- **Dynamic:** The dynamic strategy often initializes the α value to a random value at each iteration of the GRASP metaheuristic. For instance, a uniform distribution may be used in the range $[0.5, 0.9]$, or a decreasing nonuniform distribution.
- **Adaptive:** In this strategy, the value of α is self-tuned. The value is updated automatically during the search function of the memory of the search.

Example 2.42 Self-tuning in reactive GRASP. In reactive GRASP, the value of the parameter α is periodically updated according to the quality of the obtained solutions [623]. At each iteration, the parameter value is selected from a discrete set of possible values $\Psi = \{\alpha_1, \ldots, \alpha_m\}$. The probability associated with each α_i is initialized to a same value $p_i = 1/m, i \in [1, \ldots, m]$. The adaptive initialization in the reactive GRASP consists in updating these probabilities according to the quality of solutions obtained for each α_i. Let z^* be the incumbent solution and A_i the average value of all solutions found using $\alpha = \alpha_i$. Then, the probability p_i for each value of α is updated as follows:

$$p_i = \frac{q_i}{\sum_{j=1}^{m} q_j}, \quad i \in [1, \ldots, m]$$

where $q_j = z^*/A_i$. Hence, larger values of p_i correspond to more suitable values for the parameter α_i.

In addition to the parameters associated with the randomized greedy heuristic, the GRASP metaheuristic will inherit the parameters of the embedded S-metaheuristic. The larger is the variance between the solutions of the construction phase, the larger is the variance of the obtained local optima. The larger is the variance between the initial solutions, the better is the best found solution and the more time consuming is the computation.

2.10 S-METAHEURISTIC IMPLEMENTATION UNDER ParadisEO

ParadisEO–MO (moving objects) is the module of the software framework ParadisEO dedicated to the design of single-solution based metaheuristics (S-metaheuristics). An important aspect in ParadisEO–MO is that the common search concepts of both metaheuristics and S-metaheuristics are factored. All search components are defined as templates (generic classes). ParadisEO uses the genericity concept of objects to make those search mechanisms adaptable. The user implements an S-metaheuristic by deriving the available templates that provide the functionality of the different search components: problem-specific templates (e.g., objective function, representation) and problem-independent templates (e.g., neighbor selection, cooling schedule, and stopping criteria).

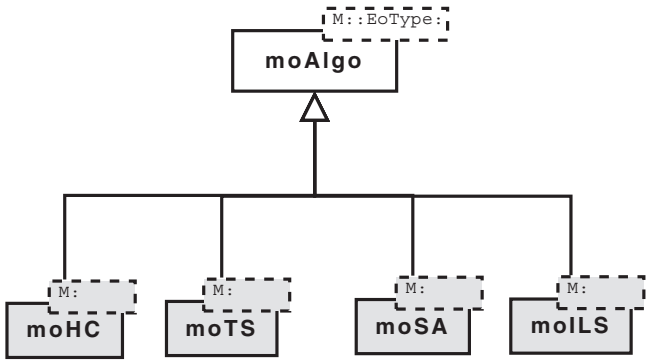

FIGURE 2.38 UML diagram of the `moAlgo` template representing an S-metaheuristic such as hill climbing, tabu search, simulated annealing, iterated local search.

In ParadisEO–MO, the `moAlgo` template represents an S-metaheuristic such as local search (hill climbing), tabu search, simulated annealing, and iterated local search (Fig. 2.38).

Figure 2.43 shows the common concepts and relationships in single-solution based metaheuristics. Hence, most of the search components will be reused by different S-metaheuristics. The aim of this section is to show not only the easy way to design an S-metaheuristic but also the high flexibility to transform an S-metaheuristic to another one reusing most of the design and implementation work. The different implementations are illustrated using the symmetric TSP problem.

2.10.1 Common Templates for Metaheuristics

As seen in Chapter 1, the common search components that have to be designed for any metaheuristic are

- **Objective function:** The objective function is associated with the template `eoEvalFunc`, which is a problem-specific template. For the TSP problem, it corresponds to the total distance. It is straightforward to implement.
- **Representation of solutions:** The encoding used for the TSP is based on permutations. Hence, an object `Route` is designed to deal with this structure. It corresponds to a permutation vector and its associated fitness. This is also a problem-specific template. However, one can reuse some popular representations. In fact, in ParadisEO, the template associated with the encoding of solutions defines, for instance, the binary, discrete vector, real vectors, and trees. Those encodings may be reused to solve an optimization problem.
- **Stopping criteria:** The stopping criteria for hill climbing are implicit, that is, the algorithm stops when a local optimal solution is found. Then there is nothing to specify.

