Chapter 10
Architecture and Design of the HeuristicLab Optimization Environment

S. Wagner, G. Kronberger, A. Beham, M. Kommenda, A. Scheibenpflug, E. Pitzer, S. Vonolfen, M. Kofler, S. Winkler, V. Dorfer, and M. Affenzeller

Abstract. Many optimization problems cannot be solved by classical mathematical optimization techniques due to their complexity and the size of the solution space. In order to achieve solutions of high quality though, heuristic optimization algorithms are frequently used. These algorithms do not claim to find global optimal solutions, but offer a reasonable tradeoff between runtime and solution quality and are therefore especially suitable for practical applications. In the last decades the success of heuristic optimization techniques in many different problem domains encouraged the development of a broad variety of optimization paradigms which often use natural processes as a source of inspiration (as for example evolutionary algorithms, simulated annealing, or ant colony optimization). For the development and application of heuristic optimization algorithms in science and industry, mature, flexible and usable software systems are required. These systems have to support scientists in the development of new algorithms and should also enable users to apply different optimization methods on specific problems easily. The architecture and design of such heuristic optimization software systems impose many challenges on developers due to the diversity of algorithms and problems as well as the heterogeneous requirements of the different user groups. In this chapter the authors describe the architecture and design of their optimization environment HeuristicLab which aims to provide a comprehensive system for algorithm development, testing, analysis and generally the application of heuristic optimization methods on complex problems.

S. Wagner · G. Kronberger · A. Beham · M. Kommenda · A. Scheibenpflug · E. Pitzer · S. Vonolfen · M. Kofler · S. Winkler · V. Dorfer · M. Affenzeller
Heuristic and Evolutionary Algorithms Laboratory, School of Informatics, Communication and Media, University of Applied Sciences Upper Austria, Softwarepark 11, 4232 Hagenberg, Austria
e-mail: heal@heuristiclab.com

R. Klempous et al. (eds.), *Advanced Methods and Applications in Computational Intelligence*, Topics in Intelligent Engineering and Informatics 6,
DOI: 10.1007/978-3-319-01436-4_10,

10.1 Introduction

In the last decades a steady increase of computational resources and concurrently an impressive drop of hardware prices could be observed. Nowadays, very powerful computer systems are found in almost every company or research institution, providing huge processing power on a broad basis which was unthinkable some years ago. This trend opens the door for tackling complex optimization problems of various domains that were not solvable in the past. Concerning problem solving methodologies, especially heuristic algorithms are very successful in that sense, as they provide a reasonable tradeoff between solution quality and required runtime.

In the research area of heuristic algorithms a broad spectrum of optimization techniques has been developed. In addition to problem-specific heuristics, particularly the development of metaheuristics is a very active field of research, as these algorithms represent generic methods that can be used for solving many different optimization problems. A variety of often nature inspired archetypes has been used as a basis for new optimization paradigms such as evolutionary algorithms, ant systems, particle swarm optimization, tabu search, or simulated annealing. Several publications show successful applications of such metaheuristics on benchmark and real-world optimization problems.

However, JÃŒrg Nievergelt stated in his article in 1994 "*No systems, no impact!*" [30] and pointed out that well-engineered software systems are a fundamental basis to transfer research results from science to industry. Of course, this statement is also true in the area of heuristic optimization. In order to apply effective and enhanced metaheuristics on real-world optimization problems, mature software systems are required which meet demands of software quality criteria such as reliability, efficiency, usability, maintainability, modularity, portability, or security. Although these requirements are well-known and considered in enterprise software systems, they are not yet satisfactorily respected in the heuristic optimization community. Most heuristic optimization frameworks are research prototypes and are funded by national or international research programs. In such scenarios it is very hard to establish a continuous development process, to take into account many different users, to provide comprehensive support for users, or to reach the maturity of a software product. Therefore, these systems cannot be easily integrated into an enterprise environment.

Another major difficulty regarding the design of general purpose heuristic optimization software systems is that there is no common model for metaheuristics. Due to the heterogeneous nature of heuristic optimization paradigms, it is hard to identify and generalize common concepts which imposes a challenging problem on software developers. Many existing software frameworks focus on one or a few particular optimization paradigms and miss the goal of providing an infrastructure which is generic enough to represent all different kinds of algorithms. The variety of existing frameworks makes it very difficult for researchers to develop and compare their algorithms to show advantageous properties of a new approach. A unified software platform for heuristic optimization would improve this situation, as it would

enable algorithm developers to assemble algorithms from a set of ready-to-use components and to analyze and compare results in a common framework.

Therefore, the development of high quality and mature heuristic optimization software systems would lead to a win-win situation for industry and for science. In this chapter, the authors describe their efforts towards this goal in the development of the flexible architecture and generic design of the HeuristicLab optimization environment. Instead of trying to incorporate different heuristic optimization algorithms into a common model, HeuristicLab contains a generic algorithm (meta-)model that is capable of representing not only heuristic optimization but arbitrary algorithms. By this means HeuristicLab can be used to develop custom algorithm models for various optimization paradigms. Furthermore, state-of-the-art software engineering methodologies are used to satisfy additional requirements such as parallelism, user interaction on different layers of abstraction, flexible deployment, or integration into existing applications.

10.1.1 Related Work

Modern concepts of software engineering such as object-oriented or component-oriented programming represent the state of the art for creating complex software systems by providing a high level of code reuse, good maintainability and a high degree of flexibility and extensibility (see for example [29, 20, 8, 17]). However, such approaches are not yet established on a broad basis in the area of heuristic optimization, as this field is much younger than classical domains of software systems (e.g., word processing, calculation, image processing, or integrated development environments). Most systems for heuristic optimization are one man projects and are developed by researchers or students to realize one or a few algorithms for solving a specific problem. Naturally, when a software system is developed mainly for personal use or a very small, well-known and personally connected user group, software quality aspects such as reusability, flexibility, genericity, documentation and a clean design are not the prime concern of developers. As a consequence, seen from a software engineering point of view, in most cases these applications still suffer from a quite low level of maturity.

In the last years and with the ongoing success of heuristic algorithms in scientific as well as commercial areas, the heuristic optimization community started to be aware of this situation. Advantages of well designed, powerful, flexible and ready-to-use heuristic optimization frameworks were discussed in several publications [34, 28, 37, 22, 44, 13, 31], identifying similar requirements as described in this chapter. Furthermore, some research groups started to head for these goals and began redesigning existing or developing new heuristic optimization software systems which were promoted as flexible and powerful white or even black box frameworks, available and useable for a broad group of users in the scientific as well as in the commercial domain. In comparison to the systems available before, main advantages of these frameworks are a wide range of ready-to-use classical algorithms,

solution representations, manipulation operators, and benchmark problems which make it easy to start experimenting and comparing various concepts. Additionally, a high degree of flexibility due to a clean object-oriented design makes it easy for users to implement custom extensions such as specific optimization problems or algorithmic ideas.

One of the most challenging tasks in the development of such a general purpose heuristic optimization framework is the definition of an object model representing arbitrary heuristic optimization paradigms. This model has to be flexible and extensible to a very high degree so that users can integrate non-standard algorithms that often do not fit into existing paradigms exactly. Furthermore the model should be very fine-grained so that a broad spectrum of existing classical algorithms can be represented as algorithm modules. Then, these modules can serve as building blocks to realize different algorithm variations or completely new algorithms with a high amount of reusable code.

Consequently, the question is on which level of abstraction such a model should be defined. A high level of abstraction leads to large building blocks and a very flexible system. A lower level of abstraction supports reusability by providing many small building blocks, but the structure of algorithms has to be predefined more strictly in that case which reduces flexibility. As a consequence, these two requirements are contradictory to some degree.

Taking a look at several existing frameworks for heuristic optimization, it can be seen that this question has been answered in quite different ways. Several publications have been dedicated to the comparison and evaluation of common frameworks for heuristic optimization and show that each of the existing systems is focused on some specific aspects in the broad spectrum of identified requirements [34, 38, 49, 13, 31]. None of the frameworks is able to dominate the others in all or at least most of the considered evaluation criteria. This indicates that time is ripe for consolidation. The lessons learned in the development of the different frameworks as well as their beneficial features should be shared and incorporated into the ongoing development. As a basis for this process detailed documentation and comprehensive publications are required which motivate and describe the architecture and design as well as the specific features of the systems. Therefore this chapter represents a step towards this direction and describes the HeuristicLab optimization environment in detail and highlights the features that set HeuristicLab apart from other existing systems.

10.1.2 Feature Overview

In contrast to most other heuristic optimization systems, the development of HeuristicLab targets two major aspects: On the one hand HeuristicLab contains a very generic algorithm model. Therefore HeuristicLab is not only dedicated to some specific optimization paradigm (such as evolutionary algorithms or neighborhood-based heuristics) but is capable of representing arbitrary heuristic optimization

algorithms. On the other hand HeuristicLab considers the fact that users of heuristic optimization software are in many cases experts in the corresponding problem domain but not in computer science or software development. Therefore HeuristicLab provides a rich graphical user interface which can be used not only to parameterize and execute algorithms but also to manipulate existing or define new algorithms in a graphical algorithm designer. Furthermore, the HeuristicLab user interface also provides powerful features to define and execute large-scale optimization experiments and to analyse the results with interactive charts. In this way also users who do not have a profound background in programming can work with HeuristicLab easily.

