Chapter 4
Search Based Software Maintenance: Methods and Tools

Gabriele Bavota, Massimiliano Di Penta and Rocco Oliveto

Summary. Software evolution is an effort-prone activity, and requires developers to make complex and difficult decisions. This entails the development of automated approaches to support various software evolution-related tasks, for example aimed at suggesting refactoring or remodularization actions. Finding a solution to these problems is intrinsically NP-hard, and exhaustive approaches are not viable due to the size and complexity of many software projects. Therefore, during recent years, several software-evolution problems have been formulated as optimization problems, and resolved with meta-heuristics.

This chapter overviews how search-based optimization techniques can support software engineers in a number of software evolution tasks. For each task, we illustrate how the problem can be encoded as a search-based optimization problem, and how meta-heuristics can be used to solve it. Where possible, we refer to some tools that can be used to deal with such tasks.

T. Mens et al. (eds.), *Evolving Software Systems*,
DOI 10.1007/978-3-642-45398-4_4,

4.1 Introduction

Software evolution activities require developers to take complex decisions, often making choices among several possible solutions. For example, let us consider a scenario where, because of maintenance and evolution tasks, the system architecture is deteriorated, resulting in lowly-cohesive and strongly-coupled modules. To mitigate such a problem, developers can reorganize the system decomposition. However, even for a relatively small system, the number of possible choices to improve the architecture can be very high, and such a number exponentially increases with the system size. Similar considerations apply to other evolution-related activities, such as selecting and performing a refactoring, or fixing a bug. In general, as other activities like testing, software evolution requires software engineers to solve problems for which finding a solution is NP-hard [312].

For such reasons, the use of search-based optimization techniques can be a very promising and effective way to deal with many software evolution activities. All that is required is to provide:

- a problem *representation*, i.e., to encode the activity through an appropriate data structure allowing the heuristics to (i) evaluate the quality of a possible problem solution, and (ii) produce new solutions;
- a way to quantitatively evaluate the quality of a given solution, often referred as *fitness* or *objective* function; and
- a set of *operators* to produce new solutions starting from existing ones.

The idea of solving software engineering problems using search-based optimization techniques has been named "Search-Based Software Engineering" [190]. The potentials and challenges of applying search-based optimization techniques to various kinds of software engineering problems have been largely discussed by Harman [365]. Also, Harman has outlined how SBSE can support various software maintenance [364] and program comprehension [366] tasks. Among others, there are two aspects that make the application of SBSE to software evolution special. First, many evolution decisions imply balancing across conflicting objectives. To this aim, it can be desirable to use multi-objective optimization, which instead of producing solutions (near) optimizing a single objective, produce a set of solutions—that, as will be explained in Section 4.2—are "Pareto-optimal" i.e., there is no solution among the found ones that is better than others with respect to all objectives. Second, many software evolution activities are highly human intensive, i.e., (i) automatic approaches must be able to account for developers' rationale, and (ii) it is hardly possible to find a completely automatic way to evaluate the quality of a solution. To this aim, it is necessary to find ways to encode rationale in the meta-heuristic fitness functions, as well as to use "interactive" optimization techniques [818] for which the fitness function is (partially) evaluated by humans.

This chapter describes the main achievements of SBSE techniques in the field of software maintenance and evolution. Specifically, we describe work related to:

- *(Re)modularization approaches*, i.e., approaches aimed at identifying and creating modules that achieve certain properties, such as high cohesion and low coupling, or aimed at reducing the application footprint.
- *Software analysis and transformation approaches*, aimed at automatically modifying source code for various specific purposes, for example to improve maintainability or fixing bugs.
- *Refactoring approaches*, aimed at automatically suggesting and applying refactoring activities, e.g., those proposed by Fowler [301]. This is a special case of code transformation, which does not alter the semantic of the source code, but improves its maintainability. Given the amount of work in this area, we discuss it in a separate section rather than together with other transformation approaches.

It is important to note that this chapter is not a systematic literature review on search-based software maintenance and evolution. There are also other pieces of work related to the use of optimization techniques in the area of software evolution. Due to space limitation, it is not possible to describe all of them. We chose to describe the aforementioned dimension because these are the one that have been investigated the most in past and recent years, also according to the SEBASE repository[1], which collects a large set of references for SBSE papers.

Instead, the chapter describes the available SBSE techniques to be used for various kinds of problems, explaining how the problem needs to be represented, how solutions can be evaluated through a fitness function, which are the operators to be used, and the meta-heuristics that work better. Also, wherever possible, the chapter points out available tools for each specific problem. In summary, the chapter aims to be a guideline for practitioners interested to apply SBSE techniques in the context of software evolution, as well as for PhD students interested to work on such a research topics, and for instructors that need to introduce such techniques in the context of software engineering or software evolution courses, especially in graduate curricula.

The chapter is organized as follows. Section 4.2 provides basic background notions about the optimization techniques used to solve software evolution problems. Section 4.3, Section 4.4, and Section 4.5 describe search-based approaches for software modularization, source code transformation, and refactoring, respectively. Section 4.6 concludes the chapter.

4.2 An Overview of Search-Based Optimization Techniques

This section provides some background on search-based optimization techniques that have been used to solve the various software maintenance problems described in this chapter. Further details can be found in the books by Goldberg [337] and by Michalewicz and Fogel [607].

For the techniques described below, the problem is encoded in a representation (named *chromosome*), an instance (solution) of which (named *genotype*) represents

[1] http://crestweb.cs.ucl.ac.uk/resources/sbse_repository/

an instance of the real world problem to be solved (referred to as *phenotype*). The quality of a solution (i.e., of a *genotype*) is evaluated by means of a *fitness function*.

2 Pseudo-code of iterated Hill Climbing (from [607]).

```
1:  t ← 0
2:  initialize best
3:  while t < MAX do
4:      local ← FALSE
5:      select a current point v_c at random
6:      evaluate v_c
7:      while not local do
8:          select all new points in the neighborhood of v_c
9:          select the point v_n from the new points with the best value of evaluation function eval
10:         if eval(v_n) is better than eval(v_c) then
11:             v_c ← v_n
12:         else
13:             local ← TRUE
14:         end if
15:     end while
16:     t ← t + 1
17:     if v_c is better than best then
18:         best ← v_c
19:     end if
20: end while
```

4.2.1 Hill Climbing

Hill Climbing (HC)—see Algorithm 2—is a "local" search method, where the search proceeds from a randomly chosen point (solution) v_c in the search space (line 5) by considering the neighbors of the point. Different families of HC exist based on how neighbors are explored. For example, stochastic HC identifies a neighbor by randomly mutating genes of the individual, i.e., by producing a slightly different solution. An iterated HC, as the one shown in Algorithm 2, iterates across all possible neighbors of a given solution (line 9). Once a fitter neighbor (v_n) is found (lines 10-11), this becomes the current point in the search space and the process is repeated. If no fitter neighbor is found (line 13), then the search terminates and a maximum has been found (by definition). To avoid local maxima, the HC algorithm is restarted multiple (t) times from a random point (lines 7-20).

Multiple ascent HC is a variant of the standard HC algorithm designed to escape from local optima. In particular, when a local optimum is reached, a set of random changes are performed in order to move away from that point and continue to explore the solution space. This procedure is repeated n times, depending on a parameter called *number of descents*, while the number of random changes applied is set through the descent depth parameter.

3 Pseudo-code of Simulated Annealing (from [607]).

1: $t \leftarrow 0$
2: $T \leftarrow T_{max}$
3: randomly select a current point v_c
4: **while** halting-criterion not met **do**
5: **while** termination-condition not met **do**
6: select a new point v_n in the neighborhood of v_c
7: **if** $eval(v_c) < eval(v_n)$ **then**
8: $v_c \leftarrow v_n$
9: **else**
10: **if** $random[0,1) < e^{\frac{eval(v_n)-eval(v_c)}{T}}$ **then**
11: $v_c \leftarrow v_n$
12: **end if**
13: **end if**
14: **end while**
15: $T \leftarrow g(T,t)$
16: $t \leftarrow t+1$
17: **end while**

4.2.2 Simulated Annealing

Simulated Annealing (SA) [604], like HC, is a local search method. As it can be seen from Algorithm 3, the algorithm is pretty similar to HC. However, one can move from a solution v_c to a neighbor v_n if (i) vc has a better fitness value than v_n (lines 7-8) or (ii) one can move from v_n to a less fit solution v_c (lines 10-11) if $p < m$, where p is a random number in the range $[0 \ldots 1]$ and $m = e^{\Delta fitness/T}$. The parameter T (temperature) regulates the likelihood to move to a less fit solution and it decreases ("cools") over time according to a function $g(T,t)$ (line 15). A typical *cooling mechanism* is given by $T = T_{max} \cdot e^{-t \cdot r}$ (T_{max} is the starting temperature (line 2), r is the *cooling factor*, t the number of iterations), and $\Delta fitness$ is the difference between the fitness values of the two neighbor individuals being compared. The effect of cooling in SA is that the probability of following an unfavorable move is reduced. This (initially) allows the search to move away from local optima in which the search might be trapped. As the simulation "cools", the search becomes more and more equivalent to a simple hill climb.

4 Pseudo-code of Particle Swarm Optimization.

```
1: for i = 1 → n do
2:     initialize the particle's position x_i
3:     set the particle's best known position p_i ← x_i
4:     initialize the particle's velocity v_i
5:     if eval(p_i) < eval(g) then
6:         g ← p_i
7:     end if
8: end for
9: while termination-condition not met do
10:    for i = 1 → n do
11:        update particle's velocity v_i
12:        x_i ← x_i + v_i
13:        if eval(x_i) < eval(p_i) then
14:            p_i ← x_i
15:            if eval(p_i) < eval(g) then
16:                g ← p_i
17:            end if
18:        end if
19:    end for
20: end while
```

4.2.3 Particle Swarm Optimization

Particle Swarm Optimization (PSO) was introduced by Kennedy and Ebhart in 1995 [456]. The basic concept of the algorithm is to create a swarm of particles which move in the space around them (the problem space) searching for their goal, the place which best suits their needs given by a fitness function. A nature analogy with birds is the following: a bird flock flies in its environment looking for the best place to rest. The best place can be a combination of characteristics like space for all the flock, food access, water access or any other relevant characteristics.