2.10.2 Common Templates for S-Metaheuristics

As seen in this chapter, the common search components for S-metaheuristics are

- **Initial solution:** As shown in Section 2.1.3, the initial solution may be generated randomly or by any other heuristic (e.g., available algorithm as a template `moAlgo` or user-specified greedy algorithm).
- **Neighborhood structure and its exploration:** A neighborhood for the TSP may be defined by the 2-opt operator. A class `TwoOpt` is then derived for the template `moMove`, which represents a move. Hence, a `TwoOpt` object is a `moMove` that can be applied to a object of type `Route` associated with the representation. To specify the exploration order of the neighborhood, the following templates have to defined:
  - **First move:** The template `moMoveInit` defines the initial move to apply. The corresponding class in our example is

```
class TwoOptInit : public moMoveInit <TwoOpt>
```

  - **Next move:** The template `moNextMove` defines the next move to apply. This template has also to check the end of the exploration.
- **Incremental evaluation function:** The incremental objective function is associated with the template `moIncrEval`. The user must derive this template to implement the function. According to a solution and a move, this template is able to compute the fitness of the corresponding neighbor with a higher efficiency.

2.10.3 Local Search Template

Figure 2.39 shows the architecture of the hill-climbing template `moHC`. Once the common templates for all metaheuristics and S-metaheuristics are defined, `only one` search component is left, the neighbor selection strategy. The template associated with the selection strategy of the next neighbor is `moMoveselect`. The usual standard strategies are available in the template (see Section 2.3.1). The user has to choose one of them:[17]

- **Best improvement** which corresponds to `moBestImprSelect` template.
- **First improvement** which corresponds to `moFirstImprSelect` template.
- **Random improvement** which corresponds to `moRandImprSelect` template.

From those defined templates, building a hill-climbing algorithm is completely done. The detailed description of the program associated with a hill-climbing algorithm using the defined search components and their associated templates is as follows:

[17] As the framework is a white box, the user can also design other selection strategies.

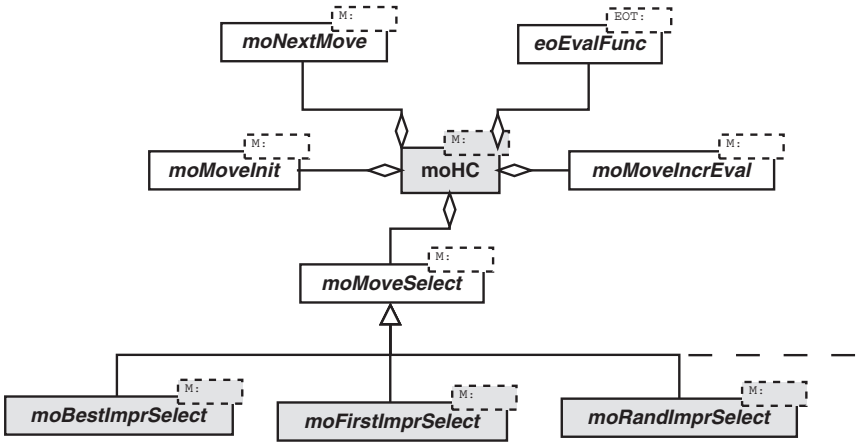

FIGURE 2.39 UML diagram of the hill-climbing template `moHC`. The templates `moEval-Func`, `moNextMove`, `moMoveIncrEval`, and `moMoveInit` are problem specific and need to be completed by the user. The template `moMoveSelect` is problem independent, that is, "plug and play" that can be completely reused.

```
    ...
//An initial solution.
Route route;
//An eoInit object (see ParadisEO-EO).
RouteInit route_init;
//Initialization.
route_init(route);
//The eoEvalFunc.
RouteEvaluation full_eval;
//Solution evaluation.
full_eval (route);
//The moMoveInit object.
TwoOptInit two_opt_init;
//The moNextMove object.
TwoOptNext two_opt_next;
//The moIncrEval object.
TwoOptIncrEval two_opt_incr_eval;
//The moMoveSelect.
moFirstImprSelect <TwoOpt> two_opt_select;
//or moBestImprSelect <TwoOpt> two_opt_select;
//or moRandImprSelect <TwoOpt> two_opt_select;
//or MyMoveSelect <TwoOpt> two_opt_select;
//The moHC object.
moHC <TwoOpt> hill_climbing (two_opt_init, two_opt_next,
    two_opt_incr_eval, two_opt_select,full_eval);
//The HC is launched on the initial solution.
hill_climbing (route) ;
```