In general the most relevant features of HeuristicLab can be roughly summarized as follows:

- **Rich Graphical User Interface**
  A comfortable and feature rich graphical user interface reduces the learning effort and enables users without programming skills to use and apply HeuristicLab.
- **Many Algorithms and Problems**
  Several well-known heuristic algorithms and optimization problems are already implemented in HeuristicLab and can be used right away.
- **Extensibility**
  HeuristicLab consists of many different plugins. Users can create and reuse plugins to integrate new features and extend the functionality of HeuristicLab.
- **Visual Algorithm Designer**
  Optimization algorithms can be modeled and extended with the graphical algorithm designer.
- **Experiment Designer**
  Users can define and execute large experiments by selecting algorithms, parameters and problems in the experiment designer.
- **Results Analysis**
  HeuristicLab provides interactive charts for a comfortable analysis of results.
- **Parallel and Distributed Computing**
  HeuristicLab supports parallel execution of algorithms on multi-core or cluster systems.

10.1.3 Structure and Content

The remaining parts of this chapter are structured as follows: In Section 10.2 the main user groups of HeuristicLab are identified and their requirements are analyzed. The architecture and design of HeuristicLab is discussed in Section 10.3. Thereby, a main focus is put on the presentation of the flexible and generic algorithm model. Section 10.4 outlines how different metaheuristics can be modeled and Section 10.5 exemplarily describes some optimization problems which are included in HeuristicLab. Finally, Section 10.6 summarizes the main characteristics of HeuristicLab and concludes the chapter by describing several application areas of HeuristicLab and future work.

10.2 User Groups and Requirements

Developing a generic software system for heuristic optimization such as HeuristicLab is a challenging task. The variety of heuristic optimization paradigms and the multitude of application domains make it difficult for software engineers to build a system that is flexible enough and also provides a large amount of reusable components. Additionally, the users of heuristic optimization software systems are also very heterogeneous concerning their individual skills and demands. As a consequence, it is essential to study the different user groups in detail in order to get a clear picture of all requirements.

When considering literature on optimization software systems for different heuristic algorithms, some requirements are repeatedly stated by researchers. For example in [13], Christian Gagné and Marc Parizeau define the following six genericity criteria to qualify evolutionary computation frameworks: generic representation, generic fitness, generic operations, generic evolutionary model, parameters management, and configurable output. Quite similar ideas can also be found in [21, 37, 23, 44, 31].

Although most of these aspects reflect important user demands, none of these publications sketch a clear picture of the system's target users groups. As a consequence, without a precise picture of the users it is hard to determine whether the list of requirements is complete or some relevant aspects have been forgotten. Thus, before thinking about and defining requirements, it is necessary to identify all users.

10.2.1 User Groups

In general, users of a heuristic optimization system can be categorized into three often overlapping groups: practitioners, trying to solve real-world optimization problems with classical or advanced heuristics; heuristic optimization experts, analyzing, hybridizing and developing advanced algorithms; and students, trying to learn about and work with heuristic optimization algorithms and problems. Therefore, these three groups of users called *practitioners*, *experts* and *students* and their views on the area of heuristic optimization as well as their individual needs are described in detail in the following.

10.2.1.1 Practitioners

Practitioners are people who have encountered some difficult (often NP-hard) optimization problem and who want to get a solution for that problem. Hard optimization problems can be found in almost every domain (for example in engineering, medicine, economics, computer science, production, or even in arts), so this group is huge and very heterogeneous. Due to that heterogeneity, it is not possible to list all the domains where heuristic optimization algorithms have already been

successfully applied or even to think of all possible domains in which they might be applied successfully in the future. Therefore, further refinement and categorization of the members of this user group is omitted.

Seen from an abstract point of view, practitioners work in a domain usually not related to heuristic optimization or software engineering. They normally have very little knowledge of heuristic algorithms but a profound and deep knowledge of the problem itself, its boundary conditions and its domain. This results in a highly problem-oriented way of thinking; in this context heuristic optimization algorithms are merely used as black box solvers to get a solution.

Usually, practitioners want to get a satisfactory solution to their problem as quickly as possible. Each second spent on computation means that some real-world system is running in a probably sub-optimal state. To them, time is money, and thus their number one concern is performance. For example, the operators of a production plant are interested in an optimal schedule of operations in order to minimize tardiness of orders and to deliver on time. Any time a machine breaks down, a new order is accepted, or the available capacity changes, a new schedule is needed at once. Each minute production is continued without following an optimized schedule may lead to the wrong operations being chosen for production. This may finally result in a higher tardiness on some high priority orders causing penalties and a severe loss of money.

Parallelism and scalability are of almost equal importance. In a production company the heuristic optimization software system used to compute optimized schedules should be able to provide equally good results, even if business is going well and there are twice as many production orders to be scheduled. A simple equation should hold: More computing power should either lead to better results or to the possibility to solve larger problems. Therefore, it has to be possible to enhance the optimization system with some additional hardware to obtain better performance.

Next, practitioners require a high level of genericity. Due to the heterogeneous nature of domains in which optimization problems might arise, a software system has to support easy integration of new problems. Usually this integration is done by implementing problem-specific objective functions, custom solution representations, and a generic way to introduce new operations on these solutions. For example, new solution manipulation operators might have to be developed, respecting some constraints a feasible solution has to satisfy.

Another important aspect is the integration of a heuristic optimization software system. Usually an optimization system is not a stand-alone application. Data defining a problem instance is provided by other existing software systems and solutions have to be passed on to other applications for further processing. For this reason, in most real-world scenarios heuristic optimization software systems have to be integrated into a complex network of existing IT infrastructure. Well-defined interfaces and technology for inter-system communication and coupling are necessary.

Finally, due to the highly problem-oriented focus of practitioners, they should not have to deal with the internals of algorithms. After a problem has been defined and represented in a heuristic optimization software system, the system should provide a comprehensive set of classical and advanced optimization algorithms. These

algorithms can be evaluated on the concrete problem at hand and a best performing one can be chosen as a black box solver for live operation on real-world data.

10.2.1.2 Experts

Experts are researchers focusing on heuristic optimization algorithm engineering. Their aim is to enhance existing algorithms or develop new ones for various kinds of problems. Following the concept of metaheuristics, especially problem-independent modifications of algorithms are of major interest. By this means different kinds of optimization problems can benefit from such improvements. As a result and in contrast to practitioners, experts consider algorithms as white box solvers. For them, concrete optimization problems are less important and are used as case studies to show the advantageous properties of a new algorithmic concept, such as robustness, scalability, and performance in terms of solution quality and runtime. In many cases, problem instances of well-known benchmark optimization problems, as for example the traveling salesman problem or n-dimensional real-valued test functions, are used. Ideally, a comprehensive set of benchmark problems is provided out of the box by a heuristic optimization software system.

Due to the focus on algorithms, one main concern of experts is genericity. A heuristic optimization software system should offer abilities to integrate new algorithmic concepts easily. There should be as few restrictions in the underlying algorithm model as possible, enabling the incorporation of techniques stemming from different areas such as evolutionary algorithms, neighborhood-based search or swarm systems (hybridization), the flexible modification of existing algorithms, or the development of new ones. The sequence of operations applied to one or more solutions during algorithm execution, in other words the algorithm model, should be freely configurable. Furthermore, experts, similarly to practitioners, demand the integration of new operations, solution representations and objective functions.

One main task of experts is testing algorithms on different kinds of problems, as empirical evaluation is necessary to analyze properties of a new algorithm. Thus, automation of test case execution and statistical analysis play an important role. Thereby, performance in terms of execution time is usually just of secondary importance, as time constraints are not that financially critical as they are for practitioners.

In order to get some hints for algorithm improvements, experts have to use various tools to obtain a thorough understanding of the internal mechanisms and functionality of an algorithm. Since many heuristic optimization algorithms are very sensitive to parameters values, a generic way of parameter management is of great value, owing to the fact that various parameters have to be adjusted from test run to test run. Furthermore, stepwise algorithm execution and customizable output are the basis for any in-depth algorithm analysis, in order to get a clearer picture of how an algorithm is performing and what effect was caused by some parameter adjustment or structural modification.

Finally, replicability and persistence have to be mentioned: Each algorithm run has to be reproducible for the sake of later reference and analysis. A persistence

mechanism is thus an essential means by which a run can be saved at any time during its execution and can be restored later on.

10.2.1.3 Students

Students entering the area of heuristic optimization are users that are located between the two user groups described above. In the beginning, they experiment with various heuristic optimization techniques and try to solve well-known benchmark problems. Therefore, a comprehensive set of classical heuristic optimization algorithms and benchmark problems should be provided by a heuristic optimization software system.

During several experiments and algorithm runs, students gain more and more insight into the internal functionality of algorithms and the interdependency between diversification and intensification of the search process. As the mystery of heuristic optimization is slowly unraveled, their view of algorithms changes from black box to white box. Hence, requirements relevant to experts - such as genericity, parameter management, automation, or customizable output - become more and more important to these users as well.

Additionally, when using heuristic optimization techniques and a heuristic optimization software system for the first time, an easy-to-use and intuitive application programming interface (API) is helpful to reduce the necessary learning effort. Even more, a graphical user interface (GUI) is extremely advantageous, so that students do not have to worry about peculiarities of programming languages and frameworks. Instead, with a GUI they are able to concentrate on the behavior of algorithms on an abstract level. Especially, studying charts and logs of the internal state of an algorithm during its execution is very effective to gain a deeper understanding of algorithm dynamics.

10.2.2 Requirements

Based on the analysis of the three user groups, the following requirements can be defined for heuristic optimization software systems (some of these requirements can be found in similar form in [22, 37, 23, 44, 31]). The requirements are listed alphabetically and the order does not reflect the importance of each requirement.

