An Architecture for Composite Combinatorial Optimization Solvers

Khalil Chrit
NOVA LINCS, University of Évora, Portugal

Jean-François Baffier
IIJ Research, Tokyo, Japan

Pedro Patinho
NOVA LINCS, University of Évora, Portugal

Salvador Abreu
NOVA LINCS, University of Évora, Portugal

Abstract

In this paper, we introduce elements for `MoSCO`, a framework for building hybrid metaheuristic-based solvers from a collection of reusable base components. The framework is implemented in Julia and provides a modular architecture for composing solvers through a pipeline-based approach. The modular design of `MoSCO` supports the creation of reusable components and adaptable solver strategies for various Constraint Satisfaction Problems (CSPs) and Constraint Optimization Problems (COPs). We validate `MoSCO`'s utility through practical examples, demonstrating its effectiveness in reconstructing established metaheuristics and enabling the creation of novel solver configurations. This work lays the foundation for future developments in automated solver construction and parameter optimization.

2012 ACM Subject Classification Software and its engineering → Constraint and logic languages; Software and its engineering → Formal language definitions; Theory of computation → Discrete optimization

Keywords and phrases Hybrid Metaheuristics, DSL

Digital Object Identifier 10.4230/OASIcs.SLATE.2025.8

Funding This work was partially supported by the Portuguese Foundation for Science and Technology under PhD fellowship UI/BD/153047/2022 and strategic project UID/04516 (NOVA LINCS).

1 Introduction

Metaheuristics are a class of problem-independent algorithms known for their ability to efficiently find approximate solutions to hard combinatorial optimization problems, e.g. traveling salesman problem (TSP) [1], quadratic assignment problem (QAP) [8] and vehicle routing problem (VRP) [23]. Metaheuristics are search techniques that navigate the search space adeptly, often producing near-optimal solutions to problems that are otherwise intractable. Despite their success, developing and fine-tuning metaheuristic solvers is a challenging problem in itself, with difficulties in guiding the stochastic process and the high sensitivity to parameter changes which impact intensification and diversification of the search process. Designing and implementing effective metaheuristic solvers requires domain expertise and much trial-and-error, making the whole process time-consuming and burdensome [3, 22]. Frequently, a particular metaheuristic will be well suited to a particular problem or problem instance, but poorly so for other problems or even just instances. This situation led to the development of *hybrid metaheuristics*, which combine multiple approaches in order to more efficiently solve a wider class of problems and instances [18].

We take a step back and propose a modular framework in which to construct hybrid metaheuristic-based solvers called MoSCO. The goal is mainly to allow for an abstracted description of the structure of a *hybrid* metaheuristic solver, which is used to synthesize a functioning problem-specific solver, by combining parts of existing solvers which are connected in a network which conducts the process of deriving a stream of solutions for the original problem, meaning to reach the optimum.

We validate the proposed architecture through a Julia-based implementation, showcasing MoSCO's potential to reconstruct well-known reference metaheuristics by applying it to a well-known problem domain.

The remainder of this paper is structured as follows: in section 2 we discuss previous approaches to the problem of describing metaheuristic solvers and proceed, in section 3, to introduce the general architecture of MoSCO. Section 4 is used to describe a textual representation for the language and provide examples. In section 5, we outline the structure of our prototype developed in Julia and finally, in section 7 we assess our work and set possible lines for future developments.

2 Related Work

2.1 Parallel-Oriented Solver Language (POSL)

Reyes-Amaro et al. [21] proposed the Parallel-Oriented Solver Language (POSL), a framework for building interconnected meta-heuristic based solvers that work in parallel. POSL focuses on sharing not only information but also behaviors among solvers, allowing for solver modifications during runtime. While POSL shares similarities with our proposed solver architecture, our work emphasizes the use of a Domain Specific Language (DSL) and incorporates meta-learning algorithms for optimizing solver parameters and architecture.

2.2 ParadisEO

Cahon et al. [7] introduced ParadisEO, a white-box object-oriented software framework dedicated to the flexible design of metaheuristic algorithms for combinatorial optimization. ParadisEO provides a broad range of metaheuristic components and encourages code reusability, allowing users to design their own solvers by assembling and customizing existing components. Our proposed architecture shares the idea of modularity and code reusability with ParadisEO but also emphasizes the use of a DSL to connect components and incorporates meta-learning algorithms for optimization.

2.3 jMetal

Durillo and Nebro [10] developed jMetal, a Java-based framework for multi-objective optimization with metaheuristics. jMetal provides a rich set of components and tools for designing, experimenting with, and benchmarking metaheuristic algorithms. While jMetal focuses on multi-objective optimization, our proposed architecture is designed for general combinatorial optimization problems and emphasizes the use of a DSL and meta-learning algorithms for optimizing solver parameters and architecture.

In summary, several approaches have been proposed to facilitate the development of metaheuristic-based solvers for combinatorial optimization problems. Our proposed modular solver architecture builds upon existing work by providing a problem-independent architecture design, incorporating a Domain Specific Language for connecting components, with plans to incorporate meta-learning algorithms to automatically optimize solver parameters and architecture.

