* deciding which of the two solutions will be the current configuration for the next iteration. The user must implement the method *spcf_execute(**DecisionPair**)* which returns a pointer to an object **Solution**.

1.2.2 Communication modules

A *communication module* is also a function receiving and returning information, but in contrast, the *communication module* can also receive information by communicating with a module from another solver. A *communication module* is the component managing the information reception in the communication between solvers (we will talk about information transmission in the next section). They can interact with *computation modules* through operators (see Figure 1.3).



(a) Data *communication module* **(b)** Object *communication module*

Figure 1.3: Communication module

A *communication module* can receive two types of information from an external solver: data or *computation modules*. It is important to notice that by sending/receiving *computation modules*, we mean sending/receiving only required information to identify and being able to instantiate the *computation module*.

In order to distinguish from the two types of *communication modules*, we will call Data Communication Module to the *communication module* responsible for the data reception (Figure 1.3a), and Object Communication Module to the one responsible for the reception and instantiation of *computation modules* (Figure 1.3b).

Definition 2 ***(Data Communication Module)*** *A* Data Communication Module *Ch is a component that produces a mapping defined as follows:*

$$Ch : U \to I \tag{1.2}$$

It returns the information I coming from an external solver, no matter what the input U is.

Definition 3 ***(Object Communication Module)*** *If we denote by $\mathbb{M}$ the space of all the computation modules defined by Definition 1.1, then an* Object Communication Module *Ch is a component that produces a computation module coming from an external solver as follows:*

$$Ch : \mathbb{M} \to \mathbb{M} \tag{1.3}$$

Users can implement new computation and connection modules but POSL already contains many useful modules for solving a broad range of problems.

Due to the fact that *communication modules* receive information coming from outside without having control on them, it is necessary to define the *NULL* information, in order to denote the absence of information. If a Data Communication Module receives a piece of information, is returned automatically. If a Object Communication Module receives a *computation module*, it is instantiated and executed with the *communication module*'s input and its result is returned. In both cases, if no available information exists (no communications performed), the *communication module* returns the *NULL* object.

1.3 Second stage: assembling POSL's modules

Modules mentioned above are defined respecting the signature of some predefined abstract module. For example, the module showed in Figure 1.2 is a *computation module* based on an abstract module that receives a configuration and returns a neighborhood. In that sense, an example of a concrete *computation module* (or just *computation module*) can be a function receiving a configuration, and returning a neighborhood constituted by N configurations which only differ from the input configuration in one entry.

In this stage an *abstract solver* is coded using POSL. It takes abstract modules as *parameters* and combines them through operators. Through the *abstract solver*, we can also decide which information to send to other solvers by using some operators to send the result of a computation module (see below). In the following we present a formal and more detailed specification of POSL's operators.

The *abstract solver* is the solver's backbone. It joins the *computation modules* and the *communication modules* coherently. It is independent from the *computation modules* and *communication modules* used in the solver. It means that they can be changed or modified during the execution, without altering the general algorithm, but still respecting the main structure. Each time we combine some of them using POSL's operators, we are creating a *compound module*. Here we formally define the concept of *module* and *compound module*.

Definition 4 *A* **module** *is (and it is denoted by the letter $\mathcal{M}$):*

a) a computation module or

b) a communication module or

c) $[\mathcal{M}_1 \; OP \; \mathcal{M}_2]$, which is the composition of two modules $\mathcal{M}_1$ and $\mathcal{M}_2$ to be executed sequentially, returning an output depending on the nature of the operator OP*; or*

d) $[\![\mathcal{M}_1 \; OP \; \mathcal{M}_2]\!]$, which is the composition of two modules $\mathcal{M}_1$ and $\mathcal{M}_2$ to be executed, 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 $\mathbb{M}$ *and call compound modules to the composition of modules described in c) and d).*

For a better understanding of Definition 4, Figure 1.4 shows graphically the structure of a *compound module.*

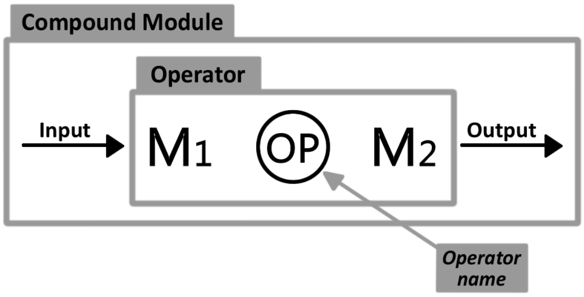


Figure 1.4: A *compound module*

As mentioned before, the *abstract solver* is independent from the *computation modules* and *communication modules* used in the solver. It means that one *abstract solver* can be used to construct many different solvers, by implementing it using different modules (see below the related concept of *abstract solver* instantiation). This is the reason why the *abstract solver* is defined only using abstract modules. Formally, we define an *abstract solver* as follows:

Definition 5 ***(Abstract Solver)*** *An* Abstract Solver *AS is a triple* $(\mathbf{M}, \mathcal{L}^m, \mathcal{L}^c)$*, where:* $\mathbf{M}$ *is a compound module (also called* root *compound module),* $\mathcal{L}^m$ *a list of abstract computation modules appearing in* $\mathcal{M}$*, and* $\mathcal{L}^c$ *a list of communication modules appearing in* $\mathcal{M}$*.*

The *root compound module* can be defined also as a free-context grammar as follows:

Definition 6 (root *compound module***'s grammar)** $G_{POSL} = (\mathbf{V}, \Sigma, \mathbf{S}, \mathbf{R})$*, where:*

a) $\mathbf{V} = \{CM, OP\}$ *is the set of variables,*

b) $\Sigma = \left\{\alpha, \beta, be, [,], [\![,]\!]_p, (,), \{,\}, (\!|, |\!)^m, |\!)^o, \bigcirc\!\!\!\!\mapsto, \bigcirc\!\!\!\!?, \circlearrowleft, \bigcirc\!\!\!\!\rho, \bigcirc\!\!\!\!\vee, \bigcirc\!\!\!\!\wedge, \bigcirc\!\!\!\!\!M, \bigcirc\!\!\!\!\!m, \bigcirc\!\!\!\!\downarrow, \bigcirc\!\!\!\!\cup, \bigcirc\!\!\!\!\cap\right\}$ *is the set of terminals,*

c) $\mathbf{S} = \{CM\}$ *is the set of start variables,*

d) and $\mathbf{R} =$

$$CM \longmapsto \alpha \mid \beta \mid (\!|CM|\!)^o \mid (\!|CM|\!)^m \mid [OP] \mid [\![OP]\!]_p$$

$$OP \longmapsto CM \bigcirc\!\!\!\!\mapsto CM \mid CM \bigcirc\!\!\!\!? CM \mid CM \bigcirc\!\!\!\!\rho CM \mid CM \bigcirc\!\!\!\!\vee CM \mid CM \bigcirc\!\!\!\!\wedge CM$$

$$OP \longmapsto CM \bigcirc\!\!\!\!\!M CM \mid CM \bigcirc\!\!\!\!\!m CM \mid CM \bigcirc\!\!\!\!\downarrow CM \mid CM \bigcirc\!\!\!\!\cup CM \mid CM \bigcirc\!\!\!\!\cap CM$$

$$OP \longmapsto CM \circlearrowleft (be) \{CM\}$$

is a set of rules

In the following I explain some of the concepts in Definition 6:

- The variables CM and OP are two very important entities in the language, as it can be seen in the grammar. We name them *compound module* and *operator*, respectively.
- The terminals α and β represent a *computation module* and a *communication module*, respectively.
- The terminal *be* is a boolean expression.
- The terminals $[\ ], [\![\ ]\!]_p$ are symbols for grouping and defining the way the involved *compound modules*are executed. Depending on the nature of the operator, this can be either sequentially or in parallel:
  a) [OP]: The involved operator is executed sequentially.
  b) $[\![\text{OP}]\!]_p$: The involved operator is executed in parallel if and only if *OP* supports parallelism. Otherwise, an exception is thrown.
- The terminals (and) are symbols for grouping the boolean expression in some operators.
- The terminals { and } are symbols for grouping *compound modules* in some operators.
- The terminals $(\!|.|\!)^m$, $(\!|.|\!)^o$, are operators to send information to other solvers (explained bellow).
- The rest of terminals are POSL operators.