- **Automation**
  As heuristic algorithms are per se non-deterministic, comparison and evaluation of different algorithms requires extensive empirical tests. Therefore, a heuristic optimization software system should provide functionality for experiment planning, automated algorithm execution, and statistical analysis.
- **Customizable Output**
  In a real-world scenario heuristic optimization never is an end in itself. To enable further processing of results with other applications, the user has to customize the

output format of a heuristic algorithm. Furthermore, user defined output is also required by experts to visualize the internal mechanisms of algorithms by logging internal states such as distribution of solutions in the solution space, similarity of solutions, or stagnation of the search.

- **Generic Algorithm Model**
  In order to represent different kinds of heuristic optimization algorithms, the main algorithm model has to be flexible and customizable. It should not be dedicated or limited to any specific heuristic optimization paradigm. Especially, this aspect should also be kept in mind in terms of naming of classes and methods, so that users are not irritated or misled by the API.
- **Generic Operators**
  Related to the demand for a generic algorithm model, the operations applied by a heuristic algorithm should be generic as well. The user has to be able to implement either problem-specific or generic operations in an easy and intuitive way. There has to be a standardized interface for all operations and a uniform way how data is represented, accessed, and manipulated.
- **Generic Objective Functions**
  To enable integration of custom optimization problems, a generic concept of objective functions has to be available. Users should be able to add custom methods for quality evaluation easily. The quality evaluation mechanism has to be based on a clearly defined interface, so that all other operations depending on quality values - such as selection or heuristic manipulation operations - do not have to take care of how the quality of a solution is calculated in detail. In that context, working with the quality of a solution has to be abstracted from the concrete quality representation. For example, there should be no difference for selection operations whether the optimization problem is single-objective or multi-objective or a minimization or maximization problem. Therefore, a generic way of comparing two solutions is necessary.
- **Generic Solution Representations**
  As users need to integrate custom optimization problems, not only generic objective functions but also a generic way of solution representation is required. Ideally, the user should be able to assemble a custom solution representation by combining different standard data representations such as single values, arrays, matrices or enumerations of different data types. As an alternative, solution representations using complex custom data structures independent of any predefined ones should also be supported. This requirement has a strong impact on the requirement of generic operators, as crossover or manipulation operators have to work on solutions directly.
- **Graphical User Interface**
  To pave the way to heuristic optimization for users not so familiar with software development or specific programming languages, a heuristic optimization software system needs to be equipped with a graphical user interface (GUI). Users should be able to modify or develop algorithms without depending on any

specific development environment. Furthermore, a GUI is also very helpful for experimenting and rapid prototyping, as algorithms can be modeled and visualized seamlessly directly within the system.

- **Integration**
  After their development, heuristic optimization algorithms for real-world optimization problems have to be integrated into some existing information technology landscape. Hence, a heuristic optimization software system should be modular to be able to integrate just the parts required in a custom scenario. Generic communication protocols for the system and its environment are necessary for passing new optimization tasks into and getting results out of the system.
- **Learning Effort**
  Users of a heuristic optimization software system should be able to start to work with the system quickly. Only little knowledge of programming languages and just basic skills in programming and software development should be necessary. The API of a heuristic optimization software system should therefore be intuitive, easy to understand, and should follow common design practices. Additionally, a high level user interface should be provided to decouple algorithm and problem engineering from software development.
- **Parallelism**
  A heuristic optimization software system should be scalable in terms of computing power. Using additional computing resources should either enable the user to solve larger problems, or to achieve better solution quality. Consequently, exploitation of parallelism is an important success factor. A heuristic optimization software system has to offer a seamless integration of parallelism for development and execution of parallel algorithms. Ideally, the user just has to define which parts of an algorithm should be executed in parallel, without having to think about how parallelization is finally done. Due to the duality of high-performance computing systems (shared memory multi-core CPUs versus distributed memory cluster or grid systems) parallelization concepts for both architectures should be provided.
- **Parameter Management**
  Many heuristic optimization algorithms offer several parameters to influence their behavior. As performance of most algorithms is very sensitive in terms of parameter values, users need to run an algorithm many times to find an optimal configuration. Consequently, a generic parameter management facility is required to change parameters without needing to modify any program code or to recompile operators or, even worse, the whole system.
- **Performance**
  Heuristic optimization applications are usually time-critical. Many objective functions of real-world optimization problems as well as heuristic optimization algorithms are very expensive in terms of execution time. Thus a heuristic optimization software system should support runtime efficient implementation and execution of algorithms.

- **Predefined Algorithms and Problems**
  To enable solving of optimization problems or comparison of algorithms out of the box, a heuristic optimization software system has to provide a broad spectrum of predefined algorithms and problems. Especially, it should be possible to use parts of existing algorithms and problems as a basis for further development or hybridization.
- **Replicability and Persistence**
  As experimental evaluation is a substantial task in heuristic algorithm development, test runs of algorithms have to be reproducible. Users should be able to save an algorithm and to restore it later on. The software system should therefore also enable stopping and saving algorithms during execution at any time. In that context, random number generators have to be handled with care, depending on whether an algorithm should be replayed with the same or a different random number sequence.

10.3 Architecture and Design

In order to fulfill the requirements identified in Section 10.2, the authors work on the development of an advanced generic and flexible environment for heuristic optimization called HeuristicLab. HeuristicLab has continuously evolved in the last ten years and three major versions have been developed until now which are referred to as HeuristicLab 1.x, HeuristicLab 2.x, and HeuristicLab 3.x. In the following, the previous versions are briefly covered and especially the newest version, HeuristicLab 3.x, is presented in detail.

10.3.1 HeuristicLab 1.x

The development of HeuristicLab 1.x [40, 44] started in 2002 as a programming project at the Johannes Kepler University Linz, Austria. The main goal of the project was to develop a generic, extensible and paradigm-independent environment for heuristic optimization that can be used by researchers in scientific and industrial projects to develop and compare new optimization algorithms and by students in lectures and exercises.

Microsoft® .NET and the C# programming language were chosen as the development platform for HeuristicLab 1.x. Reasons for this decision were that HeuristicLab 1.x had a strong focus on a graphical user interface (GUI) right from the start to enhance usability and to provide a shallow learning curve especially for students. Consequently, a powerful GUI framework was required that is well integrated into the runtime environment and provides an authentic look and feel of applications. Concerning these aspects, back in 2002 the Microsoft® .NET platform provided

a more promising approach than other alternatives such as Java™ or C++[1]. Other aspects as for example platform independence were of minor importance as the developers were always focused on the Windows® operating system.

Similarly to some other heuristic optimization frameworks such as the Templar framework described in [21, 22], the main idea of the HeurisiticLab 1.x architecture is to provide a clear separation of problem-specific and problem-independent parts. A user should be able to develop a new heuristic optimization algorithm and to test and compare it with several existing optimization (benchmark) problems. Furthermore, a new problem should be easy to integrate and to solve with existing algorithms. By this means, this concept leads to a significant level of code reuse, as heuristic optimization algorithms can be used without any modification to solve new optimization problem and vice versa.

In order to realize this concept, HeuristicLab 1.x offers two abstract base classes called *Algorithm* and *Problem* from which every new extension, either optimization algorithm or problem, has to be inherited. Furthermore, another base class *Solution* represents data entities that are created and evaluated by problems and manipulated by algorithms. Any specific solution encoding has to be inherited from that class. On top of these basic framework classes, the HeuristicLab 1.x GUI layer is located. It provides two more base classes for visualizing algorithms (*AlgorithmForm*) and problems (*ProblemForm*). These classes represent forms in terms of the Microsoft® .NET WinForms framework and are presented in the GUI. As each algorithm is executed in its own thread, transport objects called *Results* are used to inform an AlgorithmForm about the progress of its algorithm by propagating values such as the actual number of evaluated solutions or the current solution quality. To support easy integration of new algorithms and problems, HeuristicLab 1.x additionally is equipped with a plugin mechanism enabling users to add custom extensions without knowing or even having access to the whole source code. As a summary of the HeuristicLab 1.x core architecture, Figure 10.1 schematically shows all these basic classes and their interactions.

As algorithms and problems are loosely coupled to be able to exchange both parts at will, communication between algorithms and problems is realized by delegates (i.e., types representing methods). An algorithm defines method interfaces it expects in order to be able to do its work (e.g., evaluation operators, manipulation operators, or neighborhood operators). On the other side, a problem provides implementations of these interfaces. If implementations of all required delegates are available, a problem can be solved by an algorithm. Linking between delegates (interfaces) and delegate implementations (operators) is done dynamically at runtime using code attributes and reflection. The whole application does not have to be compiled again when integrating new algorithms or problems.

[1] It has to be mentioned that today a very powerful and flexible GUI framework and a comfortable development environment is also available on the Java™ side in form of the Eclipse™ IDE and the Eclipse™ Rich Client Platform. So in fact concerning GUI support the choice of an appropriate runtime environment would not be that easy today, as both solutions, Java™ and Microsoft® .NET, are well developed and reached a high degree of maturity.