A great advantage of using ParadisEO is that the user has to modify a given template (problem specific such as the objective function or problem independent such as the neighbor selection strategy), only the concerned template is updated; there is no impact on other templates and on the whole algorithm. For instance, if the user wants to define his own neighbor selection strategy, he can design a class for the `TwoOpt` move and use it as follows:

```
class MyMoveSelectionStrategy : public moMoveSelect <TwoOpt>
```

Another advantage of using the software framework is the ability to extend a metaheuristic to another metaheuristic in an easy and efficient manner. In the next section, it is shown how the hill-climbing algorithm has been evolved to tabu search, simulated annealing, and iterated local search. Most of the search components and their associated templates are reused.

2.10.4 Simulated Annealing Template

Figure 2.40 shows the architecture of the simulated annealing template `moSA`. If a comparison between Figs 2.39 and 2.40 is done, one can notice that most of the templates (search components) are the same.

The following search components and their associated templates are reused from the hill-climbing S-metaheuristic (moHC):

- Objective function that corresponds to the template `eoEvalFunc`.
- Incremental evaluation function that corresponds to `moMoveIncrEval`.

In addition to the search components associated with local search, the following components have to be defined for simulated annealing algorithm in the template `moSA`:

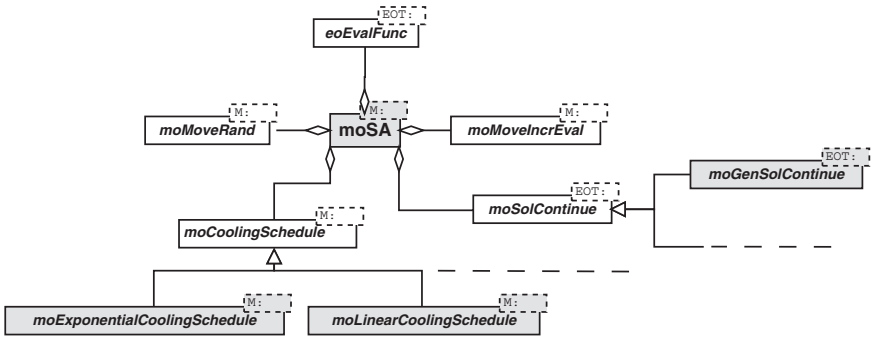

FIGURE 2.40 UML diagram of the simulated annealing template `moSA`. The template `moMoveRand` is specific to the problem; that is, the user has to define it, and the templates `moCoolingSchedule` and `moSolContinue` are problem independent and ready to use.

- **Cooling schedule:** The cooling schedule of simulated annealing is specified in the template `moCoolingSchedule`. Some cooling functions are available, such as the linear function `moLinearCoolingSchedule` and the geometric one `moExponentialCoolingSchedule`.
- **Stopping criteria** For any S-metaheuristic (e.g., tabu search), many defined stopping criteria in the template `moSolContinue` may be used.
- **Random neighbor generation:** The template `moMoveRand` defines how a random neighbor is generated.

In the following simulated annealing program, the lines different from the hill-climbing program are represented in bold. It is easy to extract the characteristics of the implemented SA: a maximum number of iterations as stopping criteria, the Boltzmann distribution acceptance probability, and so on.

```
   ...
// An initial solution.
Route route;
// An eoInit object (see ParadisEO-EO).
RouteInit route_init;
// Initialization.
route_init(route);
// The eoEvalFunc.
RouteEvaluation full_eval;
// Solution evaluation.
full_eval (route);
// The moRandMove object.
TwoOptRand two_opt_rand;
// The moIncrEval object.
TwoOptIncrEval two_opt_incr_eval;
// The chosen cooling schedule object.
moExponentialCoolingSchedule cool_sched (T_min, ratio);
// or moLinearCoolingSchedule cool_sched (T_min, quantity);
// or MyCoolingSchedule cool_sched;
// The moSolContinue.
moGenSolContinue <Route> cont (max_step);
// The moSA object.
moSA <TwoOpt> simulated_annealing (two_opt_rand, two_opt_incr_eval,
                                   cont, T_init, cool_sched, full_eval);
// The simulated_annealing is launched on the initial solution.
simulated_annealing (route) ;
```

2.10.5 Tabu Search Template

Figure 2.41 shows the architecture of the tabu search template `moTS`. If a comparison between Figs 2.39 and 2.41 is done, one can notice that most of the templates (search components) are the same.