3 Modular Solver Architecture

Here, we provide a detailed description of the various elements that make up the proposed architecture. Figure 1 shows the input/output and parameters of the operator, which forms the basic building block of our architecture.

3.1 Configuration Streams

Within the solver, the processing of solution candidates follows defined pathways, formally represented as a directed graph $G = (V, E)$, where V is the set of operators and E represents the connections between them, which we call *Configuration Streams*. Each edge $e \in E$ corresponds to a Configuration Stream that transmits configurations (solution candidates) from one operator to another.

▶ **Definition 1.** *A* Configuration Stream S *is a communication channel between operators that manages the transmission of configurations, defined as:*

$$S : \mathcal{O}_{source} \to \mathcal{O}_{destination} \tag{1}$$

where $\mathcal{O}_{source}$ *and* $\mathcal{O}_{destination}$ *are operators, and* S *transmits a set of configurations* $C = \{c_1, c_2, \ldots, c_n\}$*, where each* $c_i \in \Omega$ *represents a potential solution within the solution space* Ω*.*

- *When* $|C| = 1$*, the stream transmits exactly one configuration at a time. Typically used in trajectory-based metaheuristics such as Simulated Annealing [15] or Tabu Search [11].*
- *When* $|C| > 1$*, the stream manages a set of configurations sequentially. This supports population-based metaheuristics like Genetic Algorithms [12] or Particle Swarm Optimization [14], where multiple solution candidates evolve concurrently.*

3.2 Operators

Operators are the functional components that transform configurations as they flow through the solver pipeline. Each operator implements a specific algorithmic behavior that forms part of the overall search process.

▶ **Definition 2.** *An* operator $\mathcal{O}$ *is a transformation function defined as:*

$$\mathcal{O} : \mathcal{I}_{\mathcal{O}} \to \mathcal{O}_{\mathcal{O}}$$

Where:

- $\mathcal{I}_{\mathcal{O}}$ *represents the specific* "input domain" *for operator* $\mathcal{O}$*. This domain could be drawn from various sets depending on the operator, including the problem specification space* Θ *(for the* init *operator), the solver representation space* $\mathcal{S}$ *(for the* gen *operator specifically), the configuration space* $\mathcal{A}$*, the power set* $\mathcal{P}(\mathcal{A})$*, or Cartesian products like* $\mathcal{A} \times \mathcal{A}$*.*
- $\mathcal{O}_{\mathcal{O}}$ *represents the specific* "output range" *for operator* $\mathcal{O}$*, typically being* $\mathcal{A}$*, or* $\mathcal{P}(\mathcal{A})$*, and* $\mathcal{S}$ *for the* init *operator.*
- $\mathcal{A}$ *represents the* "set of possible configurations" *(i.e., the type of a single solution candidate) within the solution space* Ω*.*

Operators manipulate configurations, often taking elements of type $\mathcal{A}$ *or sets* $\mathcal{P}(\mathcal{A})$ *as the input and/or producing them as output.*

Next, we present the core operators within the architecture:

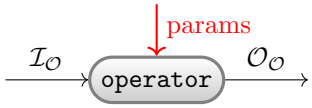

Figure 1 Graphical representation of an operator with input $\mathcal{I}_\mathcal{O}$, output $\mathcal{O}_\mathcal{O}$, and configurable parameters.

Init: Responsible for setting up the problem domain, encoding the CSP model into a solver-compatible representation (which is then used by other components to generate and evaluate solution candidates), and defining the search space and the domain of the decision variables, the objective function, constraints, and other relevant information (meta-data). Formally, the Init operator can be defined as:

$$\mathcal{O}_{\text{Init}} : \Theta \to \mathcal{S} \tag{2}$$

It encodes the problem instance Θ into a solver-compatible representation $\mathcal{S}$, which includes definitions for V (Variables), D (Domains), $f : \mathcal{A} \to \mathbb{R}$ (objective function), and the set of constraints $\mathcal{C}$ that implicitly define the solution space Ω. The Init operator can be parameterized with r (random) or h (heuristic) initialization strategies (illustrated in Figure 2).

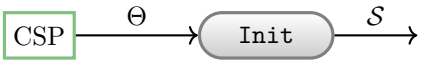

Figure 2 The Init operator transforms a problem instance into a solver-compatible representation.

Generate: Responsible for instantiating the initial set of solution candidates within the defined search space. This operator transforms the abstract problem definition into concrete configuration instances that serve as the starting point for the optimization process. Formally, the generate operator can be defined as:

$$\mathcal{O}_{\text{Generate}} : \mathcal{S} \to \mathcal{P}(\mathcal{A}) \tag{3}$$

Where $\mathcal{P}(\mathcal{A})$ denotes the set of configurations, allowing for the generation of either a singleton set (single configuration) or a population of configurations (multiple candidates). The operator's behavior can be parameterized with:

- $n = |\mathcal{P}(\mathcal{A})|$: number of generated configurations.
- σ: initialization strategy.
  - σ_r: Random, assigns values to decision variables according to a uniform probability distribution over their respective domains.
  - σ_h: Heuristic, employs problem-specific knowledge to generate configurations with potentially higher initial quality.