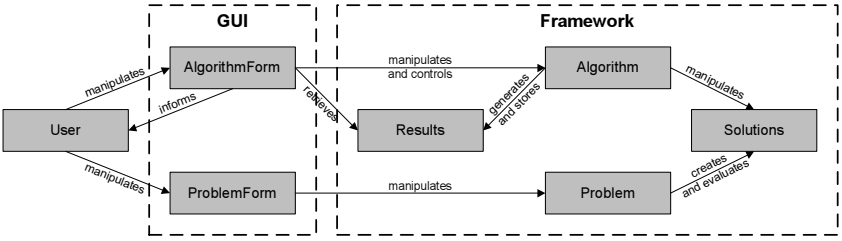


Fig. 10.1. Interaction of HeuristicLab 1.x classes

Beside this basic object model for representing arbitrary heuristic optimization algorithms and problems, HeuristicLab 1.x also includes another front-end for batch execution of algorithms called *TestBench*. In the HeuristicLab 1.x GUI multiple algorithm and problem configurations can be saved in a comma-separated text format (CSV) and can be executed in batch mode using the TestBench. This feature is especially useful for large scale experiments to compare different heuristic optimization algorithms. Furthermore, another sub-project called HeuristicLab Grid [42, 43] offers distributed and parallel batch execution of algorithm runs in a client-server architecture.

In the past, HeuristicLab 1.x has been intensively and successfully used by the research group of Michael Affenzeller in many research projects as well as in several heuristic optimization lectures. A broad spectrum of more than 40 plugins providing different heuristic optimization algorithms and problems has been developed. For example, various genetic algorithm variants, genetic programming, evolution strategies, simulated annealing, particle swarm optimization, tabu search, and scatter search are available. Furthermore, several heuristic optimization (benchmark) problems - for example, the traveling salesman problem, vehicle routing, n-dimensional real-valued test functions, the Boolean satisfiability problem, scheduling problems, or symbolic regression - are provided as plugins. An exemplary screenshot of the HeuristicLab 1.x GUI is shown in Figure 10.2.

10.3.2 HeuristicLab 2.x

Although HeuristicLab 1.x was extensively used in several research projects and was continuously extended with new algorithm and problem plugins, the authors identified a few drawbacks of HeuristicLab 1.x during its development and productive use. The most important ones of these issues are listed and discussed in the following:

- **Monolithic Plugins**
  As HeuristicLab 1.x provides a very high level of abstraction by reducing heuristic optimization to two main base classes (Algorithm and Problem), no specific

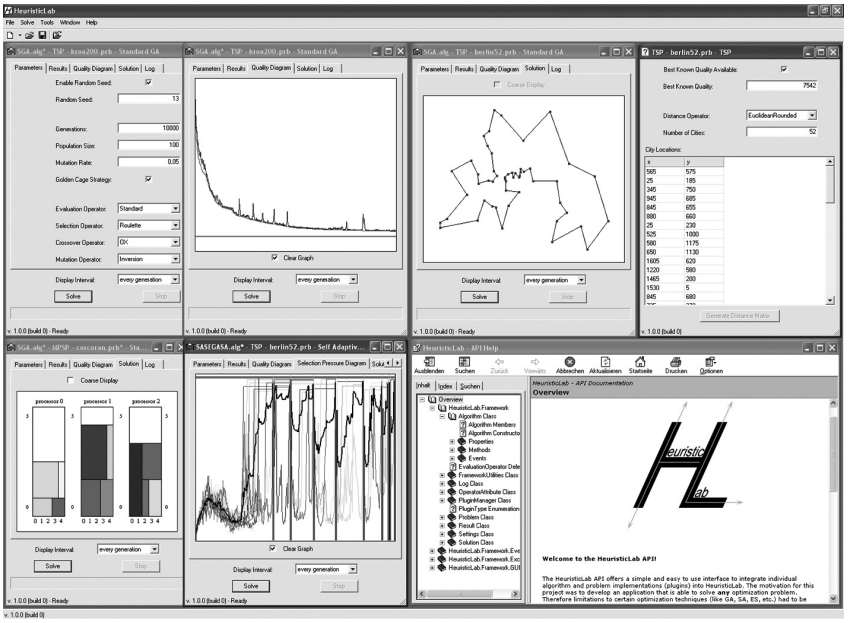


Fig. 10.2. Screenshot of HeuristicLab 1.x

APIs for particular optimization paradigms are available. For example, if a new variant of genetic algorithms is developed that differs from an existing algorithm in just some small aspects (for example using two different selection schemes for selecting solutions for reproduction [45]), the whole algorithm has to be provided in a separate and independent plugin. As more and more algorithms were added to HeuristicLab 1.x, this situation led to a severe amount of code duplication and to a significant reduction of maintainability.

- **Strict Separation of Algorithms and Problems**
  HeuristicLab 1.x requires strict separation of algorithms and problems and loose coupling between these two parts based on delegates. This approach makes it rather complicated to integrate heuristic optimization algorithms such as tabu search or hybrid algorithms that contain problem-specific concepts. A tighter interaction between algorithms and problems should be possible on demand.
- **Interruption, Saving and Restoring of Algorithms**
  The internal state of an algorithm during its execution cannot be persisted in HeuristicLab 1.x. As a consequence, it is not possible to interrupt, save and restore an algorithm during its execution. For example, if an algorithm has to be stopped because computation resources are temporarily required for some other task, the whole run has to be aborted and cannot be continued later on. As algorithm runs might take quite a long time in several application scenarios of heuristic optimization, this behavior turned out to be quite uncomfortable for users.

- **Comprehensive Programming Skills**
  Due to the high level of abstraction, comprehensive programming skills are required especially for developing new heuristic optimization algorithms; each new algorithm plugin has to be developed from scratch. There is only little support for developers in terms of more specialized APIs supporting particular heuristic optimization paradigms. Furthermore, also the HeuristicLab API has to be known to a large extent. It is not possible to assemble algorithms in the GUI dynamically at runtime by defining a sequence of operations without having to use a development environment for compiling a new plugin.

As a result of these insights, the authors decided in 2005 to redesign and extend the core architecture and the object model of HeuristicLab. Based on the Microsoft® .NET 2.0 platform a prototype was developed (HeuristicLab 2.x) that realized a more fine-grained way of representing algorithms.

In HeuristicLab 2.x so-called workbenches represent the basic entities of each heuristic optimization experiment. A workbench contains three main items in form of an algorithm, a problem and a solution representation.

In contrast to HeuristicLab 1.x, solution representations are not directly integrated into problems anymore, but are treated as independent objects. As a consequence, manipulation concepts do not have to be provided by each problem but can be shared, if solutions to a problem are represented in the same way.

Algorithms are no longer represented as single classes inherited from an abstract base class. Instead, each algorithm consists of several operators. Each operator works on either one or more solutions and represents a basic operation (e.g., manipulating or evaluating a solution, selecting solutions out of a solution set, or reuniting different solution sets). Furthermore, an operator may contain and execute other operators, leading to a hierarchical tree structure. By this means, the development of high level operators is possible which represent more complex operations and can be specialized with more specific operators on demand. For example, a solution processing operator can be defined that iterates over a solution set and executes an operator on all contained solutions (for example an evaluation operator). Another example is a mutation operator that executes a manipulation operator on a solution with some predefined probability. This concept leads to a fine-grained representation of algorithms and to better code reuse. Additionally, the strict separation between problem-specific and problem-independent parts is softened, as operators are able to access problem-specific information via the workbench.

The main idea of this enhanced algorithm model is to shift algorithm engineering from the developer to the user level. Complex heuristic optimization algorithms can be built be combining different operators in the GUI of HeuristicLab 2.x (see for example Figure 10.3). This aspect is especially important to support users who are not so well familiar with programming and software development (as for example many practitioners who are experts in some problem domain but not in software engineering). Furthermore, it also enables rapid prototyping and evaluation of new

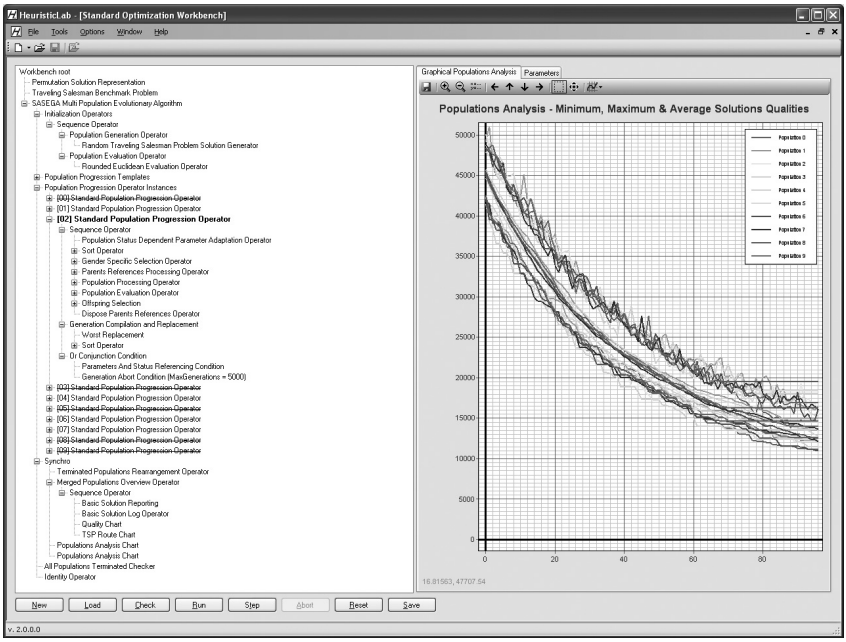


Fig. 10.3. Screenshot of HeuristicLab 2.x

algorithmic concepts, as an algorithm does not have to be implemented and compiled as a plugin.