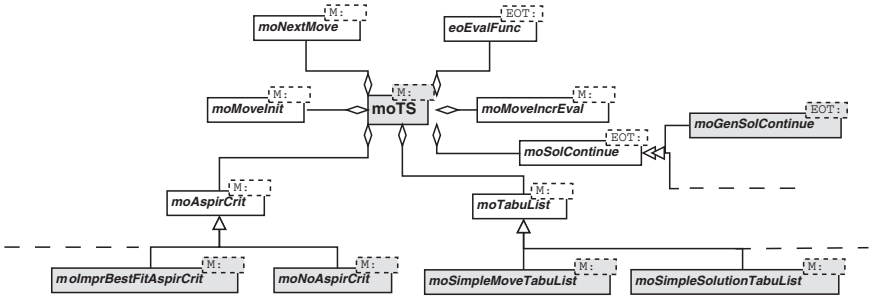

FIGURE 2.41 UML diagram of the tabu search template (moTS object). The templates `moAspirCrit`, `moTabuList`, and `moSolContinue` are problem-independent and ready-to-use templates.

The following search components and their associated templates are reused from the hill-climbing S-metaheuristic (moHC):

- Objective function that corresponds to the template `eoEvalFunc`.
- Incremental evaluation function that corresponds to `moMoveIncrEval`.
- Neighborhood structure and its exploration that corresponds to the templates `moMoveInit`, `moNextMove`.

To design a tabu search algorithm, only the following search components have to be defined in the template `moTS`:

- **Aspiration criteria** that corresponds to the template `moAspirCrit`. Two aspiration criteria are provided: the `moNoAspirCrit` that always rejects a move and the `moImpBestFitAspirCrit` that accepts a move if the fitness of the neighbor is better than the best found solution.
- **Tabu list (short-term memory)** that corresponds to the template `moTabuList`. Two basic tabu lists already exist: the ready-to-use template `moSimpleMoveTabuList` that contains the moves and the template `moSimpleSolutionTabuList` that contains the solutions.
- **Stopping criteria** that corresponds to the template `moSolContinue`. This template allows to select different stopping criteria for tabu search such as
  - `moGenSolContinue`, where the S-metaheuristic stops after a given maximum number of iteration.
  - `moFitSolContinue`, where the algorithm continues its execution until a target fitness (quality) is reached.
  - `moNoFitImprSolContinue`, where the algorithm stops when the best found solution has not be improved since a given number of iterations.
  - `moSteadyFitSolContinue`, a combination of the first and the third criteria: the algorithm performs a given number of iterations and then it stops if the best found solution is not improved for a given number of iterations.

In the following tabu search program, the lines different from the hill-climbing program are represented in bold. It is easy to extract the characteristics of this tabu search: a maximum number of iterations as stopping criterion.

```
    ...
// An initial solution.
Route route;
// An eoInit object (see ParadisEO-EO).
RouteInit route_init;
// Initialization.
route_init(route);
// The eoEvalFunc.
RouteEvaluation full_eval;
// Solution evaluation.
full_eval (route);
// The moMoveInit object.
TwoOptInit two_opt_init;
// The moNextMove object.
TwoOptNext two_opt_next;
// The moIncrEval object.
TwoOptIncrEval two_opt_incr_eval;
// The moTabuList.
moSimpleSolutionTabuList<TwoOpt> tabu_list(10);
// or moSimpleMoveTabuList<TwoOpt> tabu_list(10);
// or MyTabuList<TwoOpt> tabu_list;
// The moAspirCrit.
moNoAspirCrit <TwoOpt> aspir_crit;
// or moImprBestFitAspirCrit<TwoOpt> aspir_crit;
// or MyAspirCrit<TwoOpt> aspir_crit;
//The moSolContinue.
moGenSolContinue <Route> cont (10000);
// or MySolContinue<TwoOpt> cont;
// The moTS object.
moTS <TwoOpt> tabu_search (two_opt_init, two_opt_next, two_opt_incr_eval,
                           tabu_list, aspir_crit, cont, full_eval);
// The TS is launched on the initial solution.
tabu_search (route) ;
```

2.10.6 Iterated Local Search Template

Figure 2.42 shows the architecture of the iterated local search template `moILS`. Once a given S-metaheuristic designed, it is very easy to construct an iterated local search algorithm.