▶ Note. The parameter n doesn't determine whether the subsequent search process follows a trajectory-based or a population-based approach (this is determined by the compilator, which we'll describe in section 4.2). It only determines the number of configurations to generate.

Neighbourhood: Performs moves in the search space to generate neighboring solutions. It supports various move types including swap, shift, block moves or any other move that can be defined for the problem at hand.

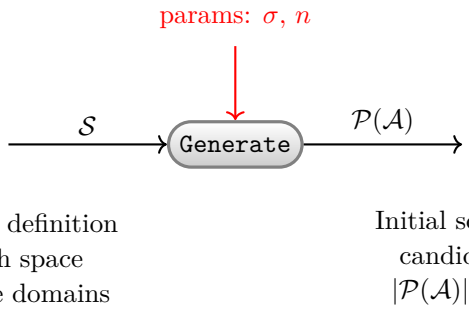

Figure 3 The Generate operator creates initial solution candidates from the problem representation.

A Neighbourhood operator, defined as: $\mathcal{O}_{\text{Neighbourhood}} : \mathcal{A} \times \mathcal{M} \to \mathcal{P}(\mathcal{A})$
is a function that, given a configuration $c \in \mathcal{A}$ and a move type $m \in \mathcal{M}$, computes the corresponding set of neighbours, denoted by $N(c, m) \in \mathcal{P}(\mathcal{A})$.
Where $N(c, m)$ is the set of neighbours of configuration c generated using move type $m \in \mathcal{M} = \{\text{swap}, \text{insert}, \text{delete}, \text{replace}, \cdots\}$. The specific neighbours generated depend on the implementation associated with the move type m.

When a stream delivers a set of configurations $C = \{c_1, c_2, \ldots, c_k\}$ to this operator, the intended architectural behavior is that the operator is implicitly applied to each configuration in the set. This would conceptually produce a collection of neighbour sets $\{N(c_1, m), N(c_2, m), \ldots, N(c_k, m)\}$.

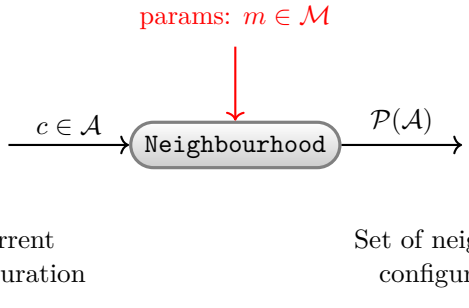

Figure 4 The Neighbourhood operator generates a set of configurations in the vicinity of the current solution.

Accept: This operator determines whether to accept or reject a candidate configuration $c_{\text{candidate}}$ based on the current configuration c_{current} and a chosen acceptance criterion a. Formally, it is a function $\mathcal{O}_{\text{Accept}} : \mathcal{A} \times \mathcal{A} \times \mathcal{Q} \to \mathcal{A}$ defined as:

$$\mathcal{O}_{\text{Accept}}(c_{\text{current}}, c_{\text{candidate}}, a) = \begin{cases} c_{\text{candidate}} & \text{if criterion } a \text{ evaluates to true} \\ c_{\text{current}} & \text{otherwise} \end{cases} \tag{4}$$

where $a \in \mathcal{Q}$ represents the selected acceptance criterion from a set of possible strategies. The acceptance criteria in $\mathcal{Q}$ include (but not limited to):

Always: The candidate is always accepted: $a_{\text{always}}(c_{\text{current}}, c_{\text{candidate}}) = \text{true}$.

Improvement: Accepted only if it improves the current solution: $a_{\text{improv}}(c_{\text{current}}, c_{\text{candidate}}) = f(c_{\text{candidate}}) < f(c_{\text{current}})$.

Probabilistic: Acceptance with a fixed probability p: Let $u \sim \text{Uniform}(0,1)$, then $a_{\text{probability}}(c_{\text{current}}, c_{\text{candidate}}, p) = u < p$.

Temperature-based (Simulated Annealing): Accepts improving solutions, or worsening ones with a probability dependent on a temperature parameter T. Let $\Delta f = f(c_{\text{candidate}}) - f(c_{\text{current}})$: $a_{\text{temperature}}(c_{\text{current}}, c_{\text{candidate}}, T) = (\Delta f < 0) \vee (u < \exp(-\frac{\Delta f}{T}))$

▶ Note.

- The described acceptance criteria assume a minimization problem where lower objective function values are preferred.
- Selection strategies like those used in *Tabu Search*, which involve evaluating multiple non-forbidden neighbors and selecting the best, are typically implemented in this framework through a sequence of operators (e.g., Neighbourhood, filtering based on the Tabu list, and selection using a dedicated operator or a `MUX` connector configured with a 'best' strategy) rather than as a single criterion within the Accept operator.

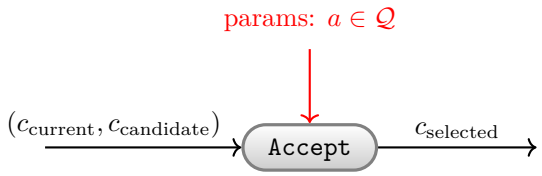

Figure 5 The Accept operator decides whether to accept a candidate solution based on the defined strategy.