In the following we define POSL operators. In order to group modules, like in Definition 4(*c)*) and 4(*d)*), we will use $|.|$ as generic grouper.

Definition 7 (Operator Sequential Execution) *Let*

a) $\mathcal{M}_1 : \mathcal{D}_1 \to \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \to \mathcal{I}_2$*,*

be modules, where $\mathcal{I}_1 \subseteq \mathcal{D}_2$*. Then the operation* $\left|\mathcal{M}_1 \circledcirc\!\!\!\mapsto \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{seq}$ *as the result of executing* $\mathcal{M}_1$ *followed by executing* $\mathcal{M}_2$*:*

$$\mathcal{M}_{seq} : \mathcal{D}_1 \to \mathcal{I}_2$$

This is an example of an operator that does not support the execution of its involved *compound modules* in parallel, because the input of the second *compound module* is the output of the first one.

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

Definition 8 (Operator Conditional Execution) *Let*

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$*,*

be modules, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$*. Then the operation* $\left|\mathcal{M}_1 ?_{<cond>} \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{cond}$ *as result of the sequential execution of* $\mathcal{M}_1$ *if* $< cond >$ *is* **true** *or* $\mathcal{M}_2$*, otherwise:*

$$\mathcal{M}_{cond} : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

We can execute modules sequentially creating also cycles.

Definition 9 (Operator Cyclic Execution) *Let* $\mathcal{M} : \mathcal{D} \rightarrow \mathcal{I}$ *be a module, where* $\mathcal{I} \subseteq \mathcal{D}$*. Then, the operation* $|\circlearrowleft_{<cond>} \mathcal{M}|$ *defines the compound module* $\mathcal{M}_{cyc}$ *as result of the sequential execution of* $\mathcal{M}$ *repeated while* $< cond >$ *remains* **true***:*

$$\mathcal{M}_{cyc} : \mathcal{D} \rightarrow \mathcal{I}$$

Definition 10 (Operator Random Choice) *Let*

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$*,*

be modules, where $\mathcal{D}_1 \subset \mathcal{D}_2$ *and a real value* ρ*. Then the operation* $\left|M_1 \rho \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{rho}$ *that executes and returns the output of* $\mathcal{M}_1$ *with probability* ρ*, or executes and returns the output of* $\mathcal{M}_2$ *with probability* $(1-\rho)$*:*

$$\mathcal{M}_{rho} : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

The following operator is very useful if the user needs to use a *communication module* inside an *abstract solver*. As explained before, if a *communication module* does not receive any information from another solver, it returns *NULL*. This may cause the undesired termination of the solver if this case is not considered correctly. Next, I introduce the operator **Operator Not** *NULL* **Execution** and illustrate how to use it in practice with an example.

Definition 11 (Operator Not *NULL* **Execution)** *Let*

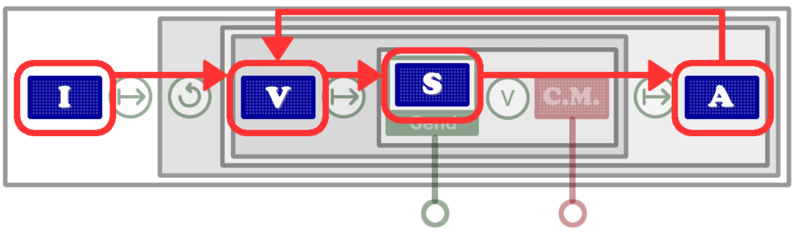
a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$,

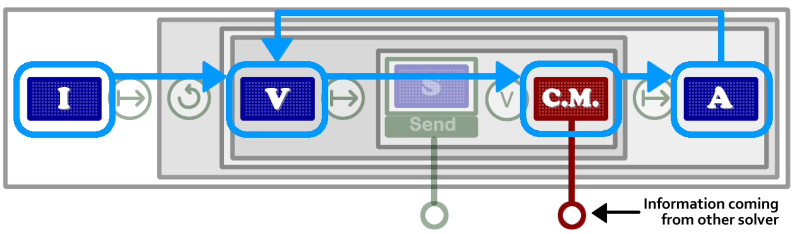
be modules, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$. *Then, the operation* $\left|\mathcal{M}_1 \circledvee \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{non}$ *that executes* $\mathcal{M}_1$ *and returns its output if it is not NULL, or executes* $\mathcal{M}_2$ *and returns its output otherwise:*

$$\mathcal{M}_{non} : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

Figure 1.5 shows how to combine a connection module with the *computation module* S through the operator $\circledvee$. Here, the *computation module* S will be executed as long as the *communication module* remains *NULL*, i.e., there is no information coming from outside. This behavior is represented in Figure 1.5a by the orange lines. If some data has been received through the *communication module*, the later is executed instead of the module S, represented in Figure 1.5b by blue lines.



(a) The solver executes the computation module **S** if no information is received through the connection module



(b) The solver uses the information coming from an external solver

Figure 1.5: Two different behaviors within the same solver

This is *short-circuit* operator. It means that if the first argument (module) does not return *NULL*, the second will not be executed. POSL provides another operator with the same functionality but not *short-circuit*:

Definition 12 (Operator *BOTH* Execution) *Let*

a) $\mathcal{M}_1 : \mathcal{D}_1 \to \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \to \mathcal{I}_2$*,*

be modules, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$*. Then the operation* $\left|\mathcal{M}_1 \wedge \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{both}$ *that executes both* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, then returns the output of* $\mathcal{M}_1$ *if it is not NULL, or the output of* $\mathcal{M}_2$ *otherwise:*

$$\mathcal{M}_{both} : \mathcal{D}_1 \cap \mathcal{D}_2 \to \mathcal{I}_1 \cup \mathcal{I}_2$$

In the following definitions, 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 13 and 14). On the other hand, competitive parallelism arise when the general result is the result of the process ending first (e.g. Definition 15).

Definition 13 (Operator Minimum) *Let*

a) $\mathcal{M}_1 : \mathcal{D}_1 \to \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \to \mathcal{I}_2$*,*

be modules, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$*. Let also* o_1 *and* o_2 *be the outputs of* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, respectively. Assume that there exists some order criteria between them. Then the operation* $\left|\mathcal{M}_1 m \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{min}$ *that executes* $\mathcal{M}_1$ *and returns* $\min\{o_1, o_2\}$*:*

$$\mathcal{M}_{min} : \mathcal{D}_1 \cap \mathcal{D}_2 \to \mathcal{I}_1 \cup \mathcal{I}_2$$

Similarly we define the operator **Maximum**:

Definition 14 (Operator Maximum) *Let*

a) $\mathcal{M}_1 : \mathcal{D}_1 \to \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \to \mathcal{I}_2$*,* *be modules, where* $\mathcal{D}_1 \subseteq \mathcal{D}_2$*. Let also* o_1 *and* o_2 *be the outputs of* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, respectively. Assume that there exists some order criteria between them. Then the operation* $\left|\mathcal{M}_1 \textcircled{M} \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{max}$ *that executes* $\mathcal{M}_1$ *and returns* $\max\{o_1, o_2\}$*:*

$$\mathcal{M}_{max} : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

Definition 15 (Operator Race) *Let*

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$*,*

be modules, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$ *and* $\mathcal{I}_1 \subset \mathcal{I}_2$*. Then the operation* $\left|\mathcal{M}_1 \textcircled{\downarrow} \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{race}$ *that executes both modules* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, and returns the output of the module ending first:*

$$\mathcal{M}_{race} : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

The operators presented in Definitions 13, 14 and 15 are very useful in terms of sharing not only information between solvers, but also *behaviors*. If one of the operands is a *communication module* it can receive an external solver's *computation module*, providing the opportunity to instantiate it in the current solver. The operator either instantiates that module if it is not null, and executes it; or executes the other operand module otherwise.