In order to enable interrupting, saving and restoring of algorithms at any time during execution, a 2-phase commit strategy has been realized. If an algorithm is stopped during its execution, the actual program flow is located somewhere in the depths of the operator tree. As operators also may have local status variables, it has to be assured that the whole operator tree is left in a consistent state, so that execution can be continued, if the algorithm is restored or restarted again. The actual iteration that has not been finished yet has to be rolled back. Consequently, each operator has to keep a local copy of its status variables to be able to restore the last save state of the last completed (and committed) iteration. At the end of an iteration (i.e., a single execution of the top level algorithm of a workbench), a commit is propagated through the whole operator tree indicating that the actual state can be taken for granted.

All these concepts described above were implemented in the HeuristicLab 2.x prototype and were evaluated in several research projects showing the advantages of the new architecture in terms of code reuse, algorithm development time and flexibility. An overview of these applications can be found in Section 10.6 at the end of this chapter.

10.3.3 HeuristicLab 3.x

Even though HeuristicLab 2.x contains fundamental improvements compared to version 1.x and has been extensively used in research projects as well as in lectures on heuristic optimization, major problems regarding its operator model emerged: For example, due to the local status variables stored in operators and due to the nested execution of operators, the implementation of parallel algorithms turned out to be difficult. Moreover, the 2-phase commit strategy caused a severe overhead concerning the development of new operators and the required memory.

Therefore, it seemed reasonable to develop a new version called HeuristicLab 3.x (HL3) completely from scratch to overcome limitations due to architectural and design decisions of previous versions. Although this decision led to starting over the design and development process again, it offered the essential possibility to build a thoroughly consistent heuristic optimization software system by picking up ideas of preliminary projects, integrating novel concepts, and always keeping learned lessons in mind. In the following, the architecture of HL3 and its object and algorithm model are presented in detail (cf. [46, 47, 41]).

10.3.3.1 HeuristicLab Plugin Infrastructure

A plugin is a software module that adds new functionality to an existing application after the application has been compiled and deployed. In other words, plugins enable modularity not only at the level of source code but also at the level of object or byte code, as plugins can be developed, compiled, and deployed independently of the main application. Due to dynamic loading techniques offered in modern application frameworks such as Java™ or Microsoft® .NET, a trend towards software systems can be observed in the last few years that use plugins as main architectural pattern. In these systems the main application is reduced to a basic plugin management infrastructure that provides plugin localization, on demand loading, and object instantiation. All other parts of an application are implemented as plugins. Communication and collaboration of plugins is based on extension points. An extension point can be used in a plugin to provide an interface for other plugins to integrate additional functionality. In other words, plugins can define extension points or provide extensions which leads to a hierarchical application structure. By this approach, very flexible applications can be built, as an application's functionality is determined just by its plugins. If a comprehensive set of plugins is available, a huge variety of applications can be developed easily by selecting, combining and deploying appropriate plugins.

In Section 10.2 genericity has been identified as an essential quality criterion of heuristic optimization software systems. In order to enable integration of custom optimization problems and algorithms, main parts of the system such as objective functions, operators, or solution encodings have to be exchangeable. A high degree of modularity is required and consequently a plugin-based architecture is also reasonable for a heuristic optimization software system [48].

Motivated by the benefits of plugin-based software systems especially in the case of heuristic optimization, plugins are used in HL3 as architectural paradigm as in the versions HeuristicLab 1.x and 2.x. In contrast to previous versions, a lightweight plugin concept is implemented in HL3 by keeping the coupling between plugins very simple: Collaboration between plugins is described by interfaces. The plugin management mechanism contains a discovery service that can be used to retrieve all types implementing an interface required by the developer. It takes care of locating all installed plugins, scanning for types, and instantiating objects. As a result, building extensible applications is just as easy as defining appropriate interfaces (contracts) and using the discovery service to retrieve all objects fulfilling a contract. Interfaces that can be used for plugin coupling do not have to be marked by any specific declaration.

Meta-information has to be provided by each plugin to supply the plugin management system with further details. For example, when installing, loading, updating, or removing a plugin, the plugin infrastructure has to know which files belong to the plugin and which dependencies of other plugins exist. With this information the plugin infrastructure can automatically install other required plugins, or disable and remove dependent plugins, if a base plugin is removed. Hence, it is guaranteed that the system is always in a consistent state. In the HL3 plugin infrastructure all plugin-related data is stored in the source files together with the plugin code. Plugin meta-information is expressed using code annotations instead of separate configuration files, keeping the configuration of the plugin system simple and clean (see Listing 10.1 for an example).

Additionally, a special kind of plugin is necessary: Some designated plugins, called application plugins, have to be able to take over the main application flow. Application plugins have to provide a main method and usually they also offer a GUI. Due to these application plugins the HeuristicLab plugin infrastructure leads to a flexible hierarchical system structure. It is possible to have several different front-ends (applications) within a single system.

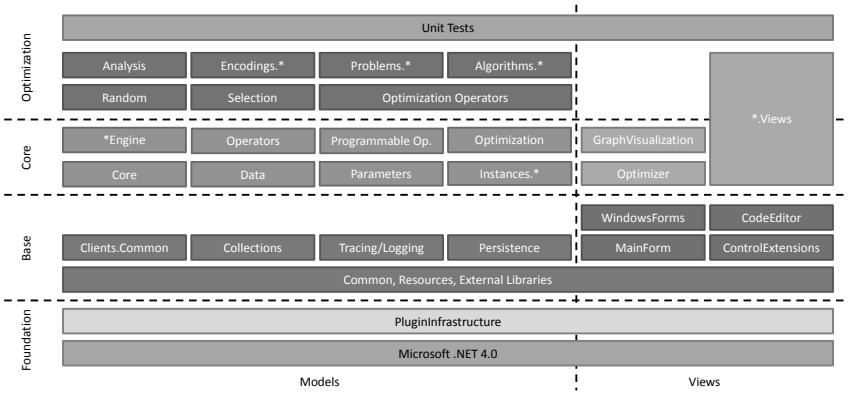


Fig. 10.4. Plugin structure of HL3

Figure 10.4 illustrates the plugin structure of HL3 and Listing 10.1 shows the meta-information of the HeuristicLab Optimizer application plugin as an example.

```
using HeuristicLab.PluginInfrastructure;

namespace HeuristicLab.Optimizer {
  [Plugin("HeuristicLab.Optimizer", "3.3.6.7400")]
  [PluginFile("HeuristicLab.Optimizer-3.3.dll",
              PluginFileType.Assembly)]
  [PluginDependency("HeuristicLab.Clients.Common", "3.3")]
  [PluginDependency("HeuristicLab.Collections", "3.3")]
  [PluginDependency("HeuristicLab.Common", "3.3")]
  [PluginDependency("HeuristicLab.Common.Resources", "3.3")]
  [PluginDependency("HeuristicLab.Core", "3.3")]
  [PluginDependency("HeuristicLab.Core.Views", "3.3")]
  [PluginDependency("HeuristicLab.MainForm", "3.3")]
  [PluginDependency("HeuristicLab.MainForm.WindowsForms", "3.3")]
  [PluginDependency("HeuristicLab.Optimization", "3.3")]
  [PluginDependency("HeuristicLab.Persistence", "3.3")]
  public class HeuristicLabOptimizerPlugin : PluginBase { }

  [Application("Optimizer", "HeuristicLab Optimizer 3.3.6.7400")]
  internal class HeuristicLabOptimizerApplication : ApplicationBase {
    public override void Run() {
      [...]
    }
  }
}
```

Listing 10.1. HeuristicLab.Optimizer application plugin

10.3.3.2 Object Model

Based on the plugin infrastructure, a generic object model is provided by HL3 which is described in detail in this section. Following the paradigm of object-oriented software development, the whole HL3 system is represented as a set of interacting objects. In a logical view these objects are structured in an object hierarchy using the principle of inheritance. Though, this logical structure is not related to the physical structure which is defined by the plugins the application consists of. Therefore, the object hierarchy is spanned over all plugins.

Similarly to the structure of object-oriented frameworks such as the Java™ or Microsoft® .NET runtime environment, a single class named `Item` represents the root of the object hierarchy. It has been decided to implement a custom root class and not to use an existing one of the runtime environment, as it is necessary to extend the root class with some additional properties and methods which usually are not available in the root classes of common runtime environments.

Due to the requirement of replicability and persistence, HL3 offers functionality to stop an algorithm, save it in a file, restore it, and continue it at any later point in time. Hence, all objects currently alive in the system have to offer some kind of persistence functionality (also known as serialization). The persistence mechanism has to be able to handle circular object references, so that arbitrarily complex object graphs can be stored without any difficulties. To simplify communication with other systems and to enable integration into an existing IT environment, XML is used as

a generic way of data representation for storing objects. Additionally, all objects should offer deep cloning functionality which is also determined by the replicability and persistence requirement. Seen from a technical point of view, cloning is very similar to persistence, as cloning an object graph is nothing else than persisting it in memory.

A graphical user interface is the second requirement which has to be considered in the design of the basic object model. The user should be able to view and access all objects currently available at any time, for example to inspect current quality values, to view some charts, or to take a look at a custom representation of the best solution found so far[2]. Nevertheless, the representation of data in a graphical user interface is a very time consuming task and therefore critical concerning performance. Consequently, visualizing all objects per default is not an option. The user has to be able to decide which objects should be presented in the GUI, and to open or close views of these objects as needed. Therefore, a strict separation and loose coupling between objects and views is required.