Loop: This operator repeats the execution of an internal operator or sequence of operators until a specific termination condition is met.
Formally, it is represented as $\mathcal{O}_{\text{Loop}} : (\mathcal{A} \to \mathcal{A}) \times \mathcal{T} \times \mathcal{A} \to \mathcal{A}$.
Where $c^{(0)} = c_{\text{initial}}$, and $c^{(i)} = \text{Op}(c^{(i-1)})$ for $i \geq 1$.
The loop terminates after k iterations, where k is the smallest index for which the termination condition $\tau \in \mathcal{T}$ is true (the condition's evaluation may depend on the current iteration k, the current configuration $c^{(k)}$, elapsed time, solution history, etc.).
The output of the Loop operator is the final configuration $c_{\text{final}} = c^{(k)}$.
The termination conditions τ supported are: iteration count (i.e. stopping after a fixed number of iterations ($i_{\max}$)), exceeding a specific time limit ($t_{\max}$), convergence (reaching a point where the solution quality has not significantly improved over a certain period, defined by a threshold ϵ), or achieving a desired target objective function value (f_{target}).

▶ Note. Unlike operators that perform a direct transformation on a configuration in a single step, the Loop operator acts as a "control-flow construct", it manages the iterative execution process and maintains states related to termination conditions (e.g., iteration count, solution history).

3.3 Connectors

The connectors are the links between components that allow for a custom flow of data within the solver. They can be used to merge or split configuration streams (enabling parallelization of the search process [16]). Just like the loop operator (although fundamentally different), the

connectors are not operators in the traditional sense, as they are not responsible for performing any transformation on configurations, but rather, only for controlling the execution flow within the solver.

There are two types of connectors:

- **DEMUX**: Split a single configuration stream into multiple streams, each of which can be processed separately. $\mathcal{O}_{\text{DEMUX}} : \mathcal{A} \times \mathbb{N} \times \mathcal{B} \to \mathcal{P}(\mathcal{A})$ is a function that's defined as:

$$\mathcal{O}_{\text{DEMUX}}(c, n, p) = \{c_1, c_2, \ldots, c_n\} \tag{5}$$

  Where:
  - $c \in \mathcal{A}$ is the input configuration
  - $n \in \mathbb{N}$ is the number of output streams
  - $p \in \mathcal{B}$ is a flag indicating whether the streams should be processed in parallel
  - $c_i = c$ for all $1 \leq i \leq n$ (the same configuration is sent to all output streams)

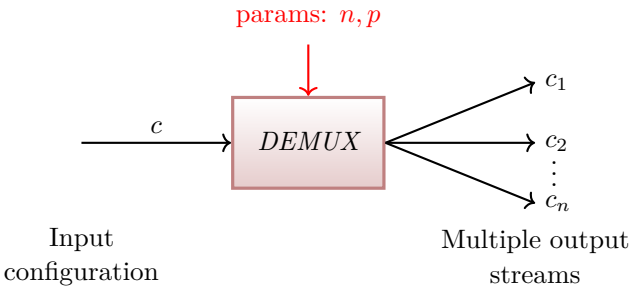

Figure 6 The DEMUX connector splits a single input configuration into multiple output streams.

- **MUX (Multiplexer/Selector)**: This component acts as a selection point, merging multiple candidate configuration streams and potentially comparing them to select a single output configuration. Formally, it can be represented as a function:

$$\mathcal{O}_{\text{MUX}} : \mathcal{P}(\mathcal{A}) \times \mathbb{N} \times \mathcal{S} \to \mathcal{A} \tag{6}$$

  Given a set of incoming candidate configurations $C_{\text{candidates}} = \{c'_1, c'_2, \ldots, c'_n\} \in \mathcal{P}(\mathcal{A})$, and a selection strategy $s \in \mathcal{S}$, and an index $i \in \mathbb{N}$ the MUX outputs a single selected configuration $c_{\text{selected}} \in C_{\text{candidates}}$.

  Indexed Candidate (i): Select the candidate c'_i arriving from the i-th conceptual input stream (this refers to the stream source, not an ordering within the set $C_{\text{candidates}}$).

  Predicate based selection: Select the candidate c'_i where $s(c'_i) = \text{true}$ for $1 \leq i \leq n$. The set $\mathcal{S}$ of selection strategies includes, but is not limited to:
  - **Best**: $s_{\text{best}}(c_i) = f(c_i) = \min_{1 \leq j \leq n} f(c_j)$ for minimization
  - **Random**: $s_{\text{random}}(c_i) = \text{true}$ with probability $\frac{1}{n}$

3.4 Leashed Solvers

Leashed solvers are composite entities within the framework that encapsulate a set of operators to act as a singular unit. Think of a leashed solver as a complete, potentially complex, solver (i.e. a local search algorithm [13], or even another metaheuristic) that is embedded within a larger solver architecture but operates under strict external control - it's kept "on a leash" by the primary solver framework. The key aspects of this concept are:

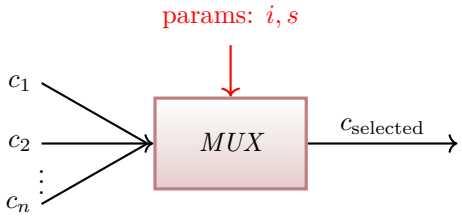

Figure 7 The MUX connector merges multiple input streams into a single output configuration.