PSO is described in the pseudocode of Algorithm 4. First, an initial population (named *swarm*) of n random solutions (named *particles*) is created. Every particle in the swarm is described by its position and velocity. A particle position represents a possible solution to the optimization problem, and velocity represents the search distances and directions that guide particle flying. At each particle is assigned an initial position x_i (line 2), which is also the best known position p_i known so far (line 3), and an initial velocity v_i (line 4). Then, each particle flies in the problem space with a velocity that is regularly adjusted according to the composite flying experience of the particle and some, or all, the other particles (line 12). Given the new velocity, the position is updated accordingly (line 12). The fitness of each particle (that depends on the position x_i) is evaluated and, if needed, the best position p_i is updated (lines 13-14). Similarly, the overall best position among all particles g is updated (lines 15-16). The process of updating particles' velocity and position (lines 9-20) is repeated until a termination criterion (e.g., maximum number of iterations) is met.

5 Pseudo-code of a Genetic Algorithm (from [607]).

```
1: t ← 0
2: initialize a population P(t) of n individuals
3: evaluate P(t)
4: while termination-condition not met do
5:     t ← t + 1
6:     select a subset P'(t − 1) of individuals from P(t − 1) to reproduce
7:     apply crossover to P'(t − 1) and introduce offspring in P(t)
8:     mutate individuals in P(t)
9:     evaluate P(t)
10: end while
```

4.2.4 Genetic Algorithms

Genetic Algorithms (GAs) [337] belong to the family of evolutionary algorithms that, inspired by the theory of natural evolution, simulate the evolution of species emphasizing the law of survival of the strongest to solve, or approximately solve, optimization problems. Thus, these algorithms create consecutive populations of individuals, considered as feasible solutions for a given problem (phenotype) to search for a solution which gives the best approximation of the optimum for the problem under investigation. To this end, a fitness function is used to evaluate the goodness (i.e., fitness) of the solutions represented by the individuals, and genetic operators based on selection and reproduction are employed to create new populations (i.e., generations).

As shown in Algorithm 5, the elementary evolutionary process of these algorithms is composed of the following steps:

1. a random initial population $P(0)$ is generated (line 1) and a fitness function is used to assign a fitness value to each individual (line 2);
2. given t the current generation (line 3), some individuals of a population $P'(t-1)$ are selected to form the parents (line 6) and new individuals are created by applying genetic operators (i.e., crossover and mutation). The crossover operator (line 7) combines two individuals (i.e., parents) to form one or two new individuals (i.e., offspring), while the mutation operator (line 8) is used to randomly modify an individual. Then, to determine the individual that will survive among the offspring and their parents a survivor selection is applied according to the individuals' fitness values (line 9);
3. step 2 is repeated until stopping criteria hold.

When designing a GA, the crossover and mutation operators play a crucial role. Different crossover operators can be used. Among the most used, there are:

- *One-point crossover*. A point in the chromosome of the two parents is selected, and all the genes beyond that point in either chromosome are swapped between the two parents.

- *Two-point crossover*. Two points in the chromosome of the two parents are selected, and everything between the two points is swapped between the parents, generating the offspring.
- *Uniform crossover*. A fixed mixing ratio between two parents is used. Unlike one- and two-point crossover, the uniform crossover enables the parent chromosomes to contribute the gene level rather than the segment level.

As for the mutation, the selection of the operator depends on the representation of the solution. For integer and float genes, a widely-used operator is the uniform mutation. Using such an operator, the value of a chosen gene is replaced with a uniform random value selected between the user-specified upper and lower bounds for that gene. If the solution is represented by a binary string a common mutation operator is the *bit flip*, where the bit of the chosen gene is inverted (i.e., if the value is 1, it is changed to 0 and vice versa).

It is worth noting that during each generation these operators are applied with a certain probability, named *crossover rate* and *mutation rate*. In addition, at each generation parents have to be selected for crossover and mutation. Thus, also the selection operator plays an important role. The most used selectors are *roulette wheel*, in which each individual's probability of selection is directly proportional to its relative fitness, and *tournament selection*, where small subsets of the population are selected randomly (a tournament) and the most fit member of the subset is selected for the next generation.

Finally, the stopping criterion for the evolutionary process is usually based on a maximum number of generations. This stopping criterion can be combined with other criteria to reduce the computation time. For example, the search process can be stopped when there is no improvement in the fitness value for a given number of generations.

A variant of GAs is Genetic Programming (GP) [477], where the aim is to generate programs (that can be also prediction models, or expressions, etc.) having certain properties. The representation is often (but not necessary) a program Abstract Syntax Tree (AST) and the fitness is evaluated by executing the program.

4.2.5 Multi-Objective Optimization

An optimization problem can have one objective, but also more than one objective (multi-objective optimization). In a multi-objective optimization problem, a solution is described in terms of a decision vector $(x_1, x_2, ..., x_n)$ in the decision space X. Then the fitness function $f : X \rightarrow Y$ evaluates the quality of a specific solution by assigning it an objective vector $(y_1, y_2, ..., y_k)$ in the objective space Y, where k is the number of objectives.

Comparing solutions in multi-objective optimization is not as trivial as in the case of single-objective optimization problems. Specifically, in multi-objective optimization problems it is necessary to exploit the concept of Pareto dominance: an objective vector y_1 is said to dominate another objective vector y_2 ($y_1 \succ y_2$) if no

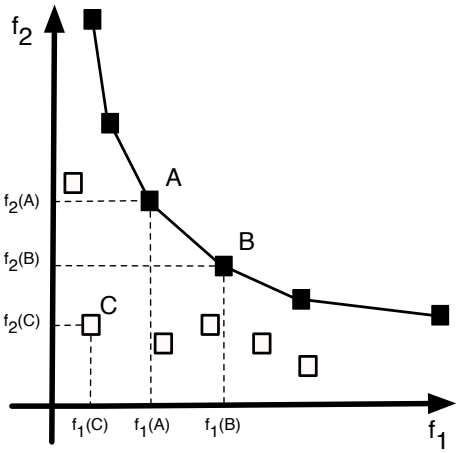

Fig. 4.1: Pareto dominance: A and B are non-dominating solutions, while C is dominated by both A and B.

component of y_1 is smaller than the corresponding component of y_2 and at least one component is greater. The Pareto dominance allows to say that a solution x_1 is better than another solution x_2, i.e., x_1 dominates x_2 ($x_1 \succ x_2$), if $f(x_1)$ dominates $f(x_2)$. For example, in Figure 4.1, point C is dominated by A and B since $f_1(A) > f_1(C)$, $f_2(A) > f_2(C)$, $f_1(B) > f_1(C)$, and $f_2(B) > f_2(C)$. Instead, A and B represent non-dominating solutions: if we consider A, there is at least another solution (B in our case) such that $f_1(B) > f_1(A)$. Similarly, if we consider B, there is at least another solution (A) such that $f_2(A) > f_2(B)$. It is worth noting that using such a definition it is possible to define a *set* of optimal solutions, i.e., solutions not dominated by any other solution. Such solutions may be mapped to different objective vectors. In other words, there may exist several optimal objective vectors representing different trade-offs between the objectives. This set of optimal solutions is generally denoted as the *Pareto set* $X^* \subseteq X$, while the fitness values achieved by such solutions represent the Pareto front $Y^* \subseteq Y$.

In principle, a multi-objective optimization problem can be reduced to a single-objective optimization problem. For instance, the different objectives can be aggregated into a single one. However, the analysis of the Pareto front can help the decision maker in (i) selecting the most suitable solution, i.e., the solution that provides the best compromise in a particular scenario; and (ii) analyzing the trade-off provided by each solution.

The concept of Pareto dominance is also used to rank solutions and to apply selection strategies based on non-domination ranks. Generally, such algorithms are elitist: the best solutions—i.e., the non-dominated solutions—are either kept in the population itself or are stored separately to be reused. In the first case, they participate to the reproduction process. However, the number of non-dominated solutions

might greatly increase with the number of objectives, which limits the number of places reserved for new individuals. Therefore, such algorithms generally use a specific operator to preserve diversity. The elitist Non-dominated Sorting Genetic Algorithm (NSGA-II) [227] is certainly one of the most popular algorithms belonging to this category.

A naive multi-objective optimization algorithm would require $\mathcal{O}(M\,N)$ comparisons to identify each solution of the first nondominated front in a population of size N and with M objectives, and a total of $\mathcal{O}(M\,N^2)$ comparisons to build first nondominated front. This is because each solution needs to be compared with all other solutions. Since the above step has to be repeated for all possible fronts—which can be at most N, if each front is composed of one solution—the overall complexity for building all fronts is $\mathcal{O}(M\,N^3)$.

NSGA-II uses a faster algorithm for nondominated sorting, which has a complexity $\mathcal{O}(M\,N^2)$:

1. for each solution p in the population, the algorithm finds the set of solutions S_p dominated by p and the number of solutions n_p that dominate p. The set of solutions with $n_p = 0$ are placed in the set first front F_1.
2. $\forall p \in F_1$, solutions $q \in S_p$ are visited and, if $n_q - 1 = 0$, then solution q is placed in the second front F_2. This step is repeated $\forall p \in F_1$ to generate F_3, etc.