Some others operators can be useful when dealing with *sets*.

Definition 16 (Operator Union) *Let*

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$*,*

be modules, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$*. Let also* V_1 *and* V_2 *be the outputs of* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, respectively. Then the operation* $\left|\mathcal{M}_1 \textcircled{\cup} \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{\cup}$ *that executes both modules* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, and returns* $V_1 \cup V_2$*:*

$$\mathcal{M}_{\cup} : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

Similarly we define the operators **Intersection** and **Subtraction**:

Definition 17 (Operator Intersection) *Let* *a)* $\mathcal{M}_1 : \mathcal{D}_1 \to \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \to \mathcal{I}_2$*,*

be modules, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$*. Let also* V_1 *and* V_2 *be the outputs of* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, respectively. Then the operation* $\left|\mathcal{M}_1 \bigcirc\!\!\!\!\cap\, \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_\cap$ *that executes both modules* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, and returns* $V_1 \cap V_2$*:*

$$\mathcal{M}_\cap : \mathcal{D}_1 \cap \mathcal{D}_2 \to \mathcal{I}_1 \cup \mathcal{I}_2$$

Definition 18 (Operator Subtraction) *Let*

a) $\mathcal{M}_1 : \mathcal{D}_1 \to \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \to \mathcal{I}_2$*,*

be modules, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$*. Let also* V_1 *and* V_2 *be the outputs of* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, respectively. Then the operation* $\left|\mathcal{M}_1 \ominus \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_-$ *that executes both modules* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, and returns* $V_1 - V_2$*:*

$$\mathcal{M}_- : \mathcal{D}_1 \cap \mathcal{D}_2 \to \mathcal{I}_1 \cup \mathcal{I}_2$$

Now, I define the operators which allows to send information to other solvers. Two types of information can be sent: i) the output of the *computation module* and send its output, or ii) the *computation module* itself. . This utility is very useful in terms of sharing behaviors between solvers.

Definition 19 (Sending Data Operator) *Let* $\mathcal{M} : \mathcal{D} \to \mathcal{I}$ *be a module. Then the operation* $|(\!|\mathcal{M}|\!)^o|$ *defines the compound module* $\mathcal{M}_{sendD}$ *that executes the module* $\mathcal{M}$ *and sends its output outside:*

$$\mathcal{M}_{sendD} : \mathcal{D} \to \mathcal{I}$$

Similarly we define the operator **Send Module**:

Definition 20 (Sending Module Operator) *Let* $\mathcal{M} : \mathcal{D} \to \mathcal{I}$ *be a module. Then the operation* $|(\!|\mathcal{M}|\!)^m|$ *defines the compound module* $\mathcal{M}_{sendM}$ *that executes the module* $\mathcal{M}$*, then returns its output and sends the module itself outside:*

$$\mathcal{M}_{sendM} : \mathcal{D} \to \mathcal{I}$$

Once all desired abstract modules are linked together with operators, we obtain the *root compound module*, an important part of an *abstract solver*. To implement a concrete solver from an *abstract solver*, one must instantiate each abstract module with a concrete one respecting the required signature. From the same *abstract solver*, one can implement many different concrete solvers simply by instantiating abstract modules with different concrete modules.

An *abstract solver* is defined as follows: after declaring the **abstract solver**'s name, the first line defines the list of abstract *computation modules*, the second one the list of abstract *communication modules*, then the algorithm of the solver is defined as the solver's body (the root *compound module*), between **begin** and **end**.

An *abstract solver* in the following simple regular expression:

$$\textbf{abstract solver}\ \textit{name}\ \textbf{computation}:\ L^m\ (\textbf{communication}:\ L^c)?\ \textbf{begin}\ \mathcal{M}\ \textbf{end}$$

where:

- *name* is the identifier of the *abstract solver*,
- L^m is the list of abstract *computation modules*,
- L^c is the list of abstract *communication modules*, and
- $\mathcal{M}$ is the root *compound module*.

For instance, Algorithm 1 illustrates the abstract solver corresponding to Figure 1.1b.

Algorithm 1: POSL pseudo-code for the *abstract solver* presented in Figure 1.1b

abstract solver *as_01*
computation *:* I, V, S, A
connection*:* $C.M.$
  begin
    $I \ \bigcirc\!\!\!\!\mapsto$
    $[\circlearrowleft$ ($\text{I}\scriptstyle{\text{TR}}$ $\%\ K_2$) **begin**
      $V \ \bigcirc\!\!\!\!\mapsto \ \left[C.M. \ \bigcirc\!\!\!\!\vee \ (\!|S|\!)^o\right] \ \bigcirc\!\!\!\!\mapsto \ A$
    end]
  end

1.4 Third stage: creating POSL solvers

With *computation* and *communication modules* composing an *abstract solver*, one can create solvers by instantiating *modules*. This is simply done by specifying that a given **solver** must **implements** a given *abstract solver*, followed by the list of *computation* then *communication modules*. These modules must match signatures required by the *abstract solver*. Algorithm 2 implements Algorithm 1 by instantiating modules shown in Figure 1.5.

Algorithm 2: An instantiation of the *abstract solver* presented in Algorithm 1

solver solver_01 **implements** as_01
computation : $I_{rand}, V_{std}, S_{best}, A_{alw}$
connection: CM_{last}

Algorithm 2 is just an example of a solver instantiation, using some *computation modules* provided by POSL, that are used and explained in details in the Chapter **??** of this document:

- I_{rand} creates a random configuration.
- V_{std} creates a neighborhood of a given configuration, changing one element at a time.
- S_{best} selects the configuration of a neighborhood with the lowest cost.
- A_{alw} always accepts the incoming configuration.
- CM_{last} returns the last configuration arrived, if at the time of its execution, there is more than one configuration waiting to be received.

1.5 Forth stage: connecting the solvers

We call *solver set* to the pool of (concrete) solvers that we plan to use in parallel to solve a problem. Once we have our solvers set, the last stage is to connect the solvers to each other. Up to this point, solvers are disconnected, but they are ready to establish the communication. POSL provides a platform to the user such that cooperative strategies can be easily defined.

In the following we present two important concepts necessary to formalize the *communication operators*.

Definition 21 (Communication Jack) *Let $\mathcal{S}$ be a solver. Then the operation $\mathcal{S} \cdot \mathcal{M}$ opens an outgoing connection from the solver $\mathcal{S}$, sending to the outside either a) the output of $\mathcal{M}$, if it is affected by a sending data operator as presented in Definition 19, or b) $\mathcal{M}$ itself, if it is affected by a sending module operator as presented in Definition 20.*

Definition 22 (Communication Outlet) *Let $\mathcal{S}$ be a solver. Then, the operation $\mathcal{S} \cdot \mathcal{CM}$ opens an ingoing connection to the solver $\mathcal{S}$, receiving from the outside either a) the output of some computation module, if $\mathcal{CM}$ is a data communication module, or b) a computation module, if $\mathcal{CM}$ is an object communication module.*

The communication is established by following the following rules guideline:

a) Each time a solver sends any kind of information by using a *sending* operator, it creates a *communication jack*.

b) Each time a solver defines a *communication module*, it creates a *communication outlet*.

c) Solvers can be connected to each other by linking *communication jacks* to *communication outlets*.

Following, we define the *connection operators* that POSL provides.