To realize this functionality, event-driven programming is a suitable approach. If the state of an objects changes and its representation has to be updated, events are used to notify all views. In object-oriented programming this design is known as the observer pattern [14] and is derived from model-view-controller. Model-view-controller (MVC) is an architectural pattern which was described by Trygve Reenskaug for the first time when working together with the Smalltalk group at Xerox PARC in 1979. It suggests the separation into a model (containing the data), views (representing the model) and a controller for handling user interactions (events). By this means, it is assured that the view is decoupled from the model and the control logic and that it can be exchanged easily. A comprehensive description of MVC is given by Krasner and Pope in [27].

In HL3 these concepts are used for object visualization. In addition, a strict decoupling of objects and views simplifies the development of complex views which represent compound objects. A view of a complex object containing several other objects can be composed by adding views of the contained objects. Figure 10.5 outlines this structure in a schematic way and Figure 10.6 shows a screenshot of the HL3 GUI.

As a summary, the basic properties of each HL3 object can be defined as follows:

- **Persistence**
  Objects can be persisted in order to save and restore themselves and all contained objects.
- **Deep Cloning**
  Objects provide a cloning mechanism to create deep copies of themselves.
- **Visualization**
  Views can be created for each object to represent it in a graphical user interface.

In Figure 10.7 the structure of HL3 objects defined by the basic object model is shown.

[2] Additionally, a graphical user interface is also very helpful for reducing the learning effort required when starting to work with the system.

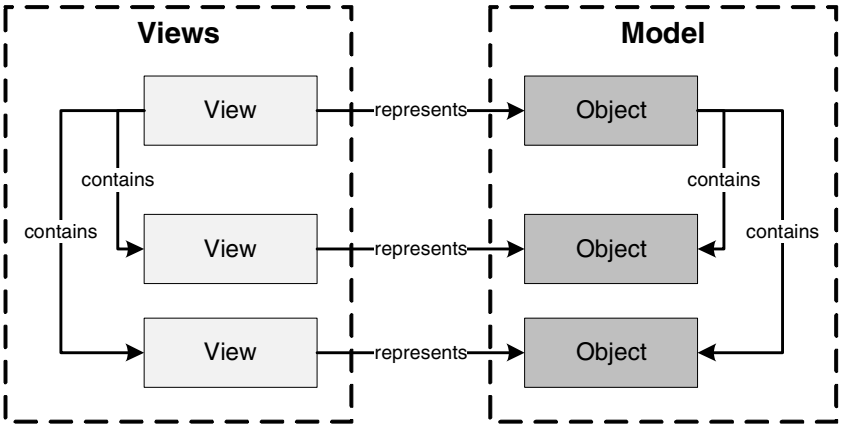


Fig. 10.5. Compound views representing complex objects

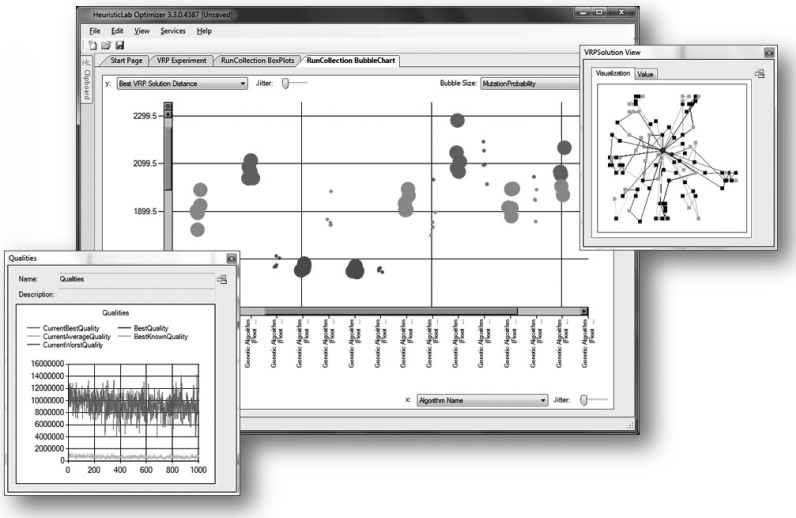


Fig. 10.6. Screenshot of HeuristicLab 3.x

10.3.3.3 Algorithm Model

In the previous section, the HL3 object model has been described. It is the basis for implementing arbitrary objects interacting in HL3. In this section the focus is now on using this object model to represent heuristic optimization algorithms and problems.

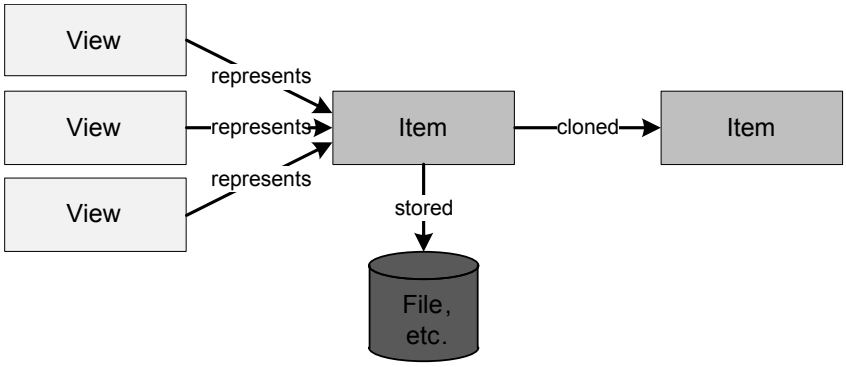


Fig. 10.7. HL3 object model

As stated in the requirements analysis in Section 10.2, the algorithm model of a heuristic optimization software system has to be very generic. Users have to be able to implement custom solution representations and objective functions and to realize individual optimization algorithms. However, the area of heuristic optimization algorithms and problems is very heterogeneous, as many different phenomena (e.g., evolution, hill climbing, foraging, or cooling of matter) were used as a source of inspiration leading to a broad spectrum of algorithms. These algorithms were applied in various problem domains including medicine, finance, engineering, economics, biology, chemistry, and many more.

Furthermore, due to the increase of computational power, many new problem domains are opened up for heuristic algorithms. Today, problems are solved for which using a (meta-)heuristic algorithm was unthinkable a few years ago because of unacceptable execution times. Every year new paradigms of heuristic optimization are introduced, new hybrid algorithms combining concepts of different optimization techniques are developed, and new optimization problems are solved. Therefore, diversity of algorithms and problems in the area of heuristic optimization is growing steadily and it can be expected that this trend will continue within the next years. Hence, developing an algorithm model capable of representing all these different cases is quite challenging.

For software engineers this reveals an interesting problem: On the one hand, a uniform and generic algorithm model is necessary to be able to represent all different optimization paradigms. Even more, the model has to be flexible enough to realize new algorithms that are not even known today and probably will be invented in the future. On the other hand, heterogeneity in the field of heuristic optimization makes it very difficult to develop such a generic model, as the different concepts and paradigms of heuristic optimization can hardly be unified. Although, some efforts were made in the scientific community to develop common models for subareas of heuristic optimization (e.g., for evolutionary algorithms as described in [9]), still there is no common theoretical model of heuristic optimization algorithms in general.

To solve this challenging task, radical abstraction and meta-modeling are essential success factors. As it is impossible to foresee which kinds of algorithms and problems will be implemented in a heuristic optimization software system, abstraction has to be shifted one level higher. Instead of developing a model from the viewpoint of heuristic optimization, the algorithm model has to be able to represent any kind of algorithm in any domain. By this means, the model turns into an algorithm meta-model that enables users to quickly build customized models that exactly fit to their needs.

In order to define such a generic and domain-independent algorithm model, an algorithm can be represented as an interaction of three parts: It is a sequence of *steps* (operations, instructions, statements) describing manipulation of *data* (input and output variables) that is executed by a *machine* (or human). Therefore, these three components (data, operators, and execution) are the core of the HL3 algorithm model and are described in detail in the following.

Data Model

Data values are represented as objects according to the HL3 object model. Therefore, each value can be persisted and viewed. Standard data types such as integers, doubles, strings, or arrays that do not offer these properties have to be wrapped in HL3 objects. In imperative programming, variables are used to represent data values that are manipulated in an algorithm. Variables link a data value with a (human readable) name and (optionally) a data type, so that they can be referenced in statements. Adapting this concept in the HL3 data model, a variable object stores a name, a description, and a value (an arbitrary HL3 object). The data type of a variable is not fixed explicitly but is given by the type of the contained value.

In a typical heuristic optimization algorithm a lot of different data values and therefore variables are used. Hence, in addition to data values and variables, another kind of objects called scopes is required to store an arbitrary number of variables. To access a variable in a scope, the variable name is used as an identifier. Thus, a variable name has to be unique in each scope the variable is contained.

Hierarchical structures are very common in heuristic optimization algorithms. For example, in an evolutionary algorithm an environment contains several populations, each population contains individuals (solutions) and these solutions may consist of different solution parts. Moreover, hierarchical structures are not only suitable for heuristic optimization. In many algorithms complex data structures (also called compound data types) are assembled from simple ones. As a result, it is reasonable to combine scopes in a hierarchical way to represent different layers of abstraction. Each scope may contain any number of sub-scopes which leads to an n-ary tree structure. For example, a scope representing a set of solutions (population) contains other scopes that represent a single solution each.

As operators are applied on scopes to access and manipulate data values or subscopes (as described in the next section), the hierarchical structure of scopes also

has another benefit: The interface of operators does not have to be changed according to the layer an operator is applied on. For example, selection operators and manipulation operators are both applied on a single scope. However, in the case of selection this scope represents a set of solutions and in the case of manipulation it stands for a single solution. Therefore, users are able to create as many abstraction layers as required, but existing operators do not have to be modified. Especially in the case of parallel algorithms, this aspect is very helpful and will be discussed in detail later on.