To compare solutions, NSGA-II uses the "crowded comparison operator". That is, given two solutions x_1 and x_2, x_1 is preferred over x_2 if it belongs to a different (better) front. Otherwise, if x_1 and x_2 belong to the same front, the solution located in the less crowded region of the front is preferred.

Then, NSGA-II produces the generation $t+1$ from generation t as follows:

1. generating the child population Q_t from the parent population P_t using the binary tournament selection and the crossover and mutation operators defined for the specific problem;
2. creating a set of $2N$ solutions $R_t \equiv P_t \bigcup Q_t$;
3. sorting R_t using the nondomination mechanism above described, and forming the new population P_{t+1} by selecting the N best solutions using the crowded comparison operator.

4.3 Search-based Software Modularization

Software (re)modularization is probably one of the software evolution tasks where SBSE techniques have been applied most. Given a set of artifacts—for example classes or source code files—the aim of software modularization is to identify groups of artifacts that, according to given criteria, are cohesive enough and exhibit low coupling with other groups. During software evolution, this can be useful to support system restructuring, but also—without restructuring the system—to support program comprehension by highlighting groups of cohesive components.

4.3.1 The Bunch approach for software modularization

Bunch [610] is a software modularization tool that relies on search-based optimization techniques.

Problem definition. Generally speaking, software modularization can be seen as a graph partitioning problem, whose solution is known to be NP-hard [312]. In the past, various authors have tackled this problem with clustering techniques [559, 701, 923]. In the following, we illustrate the problem as it has been formalized and solved—using search-based optimization techniques—by Mitchell and Mancoridis in their *Bunch* tool. Bunch operates on a system representation called Module Dependency Graph (*MDG*), a graph $G = (V, E)$ where nodes V are system artifacts and edges E are relations between such artifacts (e.g., function or method calls). The goal of modularization is to partition G into n clusters $Pi_G = \{G_1, G_2, \ldots, G_n\}$. Each cluster G_i is composed of a set of (non-overlapping) artifacts from V, i.e., $G_i \cap G_j = \emptyset \; \forall i, j \in 1 \ldots n$.

Solution representation. To find solutions of the modularization problem using search-based heuristics, the problem needs to be encoded in a chromosome. Given a software system composed of n software components (e.g., classes), the chromosome is represented as a n-sized integer array, where the value $0 < v \leq n$ of the i^{th} element indicates the cluster to which the i^{th} component is assigned. A solution with the same value (whatever it is) for all elements means that all software components are placed in the same cluster, while a solution with all possible values (from 1 to n) means that each cluster is composed of one component only.

Fitness function. Starting from the *MDG* (weighted or unweighted), the output of a software module clustering algorithm is represented by a partition of this graph. A good partition of an *MDG* should be composed of clusters of nodes having (i) high dependencies among nodes belonging to the same cluster (i.e., high cohesion), and (ii) few dependencies among nodes belonging to different clusters (i.e., low coupling). To capture these two desirable properties of the system decompositions (and thus, to evaluate the modularizations generated by Bunch), Mancoridis et al. [555] define the Modularization Quality (*MQ*) metric as in Equation 4.1, where k is the number of modules, A_i is the Intra-Connectivity (i.e., cohesion) of the i^{th} cluster and $E_{i,j}$ is the Inter-Connectivity (i.e., coupling) between the i^{th} and the j^{th} clusters.

$$MQ = \begin{cases} \left(\frac{1}{k}\sum_{i=1}^{k} A_i\right) - \left(\frac{1}{\frac{k(k-1)}{2}}\sum_{i,j=1}^{k} E_{i,j}\right) & \text{if } k > 1 \\ A_1 & \text{if } k = 1 \end{cases} \tag{4.1}$$

The Intra-Connectivity of a cluster i is given by Equation 4.2, where μ_i is the number of intra-cluster edges, N_i is the number of nodes of cluster i, and consequently N_i^2 is the maximum number of such intra edges.

$$\frac{\mu_i}{N_i^2} \tag{4.2}$$

The Inter-Connectivity between two clusters i and j is given by Equation 4.3, where $\varepsilon_{i,j}$ is the the number of edges between i and j, while N_i and N_j are the number of nodes in i and j respectively.

$$\frac{\varepsilon_{i,j}}{2 \cdot N_i \cdot N_j} \tag{4.3}$$

Figure 4.2 shows an example of *MDG* and of its representation. For the *MDG* in Figure 4.2-a, the *MQ* is equal to $1/2 \cdot (2/3^2 + 1/2^2) - (1/((2 \cdot 1)/2)) \cdot (2/(2 \cdot 3 \cdot 2)) = 0.07$. Moving Component $C2$ to Module 2 (Figure 4.2-b), the *MQ* becomes $1/2 \cdot (1/2^2 + 3/3^2) - (1/((2 \cdot 1)/2)) \cdot 1/(2 \cdot 3 \cdot 2) = 0.2$. Hence, as expected, the modularization quality increases.

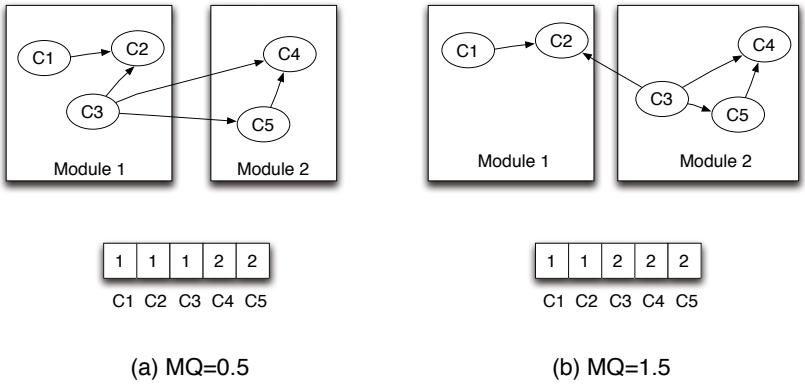

Fig. 4.2: Module Dependency Graph (MDG) used by Bunch [610], its chromosome representation, and resulting *MQ* value.

Supported search-based techniques and change operators. Bunch allows to solve the software modularization problem using different search-based optimization heuristics, namely HC, SA, and GAs. In principle Bunch also allows to solve the problem exhaustively, however—as reported by Mitchell and Mancoridis [610]—the number of possible partitions exponentially increases with the number of nodes.

The HC approach works as follows. It starts with a randomly generated modularization i.e., a chromosome filled with random numbers varying between 1 and n. Then, neighbor solutions are created by moving one artifact from a cluster to another, i.e., by randomly changing the value in a gene. After that, the fitness function—i.e., the *MQ*—of the new produced solution is evaluated, and if its fitness is better than the previous one, then the solution is accepted and the evolution continues.

The above approach has two weaknesses. The first one is that HC algorithms can converge to local optima; the second one is that the algorithm may tend to create isolated clusters, i.e., clusters composed of one artifact only. The local optima problem is mitigated through multiple restarts of the HC, using initial solutions belonging to

a population of randomly generated ones, and specifically from a subset of it having the highest *MQ*. An alternative is to use SA instead of HC. Since SA does not always proceed towards (locally) better solutions, this can mitigate the local optima problem.

Also, the problem can be solved using GAs, which evolve multiple solutions—i.e., a population of individuals—rather than single one. The GAs-based approach of Bunch [610]—named *Gadget*—has been described by Doval et al. [260]. A GA evolves the population using a *selection operator*, which selects individuals to reproduce based on the fitness function, a one-point *crossover* operator, and a *mutation* operator which is the same used for HC.

The problem of isolated clusters is dealt by assigning such isolated clusters to another, randomly chosen, cluster.

Empirical evaluation. There are different ways to evaluate the quality of solutions obtained using a modularization technique. When a reference (ideal) solution is available for a given system, it can be compared with the solution produced by the modularization technique. Such a comparison can be made using the MoJoFM eFfectiveness Measure (MoJoFM) [914], defined in Equation 4.4, where $mno(A,B)$ is the minimum number of *Move* or *Join* operations one needs to perform in order to transform the partition A into B, and $max(mno(\forall A,B)$ is the maximum possible distance of any partition A from the gold standard partition B.

$$MoJoFM(A,B) = 100 - \left(\frac{mno(A,B)}{max(mno(\forall A,B))} \times 100 \right) \tag{4.4}$$

When no reference solution is available, one can qualitatively evaluate a modularization solution (e.g., by relying on experts), and also evaluate the *stability* of the technique being used. A clustering technique is stable if it produces similar results over multiple runs. From a qualitative point of view, Mitchell and Mancoridis [610] applied Bunch on a 50 KLOC C++ program that implements a file system service. Bunch created two main clusters, related to two different file systems being accessed, and this was confirmed by the system expert. Also, Bunch created a hierarchical decomposition, which allowed experts to review the proposed modularization at different levels of granularity. To evaluate the clustering stability, Mitchell and Mancoridis [610] used (i) the *EdgeSim* similarity measurement, that normalizes the number of intra and inter cluster edges that are in agreement between two different modularizations, and (ii) the *MeCl* similarity that determines the distance between two modularizations. A study performed on the Java Swing library reported an average *EdgeSim* of 93.1% and an average *MeCl* of 96.5%.

4.3.2 Multi-Objective Modularization

The modularization approach described in Section 4.3.1 produces solutions that are (near) optimal with respect to a single objective, i.e., the *MQ*. To this extent, *MQ* achieves a compromise between having a good cohesion and low coupling.

The alternative to single-objective optimization is to use multi-objective optimization where—as explained in Section 4.2.5—the found solutions are Pareto-optimal, i.e., each solution is better than another with respect to a particular objective, while it may not be better with respect to other objectives.