- **Encapsulation**: A leashed solver bundles a complete metaheuristic solver, either external or a composite of of `MoSCO` operators (might also include its own loops, state, and logic) into a single reusable entity.
- **Externally Controlled execution**: This is the crucial "leash" part. The enclosing solver dictates when the leashed solver is executed, what input it receives (e.g., a specific configuration to improve), and how long or under what condition it runs.
- **Limited autonomy & resources**: It doesn't run indefinitely or consume unbounded resources. Its execution is typically bounded by parameters like iteration count, time limit, or convergence criteria.
- **Stateful interaction**: Given that the operators are stateless transformations, a leashed solver is expected to maintain its own internal state during execution, but this state might be reset or initialized each time it's called by the main solver.
- **Dictated purpose**: The primary solver uses the leashed solver strategically as a sub-routine for a specific purpose, and not just a generic step (e.g. intensification, diversification, neighbourhood exploration). A leashed solver acts as a stateful and controllable "sub-routine" within the pipeline, offering more complexity than a simple operator, but less independence than running a completely separate solver process.

In essence, leashed solvers allow for arbitrarily complex operations to be bundled and utilized as if they were a singular component. The driving idea is to use existing solver strategies but cap their ability to work towards a solution (hence the "leash" analogy.) Moreover, this design provides a mechanism for modularity and encourages the integration of existing specialized solvers.

4 Domain Specific Language

The elements introduced in the previous sections combine to form hybrid solvers. This may be expressed using a textual representation, which we now expound on. This section introduces the domain-specific language (DSL) meant to describe entire solvers and their structure, and gives a few examples.

4.1 Syntax

The DSL grammar can be formally defined as a 4-tuple $G = (N, T, P, S)$ where:

- N is the set of non-terminal symbols:

$$N = \{\text{Solver}, \text{Expression}, \text{Operator}, \text{Sequence}, \text{Loop}, \text{Parallel}\} \tag{7}$$

- T is the set of terminal symbols, consisting of operator names, parameters, and syntactic delimiters:

- Operator names: `init`, `gen`, `neighbourhood`, `accept`, ...
- Syntactic delimiters: `|`, `*`, `(`, `)`, `[`, `]`, `=`, `→`, ...

- P is the set of production rules, with Solver as the start symbol:

$$\text{Solver} \rightarrow \text{Expression} \tag{8}$$
$$\text{Expression} \rightarrow \text{Operator} \mid \text{Sequence} \mid \text{Loop} \mid \text{Parallel} \tag{9}$$
$$\text{Operator} \rightarrow \text{OperatorName} \mid \text{OperatorName(Parameters)} \tag{10}$$
$$\text{Sequence} \rightarrow \text{Expression} \mid \text{Expression} \tag{11}$$
$$\text{Loop} \rightarrow \text{(Expression)*} \mid \text{(Expression)*(Condition)} \tag{12}$$
$$\text{Parallel} \rightarrow \text{[Expression, Expression, ...]} \tag{13}$$

- S is the start symbol:

$$S = \text{Solver} \tag{14}$$

For future versions, we plan to implement *Conditional Processing*, selecting one of multiple paths based on a condition

$$\text{Conditional} \rightarrow \{\text{Expression, Expression, ...}\} \tag{15}$$

This feature will enable more dynamic solver behavior based on runtime conditions but is not yet implemented in the current version.

Variable Binding

The DSL also supports variable binding to create more complex data flows:

```
solver = init | gen > current_state;
current_state > loop (
neighbourhood | accept )
```

Where "`op > variable_name`" binds the output of an operator "op" to a variable, which can be referenced by subsequent operators. This enables feedback loops and more complex control structures.
An alternative, more explicit syntax would be:

```
solver =
init > v1;
gen < v1 > v2;
loop < v2 (
neighbourhood < current > candidates;
(current, candidates) > accept > current;
)
```

In this notation:

- Each statement ends with a semicolon.
- `op < v1` provides input to an operator.
- `op < v1 > v2` combines both input and output binding.
- `v1 > op > v2` is equivalent to `op < v1 > v2`.
- Multiple inputs can be specified as a tuple: `< (v1, v2) >`

This notation makes the data flow more explicit and enables complex connections between operators.

4.2 DSL Examples

Here we illustrate how common metaheuristic algorithms can be expressed using the DSL.

- **Parallel Solver Branches**

```
solver = init | gen | DEMUX | [
  neighbourhood | accept(criterion=improvement),
  neighbourhood(move_type=swap) | accept
  ] | MUX(strategy=best)
```

  This solver creates two parallel search paths with different neighborhood and acceptance strategies and combines results by selecting the best solution found in either branch.