Definition 23 (Connection Operator One-to-One) *Let*

a) $\mathcal{J} = [\mathcal{S}_0 \cdot \mathcal{M}_0, \mathcal{S}_1 \cdot \mathcal{M}_1, \ldots, \mathcal{S}_{N-1} \cdot \mathcal{M}_{N-1}]$ *be the list of communication jacks, and*

b) $\mathcal{O} = [\mathcal{Z}_0 \cdot \mathcal{CM}_0, \mathcal{Z}_1 \cdot \mathcal{CM}_1, \ldots, \mathcal{Z}_{N-1} \cdot \mathcal{CM}_{N-1}]$ *be the list of communication outlets*

Then the operation

$$\mathcal{J} \; \bigcirc\!\!\!\!\!\rightarrow \; \mathcal{O}$$

connects each communication jack $\mathcal{S}_i \cdot \mathcal{M}_i \in \mathcal{J}$ with the corresponding communication outlet $\mathcal{Z}_i \cdot \mathcal{CM}_i \in \mathcal{O}$, $\forall 0 \leq i \leq N-1$ (see Figure 1.6a).

Definition 24 (Connection Operator One-to-N) *Let*

a) $\mathcal{J} = [\mathcal{S}_0 \cdot \mathcal{M}_0, \mathcal{S}_1 \cdot \mathcal{M}_1, \ldots, \mathcal{S}_{N-1} \cdot \mathcal{M}_{N-1}]$ *be the list of communication jacks, and*

b) $\mathcal{O} = [\mathcal{Z}_0 \cdot \mathcal{CM}_0, \mathcal{Z}_1 \cdot \mathcal{CM}_1, \ldots, \mathcal{Z}_{M-1} \cdot \mathcal{CM}_{M-1}]$ *be the list of communication outlets*

Then the operation

$$\mathcal{J} \; \bigcirc\!\!\!\!\!\rightsquigarrow \; \mathcal{O}$$

connects each communication jack $\mathcal{S}_i \cdot \mathcal{M}_i \in \mathcal{J}$ with every communication outlet $\mathcal{Z}_j \cdot \mathcal{CM}_j \in \mathcal{O}$, $\forall 0 \leq i \leq N-1$ and $0 \leq j \leq M-1$ (see Figure 1.6b).

Definition 25 (Connection Operator Ring) *Let*

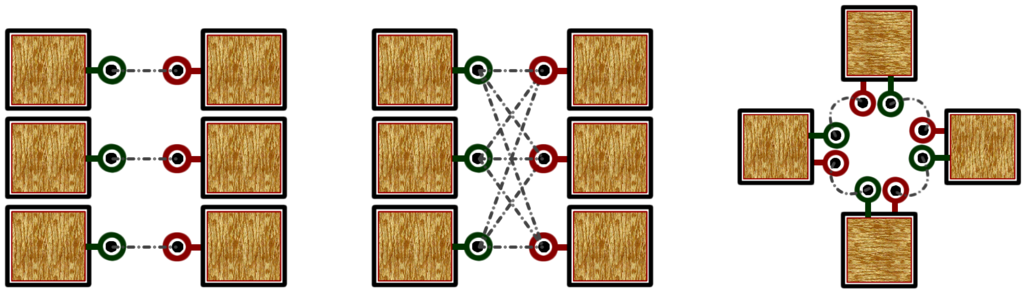
a) $\mathcal{J} = [\mathcal{S}_0 \cdot \mathcal{M}_0, \mathcal{S}_1 \cdot \mathcal{M}_1, \dots, \mathcal{S}_{N-1} \cdot \mathcal{M}_{N-1}]$ *be the list of communication jacks, and*

b) $\mathcal{O} = [\mathcal{S}_0 \cdot \mathcal{CM}_0, \mathcal{S}_1 \cdot \mathcal{CM}_1, \dots, \mathcal{S}_{N-1} \cdot \mathcal{CM}_{N-1}]$ *be the list of communication outlets*

Then the operation

$$\mathcal{J} \; \leftrightarrow \; \mathcal{O}$$

connects each communication jack $\mathcal{S}_i \cdot \mathcal{M}_i \in \mathcal{J}$ *with the corresponding communication outlet* $\mathcal{Z}_{(i+1)\%N} \cdot \mathcal{CM}_{(i+1)\%N} \in \mathcal{O}$, $\forall 0 \leq i \leq N-1$ *(see Figure 1.6c).*



(a) Communication 1 to 1 **(b)** Communication 1 to N **(c)** Cyclic communication

Figure 1.6: Graphic representation of communication operators

POSL also allows to declare non-communicating solvers to be executed in parallel, declaring only the list of solver names:

$$[\mathcal{S}_0, \mathcal{S}_1, \dots, \mathcal{S}_{N-1}]$$

When we apply a connection operator op between a *communication jacks* list $\mathcal{J}$ and a *communication outlets* list $\mathcal{O}$, internally we are assigning an *abstract computation unit* (typically a thread) to each solver that we declare in each list. This assignment receives the name of *Solver Scheduling*. Before running the *solver set*, this *abstract unit of computation* is just an integer $\tau \in [0..N]$ identifying uniquely each of the solvers. When the *solver set* is launched, the solver with the identifier τ runs into the computation unit τ. This identifier assignation remains independent of the real availability of resources of computation. It just takes into account the user declaration. This means that, if the user declares 30 solvers (15 senders and 15 receivers) and the *solver set* is launched using 20 cores, only the first 20 solvers will be executed, and in consequence, there will be 10 solvers sending information to nowhere. Users should take this into account when declaring the *solver set*.

The connection process depends on the applied conection operator. In each case the goal is to assign, to the sending operator ($(\!|.|\!)^o$ or $(\!|.|\!)^m$) inside the *abstract solver*, the identifier of the solver (or solvers, depending on the connection operator) where the information will be sent. Algorithm 3 presents the connection process.

Algorithm 3: Scheduling and connection main algorithm

input : $\mathcal{J}$ list of *communication jacks*,
$\mathcal{O}$ list of *communication outlets*

while *no available jacks or outlets* **do**
  $S_{jack} \leftarrow$ `GetNext(`$\mathcal{J}$`)`
  $R_{oulet} \leftarrow$ `GetNext(`$\mathcal{O}$`)`
  $S \leftarrow$ `GetSolverFromConnector(`S_{jack}`)`
  $R \leftarrow$ `GetSolverFromConnector(`R_{oulet}`)`
  `Schedule(`S`)`
  $R_{id} \leftarrow$ `Schedule(`R`)`
  `Connect(root(`S`),`S_{jack}, R_{id}`)`
end

In Algorithm 3:

- `GetNext(...)` returns the next available solver-jack (or solver-outlet) in the list, depending on the connection operator, e.g., for the connection operator One-to-N, each *communication jack* in $\mathcal{J}$ must be connected with each *communication outlet* in $\mathcal{O}$.
- `GetSolverFromConnector(...)` returns the solver name given a connector declaration.
- `Schedule(...)` schedules a solver and returns its identifier.
- `Root(...)` returns the *root compound module* of a solver.
- `Connect(...)` searches the *computation module* S_{jack} recursively inside the *root compound module* of S and places the identifier R_{id} into its list of destination solvers.

1.5.1 Solver name space expansion

One of the goals of POSL is to provide a way to declare sets of solvers to be executed in parallel fast and easily. For that reason, POSL provides two forms of namespace expansion, in order to create sets of solvers using already declared ones:

Solver name expansion - Uses an integer K to denote how many times the solver name S will appear in the declaration. $[\ldots S_i \cdot \mathcal{M}(K), \ldots]$ expands as $\left[\ldots S_i \cdot \mathcal{M}, S_i^2 \cdot \mathcal{M}, \ldots S_i^K \cdot \mathcal{M} \ldots\right]$ and all new solvers $S_i^j, j \in [2..K]$ are created using the same solver declaration of solver S_i.