Fitness function. Praditwong et al. [701] have proposed an approach for multi-objective optimization, where the considered objectives are the following:

- maximizing the number of intra-module edges: that is, a high number of intra-module edges denotes a high cohesion;
- minimizing the number of inter-module edges: that is, a low number of inter module edges denotes a low coupling;
- maximizing the number of clusters, which generally favors high cohesion;
- minimizing the number of isolated clusters: such objective is an alternative to the repairing solution described in Section 4.3.1 to handle isolated clusters;
- maximizing MQ which, as explained in Section 4.3.1 favors solutions achieving a compromise between cohesion and coupling.

Supported search-based techniques and change operators. Praditwong et al. [701] have implemented the multi-objective modularization using the same operators of Mitchell et al. [260], however using a NSGA-II [226] multi-objective GA instead of a simple GA.

Empirical evaluation. Praditwong et al. [701] have evaluated their multi-objective approach to modularize 17 MDG extracted from various C programs (e.g., Unix utilities such as *bison*, *ispell*, *lynx*, *ncurses*, and *rcs*), and compared it with the single-objective, hill-climbing based approach of Mitchell and Mancoridis [610]. Other than exhibiting the advantages outlined above, i.e., the capability of the software engineer to select solutions that are particularly good for specific objectives than for others, the multi-objective GA was also able to outperform single-objective modularization for each specific modularization objective (e.g., MQ increase between 15% and 50% for 10 out of 17 programs, and decrease within 5% for the other 7). In summary, besides the usual advantages of multi-objectives, it can be preferred to the single-objective alternative also for what concerns the quality of the obtained solutions [701]. However, the drawback is that multi-objective optimization is more expensive from a computational point-of-view. That is, the number of evaluations required is two orders of magnitude higher.

4.3.3 Achieving different software modularization goals

The above described approaches deal with software modularization from a structural point of view, and with the aim of obtaining cohesive and decoupled clusters. The existing literature also reports approaches where search-based remodularization was used for different purposes.

Di Penta et al. [247] and Antoniol et al. [37] deal with remodularizing software libraries with the aim of minimizing the footprint of an application in the program

memory. This is particularly useful when porting applications towards devices with a limited memory. Years ago, this was particularly true for many mobile devices; nowadays most mobile devices (tablets and smartphones) have enough memory. However, memory occupation is still a concern for some specific devices such as embedded systems or active sensors.

To deal with the software miniaturization problems, Di Penta *et al.* and Antoniol et al. [37] start from a graph highlighting dependencies between applications and libraries, and between objects composing libraries. Given this graph, the goal to be achieved is to minimize the footprint of applications, considering the set of libraries they should be linked to. Since libraries can be partitioned in different ways such that the overall size of the libraries used by each application is minimized, this is still a modularization problem that can be solved using search-based optimization techniques. The miniaturization problem is then solved by using a GA similar to the one used by Doval et al. [260] for remodularization purposes. However, instead of using the *MQ* as fitness function, a mono-objective one (to be minimized) consisting of a weighted sum of the four factors keeping into account: (i) the total number (or size) of objects linked to each application, (ii) the number of inter-library dependencies (to avoid linking a library every time another is linked), (iii) the difference with the initial libraries (to avoid scrambling the libraries completely), and (iv) feedbacks provided by experts/original developers.

Di Penta et al. [247] and Antoniol et al. [37] applied their approach on various C programs, such as Grass, QT, MySQL, and Samba. The application footprint size was reduced of over 60% for MySQL and Samba, and between 5% and 25% for Grass and QT

A different miniaturization approach has been proposed by Ali et al. [17]. In their work, they aim at determining the set of features to be included in an application when porting it towards a mobile device with the aim of (i) maximizing customers' satisfaction and (ii) keeping the devices' battery consumption low. For each feature, they also measure the estimated battery consumption, using a framework by Binder and Hulaas [107], based on bytecode analysis. Finally, they use a NSGA-II [226] multi-objective optimization to determine the set of features to include in the ported application. Ali *et al.* [17] experimented their approach to miniaturize an email client (Pooka) and an instant messenger (SIP). The minuaturization was experimented by considering some user requirements collected through a survey, and some hypothetical constraints in terms of user satisfaction and consumption. Compared with a manual minuaturization, the proposed approach allowed to save about 77% of the effort.

4.3.4 Putting the developer in the loop: interactive software modularization

The approaches for software modularization described above have the advantage of being completely automatic, i.e., they produce a possible modularization without requiring manual intervention.

While automatic re-modularization approaches proved to be very effective in increasing cohesiveness and reducing coupling of software modules, they do not take into account developers' knowledge when deciding to group together (or not) certain components. For example, a developer may decide to place a function in a given module even if, in its current implementation, the function does not communicate a lot with other functions in the same module. This is because the developer may be aware that, in future releases, such a function will strongly interact with the rest of the module. Similarly, a developer may decide that two functions must be placed in two different modules even if they communicate. This is because the two functions have different responsibilities and are used to manage semantically different parts of the system.

To deal with this problem, different authors have proposed methods to incorporate developers' feedback in search-based remodularization algorithms.

Hall et al. [360] proposed a supervised remodularization approach, named SUMO (**Su**pervised Re**mo**dularization), that integrates existing modularization approches—such as Bunch [610]—with corrections provided by the software engineer. The idea of the approach is the following:

1. First, a solution of the modularization problem is created using automatic modularization (Bunch, for example).
2. After that, the user provides corrections through a user interface. Such corrections consist of two sets of relations, Rel^+, defined as pairs of artifacts that should belong to the same cluster (i.e., go together), and Rel^-, defined as pairs of artifacts that should not go together.
3. After that, a constraint satisfaction approach is used to modify the initially produced clusters with the aim of satisfying constraints expressed by sets of relations Rel^+ and Rel^-. Steps 2 and 3 are repeated until the software engineer finds no further corrections.

Bavota et al. [82] proposed the use of Interactive GA [818] (IGA) to solve the modularization problem, considering it as both single-objective and multi-objective optimization problem. The problem is encoded as done in Gadget [260]. The single-objective GA uses as fitness function the *MQ* metric, while the multi-objective GA considers the five different objectives of the approach by Praditawong et al. [701] described in Section 4.3.2.

The basic idea of the IGA is to periodically add a constraint to the GA such that some specific components shall be put in a given cluster among those created so far. Thus, the IGA evolves exactly as the non-interactive GA. Every *nGens* generations, the best individual is selected and shown to the software engineer. After that, the

software engineer analyzes the proposed solutions and provides feedback, indicating that certain components must be placed in a specific cluster.

In principle, the IGA can ask feedback for every pair of components. However, this would be too much work for the software engineer. To limit the amount of feedback, the Algorithm 6—takes the best solution produced by the GA, randomly selects two components (from the same cluster or from different clusters), and then asks the software engineer whether, in the new solutions to be generated, such components must be placed in the same cluster (i.e., stay together) or whether they should be kept separated. In total, every *nGens* generations the software engineer is asked to provide feedback about a number *nFeedback* of component pairs from the best solution (in terms of MQ) contained in the current population.

6 Pseudocode of the Interactive GA for software modularization.

```
1: for i = 1 ... nInteractions do
2:     Evolve GA for nGens generations
3:     Select the solution having the highest MQ
4:     for j = 1 ... nFeedback do
5:         Randomly select two components c_i and c_j
6:         Ask the developer whether c_i and c_j must go together or kept separate
7:     end for
8:     Repair the solution to meet the feedback
9:     Create a new GA population using the repaired solution as starting point
10: end for
11: Continue (non-interactive) GA evolution until it converges or it reaches maxGens
```

After feedback is provided, the solution is repaired by enforcing the constraints, e.g., by randomly moving one of c_i and c_j away if the constraint tells that they shall be kept separated. After all *nFeedback* have been provided, a new population is created by randomly mutating such a repaired solution. Then, the GA starts again.

During the GA evolution, to ensure constraints specified by the software engineers are satisfied, a penalty factor is added to the fitness function (as proposed by Coello Coello [195]), to penalize solutions violating the constraints imposed by the developers. Given $CS \equiv cs_1, \ldots cs_m$ the set of feedback collected by the users, the fitness $F(s)$ for a solution s is computed by Equation 4.5, where $k > 0$ is an integer constant weighting the importance of the feedback penalty, and $vcs_{i,s}$ is equal to one if solution s violates cs_i, zero otherwise. After *nInteractions* have been performed, the GA continues its evolution in a non-interactive way until it reaches stability or the maximum number of generations.

$$F(s) = \frac{MQ(s)}{1 + k \cdot \sum_{i=1}^{m} vcs_{i,s}} \tag{4.5}$$

A variant of the interactive GA described above specifically aims at avoiding isolated clusters, by asking the developers where the isolated component must be placed. For non-isolated clusters, the developer is asked to specify for each pair of components whether they must stay together or not.

Finally, the multi-objective variants of IGA are quite similar to the single-objective ones. Also in this case, two variants have been proposed, one—referred to as R-IMGA (Random Interactive Modularization Genetic Algorithm)—where feedback is provided on randomly selected pairs of components, and one—referred to as IC-IMGA (Isolated Clusters Interactive Modularization Genetic Algorithm)—where feedback is provided on components belonging to isolated (or smallest) clusters.

In summary, interactive approaches to software modularization help software engineers to incorporate their rationale in automatic modularization approaches. The challenge of such approaches, however, is to limit the amount of feedback the software engineers have to provide. On the one hand, a limited amount of feedback can result in a modularization that is scarcely usable. On the other hand, too much feedback may become expensive and make the semi-automatic approach no longer worthwhile.