- **Stochastic Local Search**

```
solver = init | gen(n=1) | loop(
  neighbourhood(move_type=swap) |
  accept(criterion=probability, alpha=0.5)
  )*(it=1000)
```

  This representation captures the essence of Stochastic Local Search:
  1. Initialize the problem.
  2. Generate a single starting solution.
  3. Generate a neighbouring solution using the swap move.
  4. Accept the neighbour if it is better than the current solution, otherwise accept with a certain probability (alpha).
  5. repeat for 1000 iterations.

- **Simulated Annealing**

```
solver = init | gen(n=1) | loop(
  neighbourhood(move_type=swap) |
  accept(criterion=temperature, T0=100, alpha=0.95)
  )*(it=1000)
```

  This specifies:
  1. Initialize the problem.
  2. Generate a single starting solution.
  3. Generate a neighbouring solution using the swap move.
  4. Accept the neighbour based on a temperature schedule (with intial temperature t0 and cooling factor alpha).
  5. repeat for 1000 iterations.

- **Tabu Search**

```
solver = init | gen(n=1) | loop(
  neighbourhood(move_type=swap) |
  accept(criterion=tabu)
  )*(it=1000)
```

  The structure of this solver is the same as the previous examples, with the difference that the accept operator uses the tabu criterion, which translates to a tabu search solver.

▶ Note. When operators do not have parameters explicitly specified, they use their default values. In this case, the `accept(criterion=tabu)` operator uses a default tabu list size of 7.

- **Iterated Local Search**[17]

```
solver = init | gen(n=1) > current_best;
loop (
    current_best >
        perturbation_operator >
        perturbed_sol;
    perturbed_sol >
        local_search_procedure >
        new_solution;
    new_solution >
        accept(criterion=ils) >
        current_best
) * (iterations=1000)
```

In this example, we used the variable binding notation, as well as a leashed solver to encapsulate the local search procedure. `perturbation_operator` and `local_search_procedure` are `leashed solvers`, created from a sequence of basic operators.

4.3 Graphical Representation

The DSL can be visually represented as a directed graph $G_V = (V, E, \lambda)$ where:

- V represents the set of nodes corresponding to operators and connectors
- $E \subseteq V \times V$ represents the set of directed edges corresponding to configuration streams
- $\lambda : V \to \Omega$ is a labeling function that maps nodes to their operational semantics

The mapping between DSL constructs and graphical elements follows the principles expressed in Table 1.

Table 1 Mapping between DSL constructs and their graphical representations.

DSL Construct	Graph Representation
Operator (Op)	Node with label Op
Sequence (Op1 | Op2)	Directed edge from Op1 to Op2
Loop ((Expr)*)	Directed edge from Expr to Loop
Parallel ([Expr1, Expr2, ...])	DEMUX node with multiple outgoing edges, followed by parallel paths, ending with MUX node

The simplest solver follows a linear pipeline from initialization to solution output, as shown in figure 8.

Figure 8 Simple sequential solver.

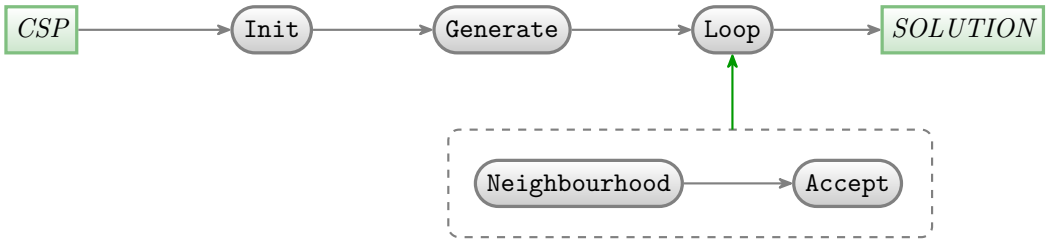

Figure 9 A simple iterative solver using the Loop operator representation.

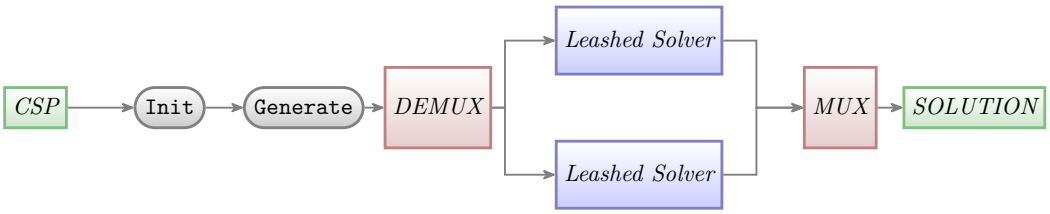

Figure 10 Simple solver with parallel processing paths using leashed solvers.

The graphical representation provides an intuitive visualization of flow of configurations and execution within the solver, making it easier to understand complex solver architectures. This complements the textual DSL by providing a clear mapping between syntactic constructs and their semantic interpretation in terms of computation and data flow. Figures 8, 9, and 10 provide examples of different solver configurations using the graphical notation.

5 Prototype Implementation

To validate the design choices made in the previous sections, we are currently working on a prototype, which should serve as a testbed for experimenting with the DSL, the composition of operators, and the overall workflow for constructing hybrid metaheuristic solvers.