Connection declaration expansion - Uses an integer K to denote how many times the connection will be repeated in the declaration. Let a) $[\mathcal{S}_1 \cdot \mathcal{M}_1, \ldots, \mathcal{S}_N \cdot \mathcal{M}_N]$ and

b) $[\mathcal{R}_1 \cdot \mathcal{CM}_1, \ldots, \mathcal{R}_M \cdot \mathcal{CM}_M]$ be the list of *communication jacks* and *communication outlets*, respectively, and c) op a connection operator. Then

$$[\mathcal{S}_1 \cdot \mathcal{M}_1, \ldots, \mathcal{S}_N \cdot \mathcal{M}_N] \; op \; [\mathcal{R}_1 \cdot \mathcal{CM}_1, \ldots, \mathcal{R}_M \cdot \mathcal{CM}_M] \, K$$

expands as

$$\begin{gathered}
[\mathcal{S}_1 \cdot \mathcal{M}_1, \ldots, \mathcal{S}_N \cdot \mathcal{M}_N] \; op \; [\mathcal{R}_1 \cdot \mathcal{CM}_1, \ldots, \mathcal{R}_N \cdot \mathcal{CM}_N] \\
\left[\mathcal{S}_1^2 \cdot \mathcal{M}_1, \ldots, \mathcal{S}_N^2 \cdot \mathcal{M}_N\right] \; op \; \left[\mathcal{R}_1^2 \cdot \mathcal{CM}_1, \ldots, \mathcal{R}_N^2 \cdot \mathcal{CM}_N\right] \\
\ldots \\
\left[\mathcal{S}_1^K \cdot \mathcal{M}_1, \ldots, \mathcal{S}_N^K \cdot \mathcal{M}_N\right] \; op \; \left[\mathcal{R}_1^K \cdot \mathcal{CM}_1, \ldots, \mathcal{R}_N^K \cdot \mathcal{CM}_N\right]
\end{gathered}$$

and all new solvers $S_i^k, i \in [1..N]$ and $R_j^k, j \in [1..M]$, $k \in [2..K]$, are created using the same solver declaration of solvers S_i and R_j, respectively.

1.6 Step-by-step POSL code example

In this section we explain how to create a solver using POSL through an example. POSL creates solvers based on local search meta-heuristics algorithms. These algorithms have a common structure: 1. they start by initializing some data structures (e.g. a *tabu list* for *Tabu Search* [34], a *temperature* for *Simulated Annealing* [32], etc.). 2. An initial configuration s is generated. 3. A new configuration s' is selected from the neighborhood $\mathcal{V}(s)$. 4. If s' is a solution for the problem P, then the process stops, and s' is returned. If not, the data structures are updated, and s' is accepted or not for the next iteration, depending on some criterion.An example of such data structure can be the penalizing features of local optima defined by Boussaïd et al [31] in their algorithm *Guided Local Search*.

Abstract computation modules composing *local search meta–heuristics* are the following:

$\boxed{\textit{Abstract Computation module} - 1}$ I: Generating a configuration s

$\boxed{\textit{Abstract Computation module} - 2}$ V: Defining the neighborhood $\mathcal{V}(s)$

$\boxed{\textit{Abstract Computation module} - 3}$ S: Selecting $s' \in \mathcal{V}(s)$

$\boxed{\textit{Abstract Computation module} - 4}$ A: Evaluating an acceptance criteria for s'

To be more specific in our example, we describe come concrete *computation modules* that we can use:

$\boxed{\mathit{Computation\ module} - 1}$ I_{rand}: Generates a random configuration s

$\boxed{\mathit{Computation\ module} - 2}$ V_{1ch}: Defines the neighborhood $\mathcal{V}(s)$ changing only one element

$\boxed{\mathit{Computation\ module} - 3}$ S_{best}: Selects the best configuration $s' \in \mathcal{V}(s)$ improving the current cost.

$\boxed{\mathit{Computation\ module} - 4}$ $A_{a.i.}$: Evaluates an acceptance criteria for s'. We have chosen the classical module selecting the configuration with the lowest global cost, *i.e.*, the one which is likely the closest to a solution.

We can combine modules to create more complex ones. For example, in the example we use in this section, we want to apply some selection criteria, but if there is not improvement in the current configuration, we select a configuration randomly. To do so, we use the operator (?) to combine the *computation module* 3 with another:

$\boxed{\mathit{Computation\ module} - 3.1}$ S_{rand}: Selects a random configuration $s' \in \mathcal{V}(s)$.

Let's make this solver a little bit more complex: in the time of applying the acceptance criteria, suppose that we want to receive a configuration from other solver, to compare it with ours and take it if its cost is lower. We can do that applying the operator (m) to combine the *computation module* 4 with a *communication module*:

$\boxed{\mathit{Communication\ module} - 1:}$ $C.M.$: Receiving a configuration.

Figure 1.7 shows a graphic representation of the *abstract solver* that we create for our example. In this Figure all the modules are abstract, so we can instantiate them afterwards to create concrete solvers. Note that we also want to send the output of the *computation module A*, in order to share the current configuration with other solvers.

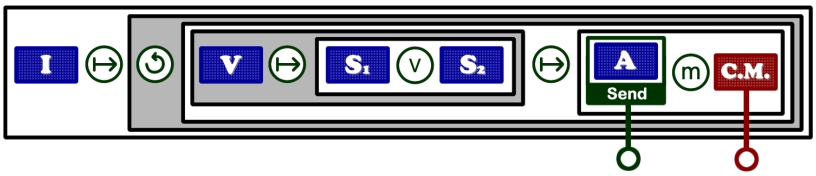


Figure 1.7: Graphic representation of an *abstract solver*

Algorithm 4 shows the POSL pseudo-code for the *abstract solver* described above, using predefined operators, and Algorithm 5 shows the concrete solver definition using the concrete *computation* and *communication modules* already presented. In the later, we have used the concrete *communication module* CM_{last}, that receives all the configuration sent to it, and selects the last arrived.

In practice, POSL provides information regarding the execution process, such as number of ITERATIONS, solver execution TIME, BEST found solution, among others.

Algorithm 4: POSL pseudo-code for *abstract solver* presented in Figure 1.7

```
abstract solver as_example
computation : I, V, S_1, S_2, A
connection: C.M.
    begin
        I (↦)
        [↺ (Itr % K_2) begin
            [V (↦) [S_1 (∨) S_2]] (↦) [(|A|)^o (m) C.M.]
        end]
    end
```

Algorithm 5: An instantiation of the *abstract solver* in Algorithm 4

```
solver S implements as_example
computation : I_rand, V_1ch, S_best, A_a.i.
connection: CM_last
```

Now, supose that we want to create another solver Z using a different neighborhood function, that we will call V_{2ch}, performing two changes instead of only one. As a communication strategy, we want also connect them through the operator *1 to N*, using Z as sender and S as receiver. Then, we need to declare, using name spare expansions, how many solvers we want to connect each others. Algorithm 6 shows the desired communication strategy.

Algorithm 6: A communication strategy

```
[ Z · A (3) ] (⇝) [ S · CM_last (3) ] 2 ;
```

Notice that the connection operation is affected also by the number 2 at the end, to expand also the topology. In that sense, supposing that we have 12 units of computation available, we obtain a *solver set* working on parallel following the topology described in Figure 1.8.

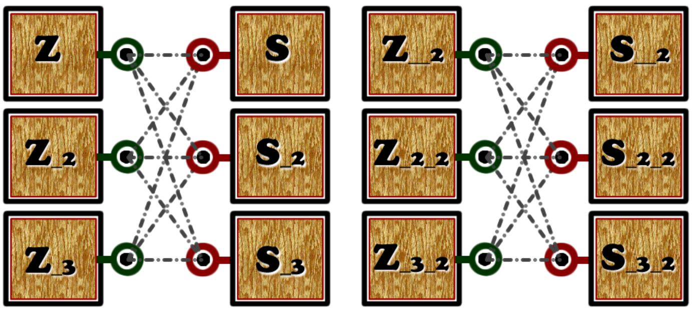


Figure 1.8: An example of connection strategy for 12 units of computation

2

Bibliography

[1] 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, 2012.