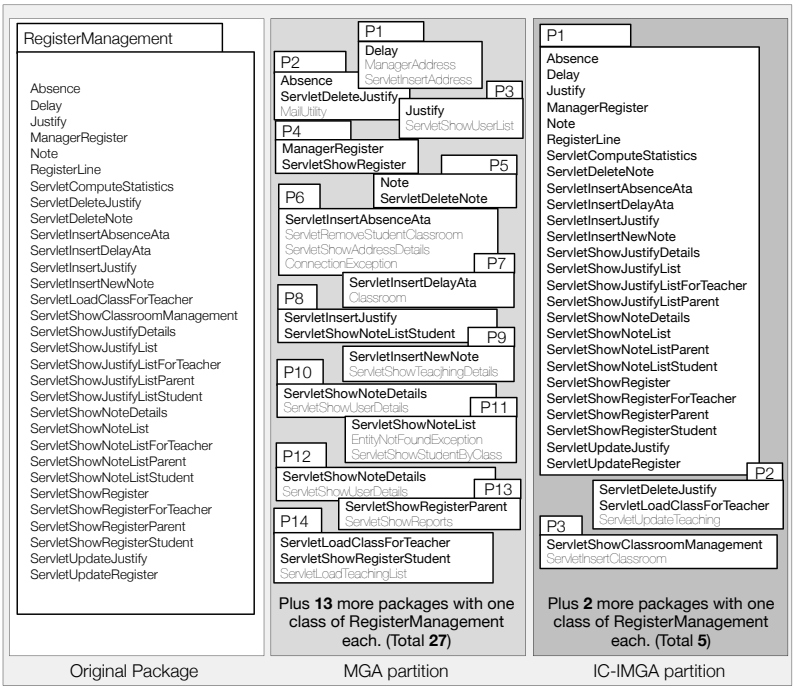

Fig. 4.3: MGA *vs* IC-IMGA in reconstructing the *RegisterManagement* package of SMOS [82].

In the evaluation reported in the paper by Bavota et al. [82], the authors compare the different variants of IGAs with their non-interactive counterparts in the context of software re-modularization. The experimentation has been carried out on two software systems, namely GESA and SMOS, by comparing the ability of

GAs and IGAs to reach a fair trade-off between the optimization of some quality metrics (that is the main objective of GAs applied to software re-modularization) and the closeness of the proposed partitions to an authoritative one (and thus, their meaningfulness). The achieved results show that the IGAs are able to propose re-modularizations (i) more meaningful from a developer's point-of-view, and (ii) not worse, and often even better in terms of modularization quality, with respect to those proposed by the non-interactive GAs.

To understand in a practical way what is the difference between the performances of interactive and non-interactive GAs, Figure 4.3 shows an example extracted from the re-modularization of the SMOS software system. The figure is organized in three parts. The first part (left side) shows how the subsystem `RegisterManagement` appears in the original package decomposition (i.e., which classes it contains) made by the SMOS's developers. This subsystem groups together all the classes in charge to manage information related to the scholar register (e.g., the students' delay, justifications for their absences and so on). The second part (middle) reports the decomposition of the classes contained in `RegisterManagement` proposed by the MGA (Modularization Genetic Algorithm). Note that some classes not belonging to the `RegisterManagement` were mixed to the original set of classes. These classes are reported in light gray. Finally, the third part (right side) shows the decomposition of the classes contained in `RegisterManagement` proposed by the IC-IMGA. Also in this case, classes not belonging to the original `RegisterManagement` package are reported in light gray. As we can notice, the original package decomposition groups 31 classes in the `RegisterManagement` package. When applying MGA, these 31 classes are spread into 27 packages, 13 of which are singleton packages. As for the remaining 14 they usually contain some classes of the `RegisterManagement` package mixed with other classes coming from different packages (light gray in Figure 4.3). The solution provided by IC-IMGA is quite different. In fact, IC-IMGA spreads the 31 classes in only 5 packages. Moreover, it groups together in one package 26 out of the 31 classes originally belonging to the `RegisterManagement` package. It is striking how much the partition proposed by IC-IMGA is closer to the original one resulting in a higher MoJoFM achieved by IC-IMGA with respect to MGA and thus, a more meaningful partitioning from a developer's point of view.

4.4 Software Analysis and Transformation Approaches

In this section we describe how to instantiate a search-based approach to automatically modify source code (and models) for various specific purposes, for example improving maintainability or fixing bugs.

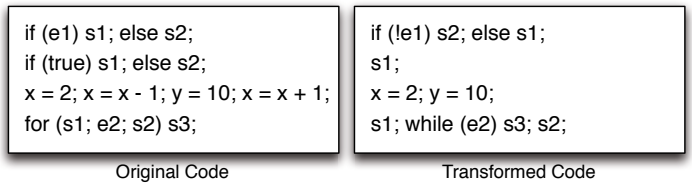

Fig. 4.4: Example of program transformation [283].

4.4.1 Program transformation

Program transformation can be described as the act of changing one program to another. Such a transformation is achieved by converting each construct in some programming language into a different form, in the same language or, possibly, in a different language. Further details about software transformation—and in particular model transformation—can be found in a paper by Mens and Van Gorp [597].

Program transformation has been recognized as an important activity to facilitate the evolution of software systems. The hypothesis is that the original source code can be progressively transformed into alternative forms. The output of the process is a program possibly easier to be understood and maintained, or without bugs (according to a given test suite).

Figure 4.4 reports an example of transformation for an original program to another program. In this case the program semantics is preserved and the transformation aims at improving the comprehensibility of the program. Such a transformation is achieved by applying a set of transformation axioms. The choice of the axioms to be applied is guided by the final goal of program transformation, that is making the code more comprehensible.

Problem definition. The program transformation problem can be considered as an optimization problem, where an optimal sequence of transformation axioms (*transformation tactic*) is required in order to make an input program easier to comprehend [283]. Transformations can be applied at different points—e.g., nodes of the Control Flow Graph (CFG)—of the program. The set of transformation rules and their corresponding application point is therefore large. In addition, many rules may need to be applied to achieve an effective overall program transformation tactic, and each will have to be applied in the correct order to achieve the desired result. All these considerations suggest that the problem is hard to solve and it represents a rich soil for search-based approaches. Specifically, search-based approaches can be used to identify a sub-optimal sequence of transformations in a search space that contains all the possible allowable transformation rules.

Solution representation. A solution is represented by a sequence of transformations that have to be applied on an input program. Fatiregun et al. [283] use a very simple representation, where each solution has a fixed sequence length of 20 possible transformations. Thus, each solution contains the identifier of the specific

transformation in the considered catalogue. The FermaT[2] [908] transformation tool is used to apply the transformation encoded in each solution. FermaT has a number of built-in transformations that could be applied directly to any point within the program. Examples of such transformations are: `@merge-right`, that merges a selected statement into the statement that follows it, or `@remove-redundant-vars`, that will remove any redundant variables in the source program.

Fitness function. A program is transformed in order to achieve a specific goal, e.g., reduce its complexity. In this case, the fitness function could be based on existing metrics for measuring software complexity, such as Lines of Code (LoC) or cyclomatic complexity. Fatiregun *et al.* [283] measure the fitness of a potential solution as the difference in the lines of code between the source program and the new transformed program created by that particular sequence of transformations. Specifically, they first compute the LOC of the original program. Then, they apply on the original program the sequence of transformations identified by the search-based approach obtaining a new version of the program. Using such a fitness function, an optimum solution would be the sequence of transformations that results in an equivalent program with the fewest possible number of statements.

Supported search-based techniques and change operators. The search-based techniques used to support program transformation are HC and GA [283]. In the HC implementation the neighbor has been defined as the mutation of a single gene from the original sequence leaving the rest unchanged. As for GA, a single point crossover has been used. Note that the solution proposed by Fatiregun et al. makes the implementation of crossover and mutation operators quite simple. Specifically, for the crossover a random point is chosen and genes are then swapped, creating two new children. As for the mutation, a single gene is chosen and it is changed arbitrarily.

Empirical evaluation. The effectiveness (measured in terms of size reduction) of search-based approaches for program transformation has been only preliminary evaluated on small synthetic program transformation problems [283]. Specifically, the transformations achieved with both GA and HC were compared with those returned by a purely random search of the search space. The comparison was based on two different aspects: the most desirable sequence of transformations that the algorithm finds and the number of fitness evaluations that it takes to arrive at that solution. Results indicated that GA outperforms both the random search and the HC as the source program size increases. In addition, the random search outperforms HC in some specific case. Unfortunately, until now only a preliminary analysis of the benefits provided by search-based approaches for program transformation has been performed. Studies with users are highly desirable to evaluate to what extent the achieved transformation are meaningful for developers.

[2] http://www.cse.dmu.ac.uk/ mward/fermat.html

4.4.2 Automatic Software Repair

Repairing defects in software systems is usually costly and time-consuming, due to the amount of software defects in a software system. Because of this very often—due to the lack of available resources—software projects are released with both known and unknown defects [521]. As example, in 2005, a Mozilla's developer claimed that, "*everyday, almost 300 bugs appear [...] far too much for only the Mozilla programmers to handle*" [41]. These considerations have prompted researchers in the definition of methods for automatic repair of defects. Specifically, Le Goues et al. [502] have formulated such a problem as an optimization problem and have proposed Genetic Program Repair (GenProg), a technique that uses existing test cases to automatically generate repairs for bugs in software systems.

Problem definition. The basic idea is inspired by the definition of a repair, that is a patch consisting of one or more code changes that, when applied to a program, cause it to pass a set of test cases [502]. Thus, given a buggy source code component and a test suite (where there is at least one test case that did not pass due to the presence of the bug), GP is used to automatically mutate the buggy code aiming at pass all the tests in a given test suite.

Solution representation. In GenProg each solution is represented by an abstract syntax tree that includes all the statements (i.e., assignments, function calls, conditionals, blocks, and looping constructs) in the program. In addition, to each solution is associated a weighted path computed by executing all the test cases in the given test suite. Specifically, a statement in the AST is weighted with 1 if the statement is covered by a test case that does not pass, 0 otherwise. The weighted path is used to localize buggy statements to be mutated (i.e., statements covered by test cases that do not pass). In addition, the weighted path is used to avoid mutating correct statements (i.e., statements covered by test cases that pass). The conjecture is that a program that contains an error in one area of source code likely implements the correct behavior elsewhere [273].