The hierarchy of scopes is also taken into account when accessing variables. If a variable is not found in a scope, looking for the variable is dedicated to the parent scope of the current scope. The lookup is continued as long as the variable is not found and as long as there is another parent scope left (i.e., until the root scope is reached). Each variable is therefore "visible" in all sub-scopes of the scope that contains it. However, if another variable with the same name is added to one of the sub-scopes, the original variable is hidden due to the lookup procedure[3].

Operator Model

According to the definition of an algorithm, steps are the next part of algorithms that have to be considered. Each algorithm is a sequence of clearly defined, unambiguous and executable instructions. In the HL3 algorithm model these atomic building blocks of algorithms are called operators and are represented as objects. In analogy to imperative programming languages, operators can be seen as statements that represent instructions or procedure calls.

Operators fulfill two major tasks: On the one hand, they are applied on scopes to access and manipulate variables and sub-scopes. On the other hand, operators define which operators are executed next.

Regarding the manipulation of variables, the approach used in the HL3 operator model is similar to formal and actual parameters of procedures. Each operator contains parameters which contain a (formal) name, a description, and a data type and are used to access and manipulate values. In general, the HL3 operator model provides two major groups of parameters: value parameters and lookup parameters. In value parameters the parameter value is stored in the parameter object itself and can be manipulated directly by the user. They usually represent input parameters of an operator which have to be set by the user. Lookup parameters do not contain but refer to a value and are responsible for retrieving the current parameter value dynamically at runtime. For this purpose an actual name has to be defined in each lookup parameter which is used in an internal value lookup mechanism to fetch the current value from the variables stored in the scope tree.

By this means, operators are able to encapsulate functionality in an abstract way: For example, a simple increment operator contains a single parameter object indicating that the operator manipulates an integer variable. Inside the operator this

[3] This behavior is very similar to blocks in procedural programming languages. That is the reason why the name "scope" has been chosen.

parameter is used to implement the increment operation. After instantiating an increment operator and adding it to an algorithm at run time, the user has to define the concrete name of the value which should be incremented. This actual name is also set in the parameter. However, the original code of the operator (i.e., the increment operation) does not have to be modified, regardless which value is actually incremented. The implementation of the operator is decoupled from concrete variables. Therefore, the operator can be reused easily to increment any integer value.

Regarding their second major purpose, operators define the execution flow of an algorithm. Each operator may contain parameters which refer to other operators, which defines the static structure of an algorithm. When an operator is executed, it can decide which operators have to be executed next. Hence, complex algorithms are built by combining operators.

Control operators can be implemented that do not manipulate data, but dynamically define the execution flow. For example, branches can be realized by an operator that chooses a successor operator depending on the value of a variable in the scope the branch operator is applied on (cf. if- or switch-statements).

In contrast to scopes, operators are not combined hierarchically, but represent a graph. An operator used in an upper level of an algorithm can be added as a sub-operator in a lower level again. Thus, operator references may contain cycles. In combination with branches, these cycles can be used to build loops (see Section 10.4 for a detailed description of control operators).

As described above, classical concepts of programming such as sequences, branches, loops, or recursion can be represented in the operator model. Therefore, the HL3 algorithm model is capable of representing any algorithm that can be described in imperative programming languages.

Execution Model

The execution of algorithms is the last aspect which has to be defined in the HL3 algorithm model. Algorithms are represented as operator graphs and are executed step-by-step on virtual machines called engines. In each iteration an engine performs an operation, i.e., it applies an operator to a scope. Before executing an algorithm, each engine is initialized with a single operation containing the initial operator and an empty global scope.

As the execution flow of an algorithm is dynamically defined by its operators, each operator may return one or more operations that have to be executed next. Consequently, engines have to keep track of all operations that wait for execution. These pending operations are kept in a stack. In each iteration, an engine pops the next operation from the top of its stack, executes the operator on the scope, and pushes all returned successor operations back on the stack again in reversed order[4]. By this means, engines perform a depth-first expansion of operators. Listing 10.2 states the main loop of engines in pseudo-code.

[4] Reversing the order of operations is necessary to preserve the execution sequence, as a stack is a last-in-first-out queue.

```
clear global scope // remove all variables and sub-scopes
clear operations stack
push initial operation // initial operator and global scope

WHILE NOT operations stack is empty DO BEGIN
    pop next operation
    apply operator on scope
    push successor operations // in reverse order
END WHILE
```

Listing 10.2. Main loop of HL3 engines

Finally, as a summary of the HL3 algorithm model and its three sub-models (data model, operator model, execution model), Figure 10.8 shows the structure of all components.

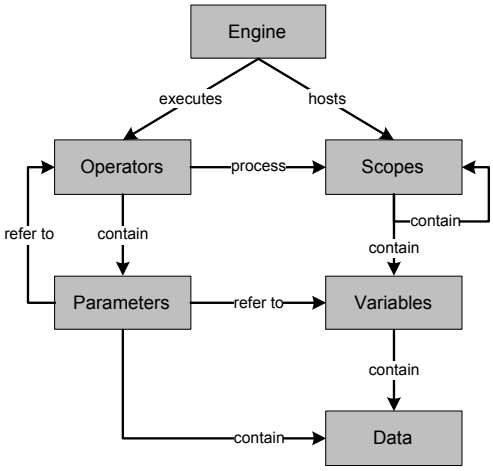


Fig. 10.8. HL3 algorithm model

10.3.3.4 Parallelism

As already pointed out in Section 10.2, short execution time is very important in many real-world applications of heuristic optimization. In order to increase performance, concepts of parallel and distributed computing are used frequently to utilize multiple cores or even computers and to distribute the work load. In the area of parallel heuristic optimization several models of parallelization have been developed which reflect different strategies. In general, there are two main approaches:

The first approach is to calculate the quality of solutions in parallel. In many optimization problems, evaluating a solution requires much more runtime than the execution of solution manipulation operators. As an example consider heuristic optimization for production planning or logistics: In that case, evaluating a solution is done by building a schedule of all jobs or vehicles (for example by using the Giffler-Thompson scheduling algorithm [15]), whereas a solution manipulation operator just needs to change permutations[5]. As another example, heuristic optimization of data representation or simulation models can be mentioned. In these applications, evaluating a solution means executing the whole model (i.e., performing the simulation or checking the quality of the model for all training data). In both examples (and there are many more), evaluating a single solution is independent of all other solutions. Therefore, quality calculation can be easily executed in parallel which is called *global parallelization* [2]. However, the heuristic algorithm performing the optimization is still executed sequentially.

The second approach is to parallelize heuristic optimization algorithms directly [2, 6]. By splitting solution candidates into distinct sets, an algorithm can work on these sets independently and in parallel. For example, parallel multi-start heuristics are simple representatives of that idea. In a parallel multi-start heuristic algorithm, multiple optimization runs are executed with different initial solutions in order to achieve larger coverage of the solution space. Nevertheless, no information is exchanged between these runs until all of them are finished and the best solution is determined. In more complex algorithms (e.g., coarse- or fine-grained parallel genetic algorithms), information is additionally exchanged from time to time to keep the search process alive and to support diversification of the search. Hence, population-based heuristic optimization algorithms are especially well suited for this kind of parallelization.

Consequently, a heuristic optimization software system should consider parallelization in its algorithm model. It has to provide sequential as well as parallel blocks, so that all different kinds of parallel algorithms can be represented. Furthermore, the definition of parallel parts in an algorithm has to be abstracted from parallel execution. By this means, users can focus on algorithm development and do not have to rack their brains on how parallelism is actually implemented.

In the HL3 algorithm model presented in the previous section, data, operators and algorithm execution have been separated strictly. Consequently, parallelism can be integrated by grouping operations into sets that might be executed in parallel. As an operator may return several operations that should be executed next, it can mark the successor operations as a parallel group. These operations are then considered to be independent of each other and the engine is able to decide which kind of parallel execution should be used. How this parallelization is actually done just depends on the engine and is not defined by the algorithm.

HL3 offers two engines which implement different parallelization concepts. The *ParallelEngine* is based on the Microsoft® .NET Task Parallel Library (TPL) and

[5] This depends on the solution encoding, but variations of permutation-based encoding are frequently used for combinatorial optimization problems and have been successfully applied in many applications.

enables parallel execution on multiple cores. The *HiveEngine* is based on HeuristicLab's parallel and distributed computing infrastructure called Hive and provides parallel execution of algorithms on multiple computers. By this means, the user can specify the parallelization concept used for executing parallel algorithms by choosing an appropriate engine. The definition of an algorithm is not influenced by that decision. Additionally, HL3 also contains a *SequentialEngine* and a *DebugEngine* which both do not consider parallelism at all and execute an algorithm sequentially in every case. These engines are especially helpful for testing algorithms before they are really executed in parallel.

Based on this parallelization concept, HL3 provides special control operators for parallel processing. Data partitioning is thereby enabled in an intuitive way due to the hierarchical structure of scopes. For example, the operator *SubScopesProcessor* can be used to apply different operators on the sub-scopes of the current scope in parallel. Therefore, parallelization can be applied on any level of scopes which enables the definition of global, fine- or coarse-grained parallel heuristic algorithms (for a detailed description of parallel control operators see Section 10.4).

10.3.3.5 Layers of User Interaction

Working with HL3 on the level of its generic algorithm model offers a high degree of flexibility. Therefore, this level of user interaction is very suitable for experts who focus on algorithm development. However, dealing with algorithms on such a low level of abstraction is not practical for practitioners or students. Practitioners require a comprehensive set of predefined algorithms which can be used as black box optimizers right away. Similarly, predefined algorithms and also predefined problems are equally important for students, so that they can easily start to experiment with different algorithms and problems and to learn about heuristic optimization. Consequently, HL3 offers several layers for user interaction that reflect different degrees of detail [3].