[2] Danny Munera, Daniel Diaz, Salvador Abreu, and Philippe Codognet. A Parametric Framework for Cooperative Parallel Local Search. In *Evolutionary Computation in Combinatorial Optimisation*, volume 8600 of *LNCS*, pages 13–24. Springer, 2014.

[3] Stephan Frank, Petra Hofstedt, and Pierre R. Mai. Meta-S: A Strategy-Oriented Meta-Solver Framework. In *Florida AI Research Society (FLAIRS) Conference*, pages 177–181, 2003.

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

[5] Mahuna Akplogan, Jérome Dury, Simon de Givry, Gauthier Quesnel, Alexandre Joannon, Arnauld Reynaud, Jacques Eric Bergez, and Frédérick Garcia. A Weighted CSP approach for solving spatio-temporal planning problem in farming systems. In *11th Workshop on Preferences and Soft Constraints Soft 2011.*, Perugia, Italy, 2011.

[6] Louise K. Sibbesen. *Mathematical models and heuristic solutions for container positioning problems in port terminals*. Doctor of philosophy, Technical University of Danemark, 2008.

[7] Wolfgang Espelage and Egon Wanke. The combinatorial complexity of masterkeying. *Mathematical Methods of Operations Research*, 52(2):325–348, 2000.

[8] Barbara M Smith. Modelling for Constraint Programming. *Lecture Notes for the First International Summer School on Constraint Programming*, 2005.

[9] Ignasi Abío and Peter J Stuckey. Encoding Linear Constraints into SAT. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming*, pages 75–91. Springer, 2014.

[10] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-Carlo Tree Search: A New Framework for Game AI. *AIIDE*, pages 216–217, 2008.

[11] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–49, 2012.

[12] Christian Bessiere. Constraint Propagation. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, chapter 3, pages 29–84. Elsevier, 1st edition, 2006.

[13] Daniel Chazan and Willard Miranker. Chaotic relaxation. *Linear Algebra and its Applications*, 2(2):199–222, 1969.

[14] Patrick Cousot and Radhia Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundations. In *ACM Symposium on Artificial Intelligence amd Programming Languages*, volume 12, pages 1–12, Rochester, NY, 1977.

[15] Krzysztof R. Apt. From Chaotic Iteration to Constraint Propagation. In *24th International Colloquium on Automata, Languages and Programming (ICALP'97)*, pages 36–55, 1997.

[16] Éric Monfroy and Jean-Hugues Réty. Chaotic Iteration for Distributed Constraint Propagation. In *ACM symposium on Applied computing SAC '99*, pages 19–24, 1999.

[17] Éric Monfroy. A coordination-based chaotic iteration algorithm for constraint propagation. In *Proceedings of The 15th ACM Symposium on Applied Computing, SAC 2000*, pages 262–269. ACM Press, 2000.

[18] Peter Zoeteweij. Coordination-based distributed constraint solving in DICE. In *Proceedings of the 18th ACM Symposium on Applied Computing (SAC 2003)*, pages 360–366, New York, 2003. ACM Press.

[19] Farhad Arbab. Coordination of Massively Concurrent Activities. Technical report, Amsterdam, 1995.

[20] Laurent Granvilliers and Éric Monfroy. Implementing Constraint Propagation by Composition of Reductions. In *Logic Programming*, pages 300–314. Springer Berlin Heidelberg, 2001.

[21] Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates. The Iterator and Composite Patterns. Well-Managed Collections. In *Head First Design Patterns*, chapter 9, pages 315–384. O'Relliy, 1st edition, 2004.

[22] Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates. The Observer Pattern. Keeping your Objects in the know. In *Head First Design Patterns*, chapter 2, pages 37–78. O'Relliy, 1st edition, 2004.

[23] Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates. Introduction to Design Patterns. In *Head First Design Patterns*, chapter 1, pages 1–36. O'Relliy, 1st edition, 2004.

[24] Charles Prud'homme, Xavier Lorca, Rémi Douence, and Narendra Jussien. Propagation engine prototyping with a domain specific language. *Constraints*, 19(1):57–76, sep 2013.

[25] Ian P. Gent, Chris Jefferson, and Ian Miguel. Watched Literals for Constraint Propagation in Minion. *Lecture Notes in Computer Science*, 4204:182–197, 2006.

[26] Mikael Z. Lagerkvist and Christian Schulte. Advisors for Incremental Propagation. *Lecture Notes in Computer Science*, 4741:409–422, 2007.

[27] Narendra Jussien, Hadrien Prud'homme, Charles Cambazard, Guillaume Rochart, and François Laburthe. Choco: an Open Source Java Constraint Programming Library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08),*, Paris, France, 2008.

[28] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. MiniZinc: Towards A Standard CP Modelling Language. In *Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

[29] Ibrahim H Osman and Gilbert Laporte. Metaheuristics : A bibliography. *Annals of Operations research*, 63(5):511–623, 1996.

[30] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: overview and conceptual comparison. *ACM Computing Surveys (CSUR)*, 35(3):268–308, 2003.

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

[32] Alexander G. Nikolaev and Sheldon H. Jacobson. Simulated Annealing. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146, chapter 1, pages 1–39. Springer, 2nd edition, 2010.

[33] Aris Anagnostopoulos, Laurent Michel, Pascal Van Hentenryck, and Yannis Vergados. A simulated annealing approach to the travelling tournament problem. *Journal of Scheduling*, 2(9):177—-193, 2006.

[34] Michel Gendreau and Jean-Yves Potvin. Tabu Search. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146, chapter 2, pages 41–59. Springer, 2nd edition, 2010.

[35] Iván Dotú and Pascal Van Hentenryck. Scheduling Social Tournaments Locally. *AI Commun*, 20(3):151—-162, 2007.

[36] Christos Voudouris, Edward P.K. Tsang, and Abdullah Alsheddy. Guided Local Search. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146, chapter 11, pages 321–361. Springer, 2 edition, 2010.

[37] Patrick Mills and Edward Tsang. Guided local search for solving SAT and weighted MAX-SAT problems. *Journal of Automated Reasoning*, 24(1):205–223, 2000.

[38] Pierre Hansen, Nenad Mladenovie, Jack Brimberg, and Jose A. Moreno Perez. Variable neighborhood Search. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146, chapter 3, pages 61–86. Springer, 2010.

[39] Noureddine Bouhmala, Karina Hjelmervik, and Kjell Ivar Overgaard. A generalized variable neighborhood search for combinatorial optimization problems. In *The 3rd International Conference on Variable Neighborhood Search (VNS'14)*, volume 47, pages 45–52. Elsevier, 2015.

[40] Edmund K. Burke, Jingpeng Li, and Rong Qu. A hybrid model of integer programming and variable neighbourhood search for highly-constrained nurse rostering problems. *European Journal of Operational Research*, 203(2):484–493, 2010.

[41] Thomas A. Feo and Mauricio G.C. Resende. Greedy Randomized Adaptive Search Procedures. *Journal of Global Optimization*, (6):109–134, 1995.

[42] Mauricio G.C Resende. Greedy randomized adaptive search procedures. In *Encyclopedia of optimization*, pages 1460–1469. Springer, 2009.

[43] Philippe Galinier and Jin-Kao Hao. A General Approach for Constraint Solving by Local Search. *Journal of Mathematical Modelling and Algorithms*, 3(1):73–88, 2004.