Fitness function. The fitness function is used to evaluate the goodness of a program variant obtained by GP. A variant that does not compile has fitness zero. The other variants are evaluated taking into account whether or not the variant passes test cases. Specifically, the fitness function for a generic variant v is a weighted sum:

$$f(v) = W_{Pos_{test}} \cdot |\{t \in Pos_{test} : v \text{ passed } t\}| + W_{Neg_{test}} \cdot |\{t \in Neg_{test} : v \text{ passed } t\}| \tag{4.6}$$

where Pos_{test} is the set of positive test cases that encode functionality that cannot be sacrificed and Neg_{test} is the set of negative test cases that encode the fault to be repaired. The weights $W_{Pos_{test}}$ and $W_{Neg_{test}}$ have been empirically determined using a trial and error process. Specifically, the best results have been achieved setting $W_{Pos_{test}} = 1$ and $W_{Neg_{test}} = 10$.

Selection and genetic operators. As for the selection operator, in GenProg two different operators have been implemented, i.e., roulette wheel and tournament selection. The results achieved in a case study indicated that the two operators provided almost the same performances. Regarding the crossover operator, in GenProg

a single point crossover is implemented. It is worth noting that only statements along the weighted paths are crossed over. Finally, the mutation operator is used to mutate a given variant. With such an operator is possible to:

- *Insert new statement*. Another statement is inserted after the statement selected for mutation. To reduce the complexity, GenProg uses only statements from the program itself to repair errors and does not invent new code. Thus, the new statement is randomly selected from anywhere in the program (not just along the weighted path). In addition, the statement's weight does not influence the probability that it is selected as a candidate repair.
- *Swap two statements*. The selected statement is swapped with another statement randomly selected, following the same approach used for inserting a new statement.
- *Delete a statement*. The selected statement is transformed into an empty block statement. This means that a deleted statement may therefore be modified in a later mutation operation.

In all cases, the weight of the mutated statement does not change.

Refinement of the solution. The output of GP is a variant of the original program that passes all the test cases. However, due to the randomness of GP, the obtained solution contains more changes than what necessary to repair the program. This increases the complexity of the original program by negatively affecting its comprehensibility. For this reason, a refinement step is required to remove unnecessary changes. Specifically, in GenProg a minimization process is performed to find a subset of the initial repair edits from which no further elements can be dropped without causing the program to fail a test case (a 1-minimal subset). To deal with the complexity of finding a 1-minimal subset, delta debugging [947] is used. The minimized set of changes is the final repair, that can be inspected for correctness.

Empirical evaluation. GenProg has been used to repair 16 programs for a total of over 1.25 million lines of code [502]. The considered programs contain eight different kinds of defects, i.e., infinite loop, segmentation fault, remote heap buffer overflow to inject code, remote heap buffer overflow to overwrite variables, non-overflow denial of service, local stack buffer overflow, integer overflow, and format string vulnerability. In order to fix such defects, 120K lines of module or program code need to be modified. The results achieved indicated that GenProg was able to fix all these defects in 357 seconds. In addition, GenProg was able to provide repairs that do not appear to introduce new vulnerabilities, nor do they leave the program susceptible to variants of the original exploit.

4.4.3 Model transformation

Similar to program transformation, model transformation aims to derive a target model from a source model by following some rules or principles. Defining transformations for domain-specific or complex languages is a time consuming and dif-

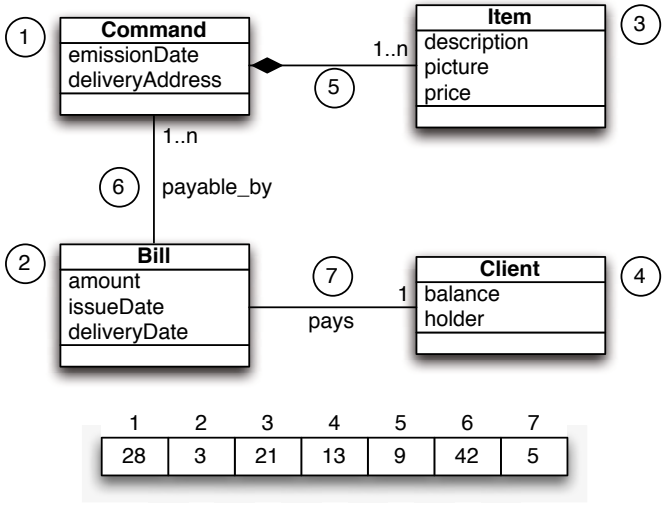

Fig. 4.5: Solution representation for search-based model transformation [459].

ficult task, that requires knowledge of the underlying meta-models and knowledge of the semantic equivalence between the meta-models' concepts[3]. In addition, for an expert it is much easier to show transformation examples than to express complete and consistent transformation rules. This has pushed researchers to define a new approach for model transformation, namely Model Transformation by Examples (MTBE).

Problem definition. Using MTBE it is possible to exploit knowledge from previously solved transformation cases (examples) to transform new models by using combinations of known solutions to a given problem. This means that the target model can be obtained through an optimization process that exploits the available examples. The high number of examples as well as the high number of sequences of application of such transformations make MTBE very expensive to be performed manually. For this reason, a search-based approach, called MOdel Transformation as Optimization by Examples (MOTOE) has been proposed to identify a sub-optimal solution automatically [459]. Specifically, the approach takes as inputs a set of transformation examples and a source model to be transformed, and then it generates as output a target model. The target model is obtained by applying a subset of transformation fragments (code snippets) in the set of examples that best matches the constructs of the source model (using a similarity function).

Solution representation. In MOTOE [459] is represented by a n-dimensional vector, where n is the number of constructs in the model. This means that each construct in the model is represented by an element of such a vector. Each construct is then transformed according to a finite set of m code snippets extracted from

[3] The interested reader can find details on the evolution between models and meta-models in Chapter 2

the transformation examples (for instance change an inheritance relationship to an association). Each code snippet has a unique value ranging from 1 to m. Thus, a particular solution is defined by a vector, where the i^{th} element contains the snippet id (i.e., the transformation id) that has to be used to transform the i^{th} construct in the model. In the example depicted in Figure 4.5 there is a class diagram with 7 constructs, i.e., 4 classes and 3 relationships. The solution is represented by a vector with 7 elements that contains the transformation to be applied to each construct. For instance, the transformation 28 is applied to the class `Command`.

Fitness function. The fitness function quantifies the quality of a transformation solution. In MOTOE, the transformation is basically a 1-to-1 assignment of snippets from the set of examples to the constructs of the source model. Thus, the authors proposed a fitness function that is represented by the sum of the quality of each transformation:

$$f = \sum_{i=1}^{n} a_i \cdot (ic_i + ec_i) \tag{4.7}$$

In order to estimate the quality of each transformation, the authors analyze three different aspects (captured by $a_i \cdot (ic_i + ec_i)$ in the formula above):

- *Adequacy* (a_i) of the assigned snippet to the i^{th} construct. Adequacy is 1 if the snippet associated to i^{th} construct contains at least one construct of the same type, and value 0 otherwise. Adequacy aims at penalizing the assignment of irrelevant snippets.
- *Internal coherence* (ic_i) of the individual construct transformation. The intern coherence measures the similarity, in terms of properties, between the i^{th} construct to transform and the construct of the same type in the assigned snippet.
- *External coherence* (ec_i) with the other construct transformations. Since a snippet assigned to the i^{th} construct contains more constructs than the one that is adequate with the i^{th} construct, the external coherence factor evaluates to which extent these constructs match the constructs that are linked to i^{th} construct in the source model.

The fitness function depends on the number of constructs in the model. To make the values comparable across models with different numbers of constructs, a normalized version of the fitness function can be used [459]. Since the quality of each transformation varies between 0 and 2 (ic_i and ec_i can be both equal to 1), the normalized fitness function is $f_{norm} = \frac{f}{2 \cdot n}$.

Supported search-based techniques and change operators. The search-based techniques used to support model transformation in MOTOE are PSO and a hybrid heuristic search that combines PSO with SA [459]. In the hybrid approach, the SA algorithm starts with an initial solution generated by a quick run of PSO. In both the heuristic the change operator assigns new snippet ids to one or more constructors. Thus, it creates a new transformation solution vector starting from the previous one.

Empirical evaluation. The performance of MOTOE has been evaluated when transforming 12 class diagrams provided by a software industry [459]. The size of the these diagrams varied from 28 to 92 constructs, with an average of 58. Altogether, the 12 examples defined 257 mapping blocks. Such diagrams have been

used to build an example base. Then a 12-fold cross validation procedure was used to evaluate the quality of transformations produced by MOTOE, i.e., one class diagram is transformed by using the remaining 11 transformation examples. The achieved transformations have been compared—construct by construct—with the known transformations in order to measure they correctness (automatic correctness). In addition, a manual analysis of the achieved transformation was performed to identify alternative but still valid transformations (manual correctness). The achieved results indicated that when using only PSO the automatic correctness measure had an average value of 73.3%, while the manual correctness measure had an average value of 93.2%. Instead, when using the hybrid search, correctness is even higher, i.e., 93.4% and 94.8% for the automatic and manual correctness, respectively. This means that the proposed transformations were almost as correct as the ones given by experts.

4.5 Search-based Software Refactoring

Refactoring has been defined as "*the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure*" [301, 664]. Different refactoring operations[4] might improve different quality aspects of a system. Typical advantages of refactoring include improved readability and reduced complexity of source code, a more expressive internal architecture and better software extensibility [301]. For these reasons, refactoring is advocated as a good programming practice to be continuously performed during software development and maintenance [88, 301, 458, 596].