The prototype is being implemented in Julia, chosen for its high performance for scientific computing and its metaprogramming capabilities, which are advantageous for implementing the DSL and the solver components efficiently. The overall implementation is composed of the following key modules:

1. **DSL Parser**: The module responsible for parsing solver descriptions written in the DSL. It transforms the textual representation into an internal Abstract Syntax Tree (AST) that represents the structure and components of the solver pipeline.
2. **Solver Engine**: The core execution engine interprets the AST generated by the parser. It manages the flow of configurations through the specified sequence of operators and connectors and executes them based on the defined pipeline structure. It handles the instantiation of operators, the management of configuration streams, and the control flow.
3. **Modular Component Library**: A library containing implementations of the core operators (e.g., `Init`, `Generate`, `Neighbourhood`, `Accept`) and connectors (`MUX`, `DEMUX`) described in Section 3. It should be easily extensible, to allow new operators or variations of existing ones to be added. Each component follows the defined input/output interfaces to ensure modularity and composability.
4. **Problem Interface**: An abstraction layer designed to decouple the solver engine and components from specific problem model descriptions. It defines structures for representing problem instances (variables, domains, constraints) and configurations.

The interaction between these modules goes as follows: the parser reads the DSL input, the engine uses the resulting AST to configure and run the pipeline, invoking operators from the component library, which in turn operate on problem-specific data structures defined via the problem interface. Listing 1 shows an example of how the DSL (tabu search example in Section 4.2) is translated and parsed into Julia code, resulting in the construction of a solver pipeline using the proposed architecture.

Listing 1 Tabu search solver pipeline in Julia.

```
  Pipeline([
    Loop(
        Pipeline([
            Neighbor(SwapMove()),
            Accept(:tabu, Dict{Symbol,Any}(:tabu_size => tabu_size))
        ]),
        max_iterations,
        0.0,  # Target fitness for early stopping
    )
])
```

While still under active development, we aim to demonstrate the expressiveness of the framework in representing known metaheuristics and the flexibility of the proposed architecture in creating novel hybrid solvers. Initial experiments focus on reconstructing standard algorithms like Simulated Annealing, Tabu Search and basic parallel search strategies to verify the interaction between the core components.

6 Experimental Results for Magic Square

We constructed five distinct solvers: Tabu Search[11], two Simulated Annealing variants, GRASP (Greedy Randomized Adaptive Search Procedure), and a hybrid of Tabu Search and Simulated Annealing. Each solver employed a local search strategy with appropriate neighborhood operators, acceptance criteria, and control flow defined within the DSL.

Table 2 shows the performance of six solver configurations on the 3x3 magic square problem, including a comparison with a standard implementation from the `Metaheuristics.jl` package. The column "Parameters Used" shows how the core parameters for each metaheuristic (such as tabu list size, SA's initial temperature and cooling rate, or GRASP's greediness factor) are declaratively specified within the DSL. The results indicate that both the moderate Simulated Annealing configuration (t0=10) and the Hybrid (TS+SA) solver achieved a 100% success rate. The Hybrid solver was efficient, finding a solution in an average of 301ms and 4514 iterations. The aggressive SA variant (t0=20) also performed well with an 87% success rate, while Tabu Search proved robust, achieving a 74% success rate. GRASP, in its current setup, found solutions remarkably quickly in its successful runs (averaging only 3ms and 71 iterations) but exhibited a lower overall success rate of 27%. For comparison, the `Metaheuristics.jl` SA solver achieved a 62% success rate. It is notably faster, which is expected since our framework is currently developed with a focus on features and architectural flexibility first; performance optimizations will be addressed once the implementation is finalized.

Table 2 Comparison of `MoSCO`-specified Solvers on 3x3 Magic Square (100 runs, max 50k iterations/run).

Solver	Parameters Used	Avg Time (ms, successful)	Avg Iterations (successful)	Success Rate (%)
Tabu Search	`tabu_list_size=7`	383	16779	74.0
Simulated Annealing (Moderate)	`t0=10, cooling_rate=0.95`	397	11757	100.0
Simulated Annealing (Aggressive)	`t0=20, cooling_rate=0.95`	577	17249	87.0
GRASP	`greediness=0.3`	3	71	27.0
Hybrid (TS+SA)	`tabu_list_size=7,t0=20, cooling_rate=0.95`	301	4514	100.0
Metaheuristics.jl (SA)	`Default`	119	N/A	62.0

▶ Note. The `Metaheuristics.jl` package uses an internal, automatic annealing schedule that is linear with respect to the fraction of evaluations completed (T = nevals/max_evals) unlike the classic geometric cooling schedule implemented in our SA variants. This means its temperature and cooling rate are not directly tunable by the user, hence the 'Default' parameter listing.

This comparative overview highlights the framework's potential for easy and rapid experimentation, as well as the analysis of solver designs based on their core components.

Further benchmarking will involve more complex problems, larger instances, and a wider array of solver configurations and performance metrics to rigorously assess the framework's capabilities and the performance characteristics of the generated solvers.