[44] Philippe Codognet and Daniel Diaz. Yet Another Local Search Method for Constraint Solving. In *Stochastic Algorithms: Foundations and Applications*, pages 73–90. Springer Verlag, 2001.

[45] Yves Caniou, Philippe Codognet, Florian Richoux, Daniel Diaz, and Salvador Abreu. Large-Scale Parallelism for Constraint-Based Local Search: The Costas Array Case Study. *Constraints*, 20(1):30–56, 2014.

[46] Danny Munera, Daniel Diaz, Salvador Abreu, Francesca Rossi, and Philippe Codognet. Solving Hard Stable Matching Problems via Local Search and Cooperative Parallelization. In *29th AAAI Conference on Artificial Intelligence*, Austin, TX, 2015.

[47] Kazuo Iwama, David Manlove, Shuichi Miyazaki, and Yasufumi Morita. Stable marriage with incomplete lists and ties. In *ICALP*, volume 99, pages 443–452. Springer, 1999.

[48] David Gale and Lloyd S. Shapley. College Admissions and the Stability of Marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[49] Laurent Michel and Pascal Van Hentenryck. A constraint-based architecture for local search. *ACM SIGPLAN Notices*, 37(11):83–100, 2002.

[50] Dynamic Decision Technologies Inc. *Dynadec. Comet Tutorial.* 2010.

[51] Laurent Michel and Pascal Van Hentenryck. The comet programming language and system. In *Principles and Practice of Constraint Programming*, pages 881–881. Springer Berlin Heidelberg, 2005.

[52] Jorge Maturana, Álvaro Fialho, Frédéric Saubion, Marc Schoenauer, Frédéric Lardeux, and Michèle Sebag. Adaptive Operator Selection and Management in Evolutionary Algorithms. In *Autonomous Search*, pages 161–189. Springer Berlin Heidelberg, 2012.

[53] Colin R. Reeves. Genetic Algorithms. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146, chapter 5, pages 109–139. Springer, 2010.

[54] Marco Dorigo and Thomas Stützle. Ant colony optimization: overview and recent advances. In *Handbook of Metaheuristics*, volume 146, chapter 8, pages 227–263. Springer, 2nd edition, 2010.

[55] Konstantin Chakhlevitch and Peter Cowling. Hyperheuristics : Recent Developments. In *Adaptive and multilevel metaheuristics*, pages 3–29. Springer, 2008.

[56] Patricia Ryser-Welch and Julian F. Miller. A Review of Hyper-Heuristic Frameworks. In *Proceedings of the Evo20 Workshop, AISB*, 2014.

[57] Kevin Leyton-Brown, Eugene Nudelman, and Galen Andrew. A portfolio approach to algorithm selection. In *IJCAI*, pages 1542–1543, 2003.

[58] Horst Samulowitz, Chandra Reddy, Ashish Sabharwal, and Meinolf Sellmann. Snappy: A simple algorithm portfolio. In *Theory and Applications of Satisfiability Testing - SAT 2013*, volume 7962 LNCS, pages 422–428. Springer, 2013.

[59] Alexander E.I. Brownlee, Jerry Swan, Ender Özcan, and Andrew J. Parkes. Hyperion 2. A toolkit for {meta- , hyper-} heuristic research. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO Comp '14, pages 1133–1140, Vancouver, BC, 2014. ACM.

[60] Enrique Urra, Daniel Cabrera-Paniagua, and Claudio Cubillos. Towards an Object-Oriented Pattern Proposal for Heuristic Structures of Diverse Abstraction Levels. *XXI Jornadas Chilenas de Computación 2013*, 2013.

[61] Laura Dioşan and Mihai Oltean. Evolutionary design of Evolutionary Algorithms. *Genetic Programming and Evolvable Machines*, 10(3):263–306, 2009.

[62] John N. Hooker. Toward Unification of Exact and Heuristic Optimization Methods. *International Transactions in Operational Research*, 22(1):19–48, 2015.

[63] El-Ghazali Talbi. Combining metaheuristics with mathematical programming, constraint programming and machine learning. *4or*, 11(2):101–150, 2013.

[64] Éric Monfroy, Frédéric Saubion, and Tony Lambert. Hybrid CSP Solving. In *Frontiers of Combining Systems*, pages 138–167. Springer Berlin Heidelberg, 2005.

[65] Éric Monfroy, Frédéric Saubion, and Tony Lambert. On Hybridization of Local Search and Constraint Propagation. In *Logic Programming*, pages 299–313. Springer Berlin Heidelberg, 2004.

[66] Jerry Swan and Nathan Burles. Templar - a framework for template-method hyper-heuristics. In *Genetic Programming*, volume 9025 of *LNCS*, pages 205–216. Springer International Publishing, 2015.

[67] Sébastien Cahon, Nordine Melab, and El-Ghazali Talbi. ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics. *Journal of Heuristics*, 10(3):357–380, 2004.

[68] Youssef Hamadi, Éric Monfroy, and Frédéric Saubion. An Introduction to Autonomous Search. In *Autonomous Search*, pages 1–11. Springer Berlin Heidelberg, 2012.

[69] Roberto Amadini and Peter J Stuckey. Sequential Time Splitting and Bounds Communication for a Portfolio of Optimization Solvers. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming*, volume 1, pages 108–124. Springer, 2014.

[70] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Features for Building CSP Portfolio Solvers. *arXiv:1308.0227*, 2013.

[71] Christophe Lecoutre. XML Representation of Constraint Networks. Format XCSP 2.1. *Constraint Networks: Techniques and Algorithms*, pages 541–545, 2009.

[72] Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. *Modeling and Programming with Gecode*. 2013.

[73] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Introduction to Parallel Computing. In *Introduction to Parallel Computing*, chapter 1, pages 1–9. Addison Wesley, 2nd edition, 2003.

[74] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 746–749, New York, 2007. ACM.

[75] Mark D. Hill and Michael R. Marty. Amdahl's Law in the multicore era. *IEEE Computer*, (7):33–38, 2008.

[76] Peter Sanders. Engineering Parallel Algorithms: The Multicore Transformation. *Ubiquity*, 2014(July):1–11, 2014.

[77] Javier Diaz, Camelia Muñoz-Caro, and Alfonso Niño. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1369–1386, 2012.

[78] Joel Falcou. Parallel programming with skeletons. *Computing in Science and Engineering*, 11(3):58–63, 2009.

[79] Ian P Gent, Chris Jefferson, Ian Miguel, Neil C A Moore, Peter Nightingale, Patrick Prosser, and Chris Unsworth. A Preliminary Review of Literature on Parallel Constraint Solving. In *Proceedings PMCS 2011 Workshop on Parallel Methods for Constraint Solving*, 2011.

[80] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Embarrassingly Parallel Search. In *Principles and Practice of Constraint Programming*, pages 596–610. Springer, 2013.

[81] Akihiro Kishimoto, Alex Fukunaga, and Adi Botea. Evaluation of a simple, scalable, parallel best-first search strategy. *Artificial Intelligence*, 195:222–248, 2013.

[82] Yuu Jinnai and Alex Fukunaga. Abstract Zobrist Hashing : An Efficient Work Distribution Method for Parallel Best-First Search. *30th AAAI Conference on Artificial Intelligence (AAAI-16)*.

[83] Alejandro Arbelaez and Luis Quesada. Parallelising the k-Medoids Clustering Problem Using Space-Partitioning. In *Sixth Annual Symposium on Combinatorial Search*, pages 20–28, 2013.

[84] Hue-Ling Chen and Ye-In Chang. Neighbor-finding based on space-filling curves. *Information Systems*, 30(3):205–226, may 2005.

[85] Pavel Berkhin. Survey Of Clustering Data Mining Techniques. Technical report, Accrue Software, Inc., 2002.

[86] Farhad Arbab and Éric Monfroy. Distributed Splitting of Constraint Satisfaction Problems. In *Coordination Languages and Models*, pages 115–132. Springer, 2000.