Despite its advantages, to perform refactoring in non-trivial software systems might be very challenging. First, the identification of refactoring opportunities in large systems is very difficult, due to the fact that the design flaws are not always easy to identify [301]. Second, when a design problem has been identified, it is not always easy to apply the correct refactoring operation to solve it. For example, splitting a non-cohesive class into different classes with strongly related responsibilities (i.e., Extract Class refactoring) requires the analysis of all the methods of the original class to identify groups of methods implementing similar responsibilities and that should be clustered together in new classes to be extracted. This task becomes even more difficult when the size of the class to split increases. Moreover, even when the refactoring solution has been defined, the software engineer must apply it without changing the external behavior of the system.

All these observations highlight the need for (semi)automatic approaches supporting the software engineer in (i) identifying refactoring opportunities (i.e., design flaws) and (ii) designing and applying a refactoring solution. To this aim, several different approaches have been proposed in the literature to automate (at least in part) software refactoring [83, 84, 618, 649, 659, 765, 858]. Among them, of interest for

[4] A complete refactoring catalog can be found at http://refactoring.com/catalog/.

this chapter are those formulating the task of refactoring as a search problem in the space of alternative designs, generated by applying a set of refactoring operations. This idea has been firstly presented by O'Keeffe and Cinnéide [656] that propose to treat object-oriented design as a combinatorial optimization problem, where the goal is the maximization of a set of design metrics capturing design quality (e.g., cohesion and coupling). In short, the idea is to represent a software system in an easily manipulable way (*solution representation*), in order to apply a set of refactoring operations to it (*change operators*). This can be done by selecting, during the population evolution, the solutions (i.e., refactored version of the system) maximizing a set of metrics capturing different aspects of design quality (*fitness function*).

Starting from the work by O'Keeffe and Cinnéide, search-based refactoring approaches have been applied to improve several different quality aspects of source code maintainability [659, 765], testability [649], and security [323]. Moreover, search-based refactoring techniques have also been proposed with the aim of introducing design patterns [438], and improving the alignment of code to high-level documentation [615].

The main advantages of search based refactoring techniques as compared to non search based ones are:

- *Higher flexibility.* They are suited to support a wide range of refactoring operations, also allowing to apply several of them in combination with the aim of maximizing the gain in terms of the defined fitness function.
- *Wider exploration of the solution space.* Generally, refactoring approaches are based on heuristics suggesting when a refactoring should be applied. For example, if a method has more dependencies toward a class other than the one it is implemented in, it can be a good candidate for move method refactoring. On the one side, while these heuristics can help in solving very specific design problems, they do not allow a wide exploration of the solution space (i.e., the alternative designs). On the other side, search-based algorithms explore several alternative designs by applying many thousands of refactorings in different orders, with the aim of finding a (sub)optimal solution for the defined fitness function.

We will mainly focus our discussion of search-based software refactoring on the CODe-Imp (Combinatorial Optimisation Design-Improvement) tool [614], used for the first time by O'Keeffe and O'Cinnéide [656] to test the conjecture that the maintainability of object-oriented programs can be improved by automatically refactoring them to adhere more closely to a pre-defined quality model. Then, we will briefly overview other applications of search-based refactoring presented in literature.

4.5.1 The CODe-Imp tool

CODe-Imp [614] is a tool designed to support search-based refactoring and used by several works in this field [323, 615, 649, 658, 659]. In its current implementation, the tool can be applied to software systems written in Java.

Solution Representation. In CODe-Imp the solution representation is the program itself and, in particular, its AST. Having such a representation allows to:

1. Easily evaluate the fitness of a refactoring solution. In fact, through the program AST it is possible to easily extract information exploited by almost all the fitness functions used in the search-based refactoring field, such as the number of methods in a class, the list of attributes, methods, and constructors accessed/called by a particular method, and so on. Thus, the evaluation of a generated solution—i.e., an AST representing a refactored version of the original program—is quite straightforward.
2. Determine which refactorings can legally be applied by the change operator. By "legally" we mean refactorings that do not alter the external behavior of a software system.
3. Easily apply the selected refactorings to a program. Once the search-based algorithm has found the list of refactorings to apply on the object system, it is important that these refactorings can be mapped and applied to the system source code. The mapping with source code is performed through its AST representation.

Change Operators. In the context of search-based refactoring, the change operator is a transformation of the solution representation that corresponds to a refactoring that can be carried out on the source code [659]. The current implementation of CODe-Imp supports the 14 design-level refactorings reported in Table 4.1.

Table 4.1 shows how each refactoring operation supported in CODe-Imp also has its complement (i.e., a refactoring operation undoing it). For example, a Push Down Method refactoring can be undone by applying a Pull Up Method refactoring, as well as a Replace Inheritance with Delegation can be undone by the inverse refactoring operation, i.e., Replace Delegation with Inheritance. This choice is not random, but it is driven by the fact that some search techniques (e.g., SA) must be able to freely move in the solution space. Thus, it must be possible to undo each performed refactoring operation.

Also, before applying a refactoring operation, a set of pre- and post-conditions is verified, to allow the preservation of the system's external behavior. For example, before performing a Push Down Method refactoring it is important to verify that the subclasses involved in this refactoring do not override the method inherited from their superclass. Only refactoring operations satisfying the set of defined pre- and post-conditions are considered as legal change operators in the search algorithm. CODe-Imp adopts a conservative static program analysis to verify pre- and post-conditions.

Note that, the set of change operators reported in Table 4.1 is the one adopted in all work involving the CODe-Imp despite the different final goals of the presented refactoring process, like maintainability [657–659], testability [649], and security [323].

Fitness Function. In search-based refactoring the employed fitness functions are composed of a set of metrics capturing different aspects of source code design quality. The set of adopted metrics strongly depends on the objective of the refactoring process. For example, given a hypothetic search-based approach designed to support

Table 4.1: Refactoring operations supported by CODe-Imp [614].

Refactoring Operation	Description
Push Down Method	Moves a method from a superclass to the subclasses using it
Pull Up Method	Moves a method from some subclasses to their superclass
Decrease/Increase Method Visibility	Changes the visibility of a method by one level (e.g., from private to protected)
Push Down Field	Moves a field from a superclass to the subclasses using it
Pull Up Field	Moves a field from some subclasses to their superclass
Decrease/Increase Field Visibility	Changes the visibility of a field by one level (e.g., from private to protected)
Extract Hierarchy	Adds a new subclass to a non-leaf class *C* in an inheritance hierarchy. A subset of the subclasses of *C* will inherit from the new class.
Collapse Hierarchy	Removes a non-leaf class from an inheritance hierarchy.
Make Superclass Abstract	Declares a constructorless class explicitly abstract.
Make Superclass Concrete	Removes the explicit abstract declaration of an abstract class without abstract methods.
Replace Inheritance with Delegation	Replaces an inheritance relationship between two classes with a delegation relationship
Replace Delegation with Inheritance	Replaces a delegation relationship between two classes with an inheritance relationship

Extract Class refactoring—i.e., the decomposition of a complex low-cohesive class in smaller more cohesive classes—it would be necessary to verify that the extracted classes are (i) strongly cohesive, and (ii) lowly coupled between them. These two characteristics would indicate a good decomposition of the refactored class. Thus, in such a case cohesion and metrics should be part of the defined fitness function.

In the studies conducted with CODe-Imp, several different fitness functions have been used and tuned to reach different final goals in source code design. O'Keeffe and Ó Cinnéide [657] try to maximize the understandability of source code by adopting as fitness function an implementation of the *Understandability* function from the Quality Model for Object-Oriented Design (QMOOD) defined by Bansiya

and Davis [69][5]. QMOOD relates design properties such as encapsulation, modularity, coupling, and cohesion to high-level quality attributes such as reusability, flexibility, and understandability using empirical and anecdotal information [69]. In the work by O'Keeffe and Ó Cinnéide [659] the employed *Understandability* function is defined by Equation 4.8 where software characteristics having a positive coefficient (e.g., *Encapsulation*) positively impact source code understandability, while those having a negative coefficient (e.g., *Abstraction*) negatively impact code understandability. A detailed description of these metrics can be found in [657].

$$\begin{aligned} Understandability = -0.33 \times (&Abstraction + Encapsulation - Coupling \\ &+ Cohesion + Polymorphism - Complexity - DesignSize) \end{aligned} \quad (4.8)$$

O'Keeffe and Ó Cinnéide [659] perform a broader experimentation using, besides the *Understandability* function, also QMOOD's *Flexibility* and *Reusability* functions as evaluation functions. The definition of the *Flexibility* function is given in Equation 4.9 while the *Reusability* function is defined in Equation 4.10.

$$\begin{aligned} Flexibility = 0.25 \times Encapsulation - 0.25 \times Coupling \\ + 0.5 \times Composition + 0.5 \times Polymorphism \end{aligned} \quad (4.9)$$

$$\begin{aligned} Reusability = -0.25 \times Coupling + 0.25 \times Cohesion \\ + 0.5 \times Messaging + 0.5 \times DesignSize \end{aligned} \quad (4.10)$$

It is worth noting that there are very interesting differences across the three above presented fitness functions. For example, in the flexibility fitness function the *Polymorphism* is considered a good factor (i.e., a design quality increasing the flexibility) and thus is multiplied by a positive coefficient (0.5). On the contrary, in the understandability one the *Polymorphism* plays a negative role (-0.33 as coefficient), decreasing the fitness function. Also, the *Design Size* represents a positive factor for the reusability of a software system (+0.5 of coefficient) while it is considered a negative factor for the code understandability (-0.33). These observations highlight that *the fitness function is strongly dependent on the goal of the refactoring process*.