7 Future Directions

We presented a modular architecture for constructing meta-heuristic based solvers. This approach simplifies the process of creating high-performance solvers for different problem domains, by providing a DSL for connecting reusable components into complete architectures. We also presented an analysis framework that can be used to measure the performance of the generated architectures in terms of the quality of solutions and speedups obtained. In the future, we plan to further develop the DSL to include additional elements, such as logical expressions and control flow constructs.

Additionally, we plan to explore the use of meta-learning algorithms to optimize both the parameters and overall architecture of our proposed solver [5] [20] [9] [6], incorporating techniques from the reinforcement learning world to guide the design and configuration of solver components.

We also want to widen the range of problem domains, in order to demonstrate the solvers' adaptability. This will involve testing the architecture on problems with varying levels of complexity and characteristics.

Another line of work which we plan to develop is the design of an XCSP3 [4] or MiniZinc [19] back-end, which would allow us to plug in existing complex model descriptions. A related aspect would be the inclusion of specialized components for global constraints [2].

Overall, the proposed architecture has the potential to improve the performance of existing optimization algorithms and generate new, high-performance solvers for a wide range of problem domains, by simplifying the design process and incorporating meta-learning algorithms for optimization.

References

1 David L Applegate, Robert E Bixby, Vasek Chvatal, and William J Cook. The traveling salesman problem: a computational study. *Princeton university press*, 2006.

2 Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global constraint catalogue: Past, present and future. *Constraints An Int. J.*, 12(1):21–62, 2007. doi:10.1007/S10601-006-9010-8.

3 Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM computing surveys (CSUR)*, 35(3):268–308, 2003. doi:10.1145/937503.937505.

4 Frédéric Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. Xcsp3: an integrated format for benchmarking combinatorial constrained problems. *arXiv preprint arXiv:1611.03398*, 2016.

5 Edmund Burke, Graham Kendall, Jim Newall, Emma Hart, Peter Ross, and Sonia Schulenburg. Hyper-heuristics: An emerging direction in modern search technology. In *Handbook of metaheuristics*, pages 457–474. Springer, 2003. doi:10.1007/0-306-48056-5_16.

6 Edmund K Burke, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and John R Woodward. A classification of hyper-heuristic approaches. In *Handbook of metaheuristics*, pages 449–468. Springer, 2010.

7 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. doi:10.1023/B:HEUR.0000026900.92269.EC.

8 Eranda Çela. *The quadratic assignment problem: theory and algorithms*. Springer Science & Business Media, 1998.

9 Konstantin Chakhlevitch and Peter Cowling. Hyperheuristics: recent developments. In *Adaptive and multilevel metaheuristics*, pages 3–29. Springer, 2008. doi:10.1007/978-3-540-79438-7_1.

10 Juan J Durillo and Antonio J Nebro. jmetal: A java framework for multi-objective optimization. *Advances in Engineering Software*, 42(10):760–771, 2011. doi:10.1016/J.ADVENGSOFT.2011.05.014.

11 Fred Glover. Tabu search—part i. *ORSA Journal on computing*, 1(3):190–206, 1989.

12 John Henry Holland et al. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

13 Holger H Hoos and Thomas Stützle. *Stochastic local search: Foundations and applications*. Elsevier, 2004.

14 James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of International Conference on Neural Networks (ICNN'95), Perth, WA, Australia, November 27 - December 1, 1995*, pages 1942–1948. IEEE, 1995. doi:10.1109/ICNN.1995.488968.

15 Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983. doi:10.1126/SCIENCE.220.4598.671.

16 Manuel López-Ibáñez, Danny Múnera, Daniel Díaz, and Salvador Abreu. Weaving of metaheuristics with cooperative parallelism. In *Parallel problem solving from nature—PPSN XV: 15th international conference, coimbra, portugal, september 8–12, 2018, proceedings, part I 15*, pages 436–448. Springer, 2018.

17 Helena R. Lourenço, Olivier C. Martin, and Thomas Stützle. *Iterated Local Search*, pages 320–353. Springer US, Boston, MA, 2003. doi:10.1007/0-306-48056-5_11.

18 Danny Munera, Daniel Diaz, and Salvador Abreu. Hybridization as cooperative parallelism for the quadratic assignment problem. In *Hybrid Metaheuristics: 10th International Workshop, HM 2016, Plymouth, UK, June 8-10, 2016, Proceedings 10*, pages 47–61. Springer, 2016. doi:10.1007/978-3-319-39636-1_4.

19 Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007. doi:10.1007/978-3-540-74970-7_38.

20 Ender Özcan, Burak Bilgin, and Emin Erkan Korkmaz. A comprehensive analysis of hyper-heuristics. *Intelligent Data Analysis*, 12(1):3–23, 2008. URL: http://content.iospress.com/articles/intelligent-data-analysis/ida00313.

21 Alejandro Reyes-Amaro, Eric Monfroy, and Florian Richoux. Posl: A parallel-oriented metaheuristic-based solver language. *International Journal of Metaheuristics*, 6(1-2):6–29, 2017.

22 El-Ghazali Talbi. *Metaheuristics: from design to implementation*, volume 74. John Wiley & Sons, 2009.

23 Paolo Toth and Daniele Vigo. *The vehicle routing problem*. SIAM, 2002.