[87] Mark D. Hill. What is Scalability? *ACM SIGARCH Computer Architecture News*, 18:18–21, 1990.

[88] Danny Munera, Daniel Diaz, and Salvador Abreu. Solving the Quadratic Assignment Problem with Cooperative Parallel Extremal Optimization. In *Evolutionary Computation in Combinatorial Optimization*, pages 251–266. Springer, 2016.

[89] Stefan Boettcher and Allon Percus. Nature's way of optimizing. *Artificial Intelligence*, 119(1):275–286, 2000.

[90] Daisuke Ishii, Kazuki Yoshizoe, and Toyotaro Suzumura. Scalable Parallel Numerical CSP Solver. In *Principles and Practice of Constraint Programming*, pages 398–406, 2014.

[91] Charlotte Truchet, Alejandro Arbelaez, Florian Richoux, and Philippe Codognet. Estimating Parallel Runtimes for Randomized Algorithms in Constraint Solving. *Journal of Heuristics*, pages 1–36, 2015.

[92] 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.

[93] Youssef Hamadi, Cedric Piette, Said Jabbour, and Lakhdar Sais. Deterministic Parallel DPLL system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:127–132, 2011.

[94] Andre A. Cire, Sendar Kadioglu, and Meinolf Sellmann. Parallel Restarted Search. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 842–848, 2011.

[95] Long Guo, Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Diversification and Intensification in Parallel SAT Solving. *Principles and Practice of Constraint Programming*, pages 252–265, 2010.

[96] M Yasuhara, T Miyamoto, K Mori, S Kitamura, and Y Izui. Multi-Objective Embarrassingly Parallel Search. In *IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, pages 853–857, Singapore, 2015. IEEE.

[97] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Improvement of the Embarrassingly Parallel Search for Data Centers. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming*, pages 622–635, Lyon, 2014. Springer.

[98] Prakash R. Kotecha, Mani Bhushan, and Ravindra D. Gudi. Efficient optimization strategies with constraint programming. *AIChE Journal*, 56(2):387–404, 2010.

[99] Peter Zoeteweij and Farhad Arbab. A Component-Based Parallel Constraint Solver. In *Coordination Models and Languages*, pages 307–322. Springer, 2004.

[100] Akihiro Kishimoto, Alex Fukunaga, and Adi Botea. Scalable, Parallel Best-First Search for Optimal Sequential Planning. *In ICAPS-09*, pages 201–208, 2009.

[101] Claudia Schmegner and Michael I. Baron. Principles of optimal sequential planning. *Sequential Analysis*, 23(1):11–32, 2004.

[102] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Programming Using the Message-Passing Paradigm. In *Introduction to Parallel Computing*, chapter 6, pages 233–278. Addison Wesley, second edition, 2003.

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

[104] Mauro Birattari, Mark Zlochin, and Marrco Dorigo. Towards a Theory of Practice in Metaheuristics Design. A machine learning perspective. *RAIRO-Theoretical Informatics and Applications*, 40(2):353–369, 2006.

[105] Agoston E Eiben and Selmar K Smit. Evolutionary algorithm parameters and methods to tune them. In *Autonomous Search*, pages 15–36. Springer Berlin Heidelberg, 2011.

[106] Maria-Cristina Riff and Elizabeth Montero. A new algorithm for reducing metaheuristic design effort. *IEEE Congress on Evolutionary Computation*, pages 3283–3290, jun 2013.

[107] Holger H. Hoos. Automated algorithm configuration and parameter tuning. In *Autonomous Search*, pages 37–71. Springer Berlin Heidelberg, 2012.

[108] Frank Hutter, Holger H Hoos, and Kevin Leyton-brown. ParamILS: An Automatic Algorithm Configuration Framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

[109] Frank Hutter. Updated Quick start guide for ParamILS, version 2.3. Technical report, Department of Computer Science University of British Columbia, Vancouver, Canada, 2008.

[110] Volker Nannen and Agoston E. Eiben. Relevance Estimation and Value Calibration of Evolutionary Algorithm Parameters. *IJCAI*, 7, 2007.

[111] S. K. Smit and A. E. Eiben. Beating the 'world champion' evolutionary algorithm via REVAC tuning. *IEEE Congress on Evolutionary Computation*, pages 1–8, jul 2010.

[112] E. Yeguas, M.V. Luzón, R. Pavón, R. Laza, G. Arroyo, and F. Díaz. Automatic parameter tuning for Evolutionary Algorithms using a Bayesian Case-Based Reasoning system. *Applied Soft Computing*, 18:185–195, may 2014.

[113] Agoston E. Eiben, Robert Hinterding, and Zbigniew Michalewicz. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2):124–141, 1999.

[114] Junhong Liu and Jouni Lampinen. A Fuzzy Adaptive Differential Evolution Algorithm. *Soft Computing*, 9(6):448–462, 2005.

[115] A Kai Qin, Vicky Ling Huang, and Ponnuthurai N Suganthan. Differential evolution algorithm with strategy adaptation for global numerical optimization. *IEEE Transactions on Evolutionary Computation*, 13(2):398–417, 2009.

[116] Vicky Ling Huang, Shuguang Z Zhao, Rammohan Mallipeddi, and Ponnuthurai N Suganthan. Multi-objective optimization using self-adaptive differential evolution algorithm. *IEEE Congress on Evolutionary Computation*, pages 190–194, 2009.

[117] Martin Drozdik, Hernan Aguirre, Youhei Akimoto, and Kiyoshi Tanaka. Comparison of Parameter Control Mechanisms in Multi-objective Differential Evolution. In *Learning and Intelligent Optimization*, pages 89–103. Springer, 2015.

[118] Jeff Clune, Sherri Goings, Erik D. Goodman, and William Punch. Investigations in Meta-GAs: Panaceas or Pipe Dreams? In *GECOO'05: Proceedings of the 2005 Workshop on Genetic an Evolutionary Computation*, pages 235–241, 2005.

[119] Emmanuel Paradis. R for Beginners. Technical report, Institut des Sciences de l'Evolution, Université Montpellier II, 2005.

[120] Scott Rickard. Open Problems in Costas Arrays. In *IMA International Conference on Mathematics in Signal Processing at The Royal Agricultural College*, Cirencester, UK., 2006.

[121] Frédéric Lardeux, Éric Monfroy, Broderick Crawford, and Ricardo Soto. Set Constraint Model and Automated Encoding into SAT: Application to the Social Golfer Problem. *Annals of Operations Research*, 235(1):423–452, 2014.

[122] Konstantinos Drakakis. A review of Costas arrays. *Journal of Applied Mathematics*, 2006:32 pages, 2006.

[123] Jordan Bell and Brett Stevens. A survey of known results and research areas for n-queens. *Discrete Mathematics*, 309(1):1–31, 2009.

[124] Rok Sosic and Jun Gu. Effcient Local Search with Conflict Minimization: A Case Study of the N-Queens Problem. *IEEE Transactions on Knowledge and Data Engineering*, 6:661–668, 1994.

[125] Stephen W. Soliday, Abdollah. Homaifar, and Gary L. Lebby. Genetic algorithm approach to the search for Golomb Rulers. In *International Conference on Genetic Algorithms*, volume 1, pages 528–535, Pittsburg, 1995.

[126] Alejandro Reyes-amaro, Éric Monfroy, and Florian Richoux. POSL: A Parallel-Oriented metaheuristic-based Solver Language. In *Recent developments of metaheuristics*, to appear. Springer.

[127] Alejandro Reyes-Amaro, Éric Monfroy, and Florian Richoux. A Parallel-Oriented Language for Modeling Constraint-Based Solvers. In *Proceedings of the 11th edition of the Metaheuristics International Conference (MIC 2015)*. Springer, 2015.