The following search components have to be defined for the ILS algorithm in the template `moILS`:

- **Local search algorithm:** The local search algorithm to be used must be specified in the `moAlgo` template. Any developed S-metaheuristic may be used,

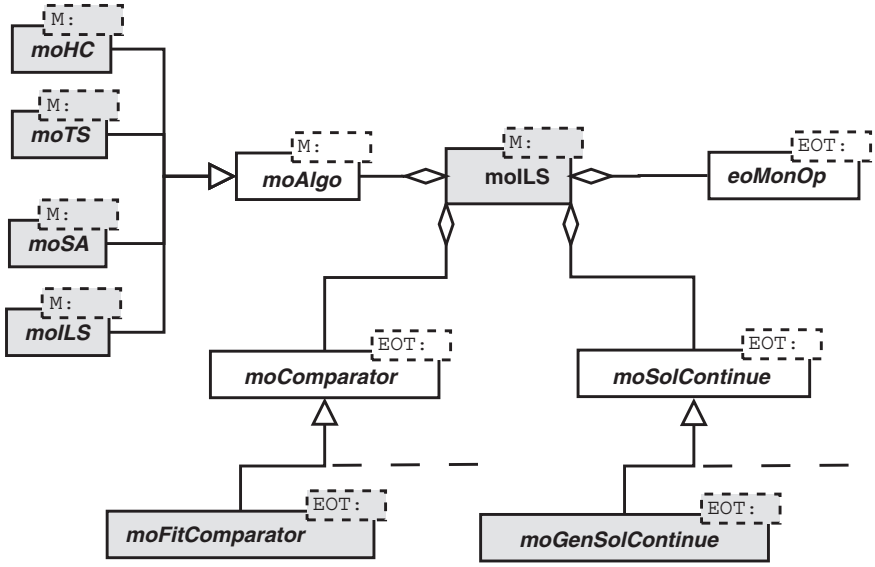

FIGURE 2.42 UML diagram of the iterated local search template `moILS`.

such as hill climbing (`moHC` template), tabu search (`moTS` template), simulated annealing (`moSA` template), any other user-defined S-metaheuristic, or why not another ILS `moILS`.

- **Perturbation method:** The perturbation method is defined in the template `eoMonOp`.
- **Acceptance criteria:** The search component dealing with the acceptance of the generated solution is defined in the template `moComparator`. Some usual acceptance functions are available such as `moFitComparator` that selects the new solution if it is better than the best found. Nevertheless, any specific acceptance method can be implemented; that is, a new class must be defined:

```
class MyComparisonStrategy : public moComparator <Route>
```

- **Stopping criteria:** For any S-metaheuristic (e.g., simulated annealing, tabu search), many defined stopping criteria in the template `moSolContinue` may be used.

Starting from any S-metaheuristic program, the user can easily design an ILS for the symmetric TSP problem. In the following program, the additional lines to a S-metaheuristic program are represented in bold:

```
...
// An initial solution.
Route route;
// An object to initialise this solution.
RouteInit route_init;
```

```
// Initialization.
route_init(route);
// A full evaluation method <=> eoEvalFunc.
RouteEvaluation full_eval;
// Solution evaluation.
full_eval (route);
// The moMoveInit object.
TwoOptInit two_opt_init;
// The moNextMove object.
TwoOptNext two_opt_next;
// The moIncrEval object.
TwoOptIncrEval two_opt_incr_eval;
// moMoveSelect.
moFirstImprSelect <TwoOpt> two_opt_select;
// or moBestImprSelect <TwoOpt> two_opt_select;
// or moRandImprSelect <TwoOpt> two_opt_select;
// or MyMoveSelect <TwoOpt> two_opt_select;
// The moHC object.
moHC <TwoOpt> hill_climbing (two_opt_init, two_opt_next,
                                        two_opt_incr_eval, two_opt_select, full_eval);
   // The moSolContinue object.
moGenSolContinue <Route> cont (1000);
// The moComparator object.
moFitComparator <Route> comparator;
// The eoMonOp, in this example the well known CitySwap.
CitySwap perturbation;
// The moILS object.
moILS<TwoOpt> iterated_local_search(hill_climbing, comparator,
                                                perturbation, full_eval);
//The iterated local search is launched on the initial solution.
iterated_local_search (route) ;
```

2.11 CONCLUSIONS

In addition to the representation, the objective function and constraint handling that are common search concepts to all metaheuristics, the common concepts for single-solution based metaheuristics are as follows (Fig. 2.43):

- **Initial solution:** An initial solution may be specified randomly or by a given heuristic.
- **Neighborhood:** The main concept of S-metaheuristics is the definition of the neighborhood. The neighborhood has an important impact on the performances of this class of metaheuristics. The interdependency between representation and neighborhood must not be neglected. The main design question in S-metaheuristics is the trade-off between the efficiency of the representation/neighborhood and its effectiveness (e.g., small versus large neighborhoods).