As further support to this claim, the fitness function used in [649] and aimed at increasing program testability is, as expected, totally different from those described above: it is represented by just one metric, the Low-level design Similarity-based Class Cohesion (LSCC) defined by Al Dallal and Briand [11].

Finally, a customized fitness function has been used in CODe-Imp to improve the security of software systems [323]. In this case, the fitness function has been defined as a combination of 16 quality metrics, including cohesion and coupling metrics,

[5] The interested reader can find quality models useful to define alternative fitness functions in Chapter 3

design size metrics, encapsulation metrics, composition metrics, extensibility and inheritance metrics [323].

Supported search-based techniques and change operators. In CODe-Imp a variety of local and meta-heuristic search techniques are implemented [658]. O'Keeffe and O'Cinnéide [658] evaluate the performances of four search techniques, namely HC, Multiple ascent HC, SA, and GAs in maximizing the QMOOD understandability previously described. As for GAs, the solution representation previously described (i.e., based on the AST) is considered as the phenotype, while the sequence of refactorings performed to reach that solution is considered as the genotype. The mutation operator simply add one random refactoring to a genotype, while the crossover operator consists of "cut and splice" crossover of two genotypes, resulting in a change in length of the children strings [658].

Empirical Evaluation. The results of the study performed by O'Keeffe and O'Cinnéide [658] indicated that HC and its variant produce the best results. A similar study has also been performed in [659], where the authors compared HC, Multiple ascent HC, and SA in maximizing all three QMOOD functions described above (i.e., *Understandability*, *Flexibility* and *Reusability*). Also this study highlighted the superiority of HC and its variants against the other techniques, with a quality gain in the values of the fitness function of about 7% for *Flexibility*, 10% for *Reusability*, and 20% for *Understandability* as compared to the original design. For this reason the current version of CODe-Imp just supports the HC algorithm and some of its variants.

4.5.2 Other search-based refactoring approaches

Seng et al. [765] proposed a refactoring approach based on GA aimed at improving the class structure of a system. The phenotype consists of a high-level abstraction of the source code and of several model refactorings simulating the actual source code refactorings. The source code model represents classes, methods, attributes, parameters, and local variables together with their interactions, e.g., a method that invokes another method. The goal of this abstraction is simply to avoid the complexity of the source code, allowing (i) an easier application of the refactoring operations and (ii) a simpler verification of the refactoring pre- and post- conditions needed to preserve the system external behavior. The refactorings supported are Move Method, Pull Up Attribute, Push Down Attribute, Pull Up Method, Push Down Method. Note that also in this case each refactoring operation can be undone by another refactoring operation, allowing a complete exploration of the search space.

Concerning the genotype, it consists of an ordered list of executed model refactorings needed to convert the initial source code model into a phenotype. As for the mutation operator, it is very similar to that discussed for the work of O'Keeffe and O'Cinnéide [658], and simply extends a current genome with an additional model refactoring. As for the crossover operator, it combines two genomes by selecting

the first *n* model refactorings from parent one and adding the model refactorings of parent two to the genome; *n* is randomly chosen [765].

Given the goal of the refactoring process proposed by Seng et al. [765]—i.e., to improve the class structure of a system—the fitness function is defined, as expected, as a set of quality metrics all capturing class quality aspects, and in particular by two coupling metrics (Response for class and the Information-flow-based-coupling), three cohesion metrics (Tight class cohesion, Information-flow-based-cohesion, and the Lack of cohesion of methods), and a complexity metric (a variant of the Weighted method count).

The evaluation reported in [765] shows that the above described approach executed on an open source software system is able to improve the value of several quality metrics measuring class cohesion and coupling. In particular, the improvements in terms of cohesion go from 31% up to 81%, while the reduction of coupling is between 3% and 87%. Moreover, since the approach is fully automated and does not require any developer interaction, the authors manually inspected the proposed refactoring operations to verify their meaningfulness. They found all of them justifiable.

Jensen and Cheng [438] use GP to identify a set of refactoring operations aimed at improving software design by also promoting the introduction of design patterns. As for the previously discussed approaches their solution representation is a high-level representation of the refactored software design and the set of steps (i.e., refactorings) needed to transform the original design into the refactored design.

The change operators defined in the approach by Jensen and Cheng have been conceived for creating instances of design patterns in the source code. An example of these operators is the *Abstraction*, that constructs a new interface containing all public methods of an existing class, thus enabling other classes to take a more abstract view of the original class and any future classes to implement the interface [438].

As for the fitness function, it awards individuals in the generated population exhibiting (i) a good design quality as indicated by the QMOOD metrics [69] previously described in Section 4.5.1, (ii) a high number of design patterns retrieved through a Prolog query executed on the solution representation, and (iii) a low number of refactorings needed to obtain them. The evaluation reported by Jensen and Cheng [438] shows as the proposed approach, applied on a Web-based system, is able to introduce on average 12 new design pattern instances.

4.6 Conclusions

This chapter described how search-based optimization techniques can support software evolution tasks. Table 4.2 summarizes the works we discussed, reporting for each of them (i) the maintenance activity it is related to, (ii) the objectives it aims at maximizing/minimizing, (iii) the exploited search-based techniques, (iv) a reference to the work. We have identified three main kinds of activities for which search-based techniques can be particularly useful. The first area concerns the identification of modules in software projects, with the aim of keeping an evolving system maintainable, of restructuring existing systems, or even of restructuring applications for particular objectives such as the porting towards limited-resource devices. The second area concerns source code (or model) analysis transformation, aimed at achieving different tasks, e.g., finding a patch for a bug. The third area concerns software refactoring, where on the one hand different kinds of refactoring actions are possible [301] on different artifacts belonging to a software system and, on the other hand, there could be different refactoring objectives, such as improving maintainability, testability, or security.

In summary, software engineers might have different possible alternatives to solve software evolution problems, and very often choosing the most suitable one can be effort prone and even not feasible given the size and complexity of the system under analysis, and in general given the number of possible choices for the software engineers. Therefore, it turns out that finding a solution for many software evolution problems is NP-hard [312] and, even if an automatic search-based approach is not able to identify a unique, exact solution, at least it can provide software engineers with a limited set of recommendations, hence reducing information overload [623]. All software engineers need to do in order to solve a software evolution problem using search-based optimization techniques is to (i) encode the problem using a proper representation, (ii) identify a way (fitness function) to quantitatively evaluate how good is a solution for the problem, (iii) define operators to create new solutions from existing ones (e.g., Genetic Algorithms (GAs) selection, crossover, and mutation operators, or hill climbing neighbor operator), and (iv) finally, apply a search-based optimization techniques, such as GAs, hill climbing, simulated annealing, or others. In some cases, there might not be a single fitness function; instead, the problem might be multi-objective and hence sets of Pareto-optimal solutions are expected rather than single solutions (near) optimizing a given fitness function.

Despite the noticeable achievements, software engineers need to be aware of a number of issues that might limit the effectiveness of automatic techniques—including search-based optimization techniques—when being applied to software evolution problems. First, software development—and therefore software evolution—is still an extremely human-centric activity, in which many decisions concerning design or implementation are really triggered by personal experience, that is unlikely to be encoded in heuristics of automated tools. Automatically-generated solutions to software evolution problems tend very often to be meaningless and difficult to be applied in practice. For this reason, researchers should focus their effort in developing optimization algorithms—for example Interactive GAs [818]—where human

Table 4.2: Search-based approaches discussed in this chapter.

Activity	Objectives	Techniques	Reference
Software Modularization	Maximize Modularization Quality (MQ) [555]	HC, SA, GA	[610]
Software Modularization	Multi-objective for maximizing cohesion, minimizing coupling and number of isolated clusters	NSGA-II	[701]
Software Modularization	Maximize MQ and User Constraints	Interactive GA	[82, 360]
Software Miniaturization	Minimize the footprint of an application	GA	[247]
Software Miniaturization	Select features to include in an application when porting it towards a mobile device maximizing customers' satisfaction and minimizing battery consumption	GA	[17]
Program Transformation	Minimize code complexity	HC, GA	[283]
Model transformation	Derive a target model from a source model by following some rules or principles maiming adequacy, internal coherence, and external coherence	PSO, PSO+SA	[459]
Automatic Software Repair	Maximize the tests passed in a given test suite	GP	[502]
Refactoring	Maximize understandability of source code	GA, HC	[657]
Refactoring	Maximize understandability, flexibility, and reusability of source code	HC, SA, GA	[659]
Refactoring	Maximize program testability	HC	[649]
Refactoring	Maximize software security	HC, SA	[323]
Refactoring	Maximize class cohesion, minimize class coupling, minimize class complexity	GA	[765]
Refactoring	Maximize the presence of design patterns	GP	[438]

evaluations (partially) drive the production of problem solutions. For example, in Section 4.3.4 we have described how such techniques can be applied in the context of software modularization. This, however, requires to deal with difficulties occurring when involving humans in the optimization process: human decisions may be inconsistent and, in general, the process tend to be fairly expensive in terms of required effort. To limit such effort, either the feedback can be asked periodically (see Section 4.3.4), or it could be possible to develop approaches that, after a while, are able to learn from feedback using machine learning techniques [535].

Second, especially when the search space of solutions is particularly large, search-based optimization techniques might require time to converge. This may be considered acceptable for tasks having a batch nature. If, instead, one wants to in-

tegrate such heuristics in IDEs—e.g., to continuously provide suggestions to developers [158]—then performance becomes an issue. In such a case, it is necessary to carefully consider the most appropriate heuristic to be used or, whenever possible, to exploit parallelization (which is often possible when using GAs). Last but not least, it is worthwhile to point out that such performance optimization can be particularly desirable when, rather than traditional off-line evolution, one expects automatic run-time system reconfiguration, e.g., in a service-oriented architecture [159] or in scenarios of dynamic evolution such as those described in Chapter 7.6 of this book.