Predefined algorithms are provided that contain entire operator graphs representing algorithms such as evolutionary algorithms, simulated annealing, hill climbing, tabu search, or particle swarm optimization. Therefore, users can work with heuristic optimization algorithms right away and do not have to worry about how an algorithm is represented in detail. They only have to specify problem-specific parts (objective function, solution encoding, solution manipulation) or choose one of the predefined optimization problems. Additionally, custom views can be provided for these algorithms that reveal only the parameters and outputs the user is interested in. By this means, the complexity of applying heuristic optimization algorithms is significantly reduced.

Between the top layer of predefined solvers and the algorithm model, arbitrary other layers can be specified that offer algorithm building blocks in different degrees of detail. For example, generic models of specific heuristic optimization paradigms, as for example evolutionary algorithms or local search algorithms, are useful for

experimenting with these algorithms. These models also hide the full complexity and flexibility of the algorithm model from users who therefore can solely concentrate on the optimization paradigm. Especially in that case the graphical user interface is very important to provide a suitable representation of algorithms.

In Figure 10.9 the layered structure of user interaction in HL3 is shown schematically. Users are free to decide how much flexibility they require and on which level of abstraction they want to work. However, as all layers are based on the algorithm model, users can decrease the level of abstraction step by step, if additional flexibility is necessary in order to modify an algorithm.

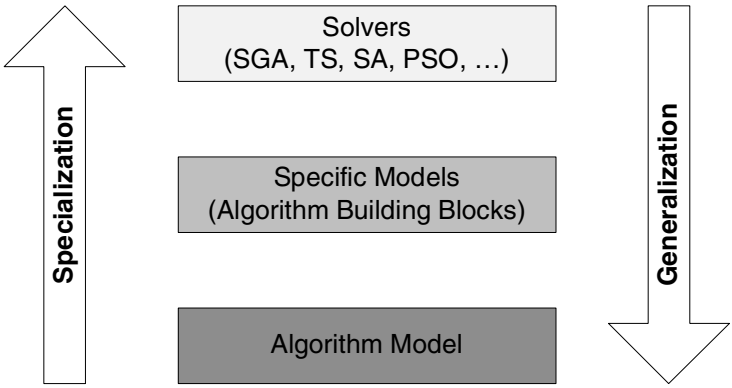


Fig. 10.9. Layers of user interaction in HL3

10.3.4 Analysis and Comparison

A comparison of the different versions of HeuristicLab concerning the requirements defined in Section 10.2 is shown in Table 10.1. A bullet (•) indicates that a requirement is fulfilled, a circle (○) marks requirements which are somehow addressed but are not satisfactorily fulfilled, and a dot (·) shows requirements which are not considered.

HeuristicLab 1.x provides a paradigm-independent algorithm model which is very similar to the model of Templar [21, 22], although it lacks a clean separation of optimization problems and solution representations. Furthermore, algorithm runs cannot be interrupted, saved, and restarted. It provides a plugin mechanism that enables dynamic extension, web-based deployment, and clean integration into other applications. A broad spectrum of plugins for trajectory-based as well as population-based metaheuristics and many different optimization problems have been implemented. Furthermore, a bunch of plugins developed in the HeuristicLab Grid project enables parallel and distributed batch execution. Additionally, GUI components are integrated into the core framework and enable graphical and interactive configuration and analysis of algorithms.

Table 10.1. Evaluation and comparison of HeuristicLab

	HeuristicLab 1.x	HeuristicLab 2.x	HeuristicLab 3.x
Automation	●	○	●
Customizable Output	●	●	●
Generic Algorithm Model	●	●	●
Generic Operators	○	●	●
Generic Objective Functions	○	○	●
Generic Solution Representations	○	●	●
Graphical User Interface	●	●	●
Integration	●	●	●
Learning Effort	●	○	●
Parallelism	○	○	●
Parameter Management	○	○	●
Performance	○	○	○
Predefined Algorithms and Problems	●	○	●
Replicability and Persistence	·	●	●

HeuristicLab 2.x was designed to overcome some of the drawbacks of the previous version. The main purpose was to combine a generic and fine-grained algorithm model with a GUI to enable dynamic and interactive prototyping of algorithms. Algorithm developers can use the GUI to define, execute, and analyze arbitrary complex search strategies and do not have to be experienced programmers. Furthermore, HeuristicLab 2.x also offers support for replicability and persistence, as algorithm runs can be aborted, saved, loaded and continued. However, HeuristicLab 2.x is a research prototype and has never been officially released. Therefore, it still contains some drawbacks concerning performance, stability, usability, learning effort, and documentation. Additionally, the implementation of parallel metaheuristics is difficult due to the nested execution of operators and the local status variables which are used in many operators.

Finally, HL3 fulfills almost all requirements identified so far. Just performance remains as the last requirement that is still a somehow open issue. Even though HL3 enables parallel execution of algorithms and can therefore utilize multi-core CPUs or clusters, the runtime performance of sequential algorithms is worse compared to other frameworks. Reasons for this drawback are the very flexible design of HL3 and the dynamic representation of algorithms. As algorithms are defined as operator graphs, an engine has to traverse this graph and execute operators step by step. Of course, this requires more resources in terms of runtime and memory as is the case when algorithms are implemented as static blocks of code. However, if more effort has to be put on the evaluation of solutions, the overhead of the HeuristicLab

3.x algorithm model is less critical. This is in fact an important aspect, as most resources are consumed for the evaluation of the objective function in many real-world optimization applications. Therefore, the benefits of HeuristicLab 3.x in terms of flexibility, genericity and extensibility should outweigh the additional overhead. Furthermore, the efficiency of HeuristicLab 3.x can be easily improved by executing operators in parallel.

10.4 Algorithm Modeling

In HeuristicLab algorithms are modeled using the operator concept described in Section 10.3. HeuristicLab offers a wide range of operators that can be used to model any type of algorithm. Of course because HL is primarily used to implement metaheuristics, there are a lot of operators which are specially designed for this kind of algorithms. In the following section, operators are presented that are later used to build some typical examples of standard heuristic optimization algorithms. Furthermore, these operators serve as algorithm building blocks for successively defining more complex parallel and hybrid metaheuristics, showing the flexibility and genericity of the framework.

10.4.1 Operators

First of all, simple operators are discussed that perform typical tasks required in every kind of algorithm.

EmptyOperator

The *EmptyOperator* is the most simple form of an operator. It represents an operator that does nothing and can be compared to an empty statement in classical programming languages.

To get an idea of how an operator implementation looks like in HL, Listing 10.3 shows its implementation. Each operator is inherited from the abstract base class *SingleSuccessorOperator*. This base class takes care of aspects that are identical for all operators (e.g., storing of sub-operators and variable information, persistence and cloning, and events to propagate changes). Note that the most important method of each operator is *Apply* which is called by an engine to execute the operator. Apply may return an object implementing *IOperation* which represents the successor operations. If the operator has no successor operations, null is returned.

```
1  public sealed class EmptyOperator : SingleSuccessorOperator {
2    public EmptyOperator() : base() { }
3
4    public override IOperation Apply() {
5      return base.Apply();
6    }
7  }
```

Listing 10.3. Source code of EmptyOperator

Random

As most heuristic optimization algorithms are stochastic processes, uniformly distributed high quality pseudo-random numbers are required. For creating random numbers a single pseudo-random number generator (PRNG) should be used to enable replicability of runs. By setting the PRNG's random seed the produced random number sequence is always identical for each run; it is therefore useful to create a single PRNG into the global scope. Although PRNGs are also represented as data objects and consequently can be stored in variables, it has been decided to implement a custom operator *RandomCreator* for this task. Reasons are that it can be specified for a RandomCreator whether to initialize the PRNG with a fixed random seed to replay a run or to use an arbitrary seed to get varying runs.

Counter

IntCounter is a basic operator for incrementing integer variables. It increases the value of a variable in a scope by a specified value. As an example, this operator can be used for counting the number of evaluated solutions or to increment the actual iteration number.

As the Counter operator is the first operator that changes a variable, Listing 10.4 shows its implementation to give an impression how the manipulation of variables in a scope is done. In the constructor two parameters are defined for looking up the values needed for executing the increment operation. First of all a parameter for looking up the *Value* to increment is added as well as a parameter that contains the value with which the number will be incremented. If there is no *Increment* variable defined in the scope a default value of 1 is used. In the *Apply* method the actual value of *Value* and *Increment* are used. *ActualValue* automatically translates the formal name into the actual name of the parameter (which e.g. has been specified by the user in the GUI) and does the parameter lookup on the scope. The *ValueLookupParameter* uses the locally defined value if it can't find the name on the scope. If the *ValueParameter* can't find a value for the given name, a new value is created in the *Apply* method. Finally, the value is incremented and the successor operation is executed.

```
1  public class Counter : SingleSuccessorOperator {
2    public ILookupParameter<IntValue> ValueParameter {
3      get {
4        return (ILookupParameter<IntValue>)Parameters["Value"];
5      }
6    }
7    public IValueLookupParameter<IntValue> IncrementParameter {
8      get {
9        return (IValueLookupParameter<IntValue>)Parameters["Increment"];
10     }
11   }
12
13   public Counter() {
14     Parameters.Add(new LookupParameter<IntValue>("Value",
15       "The value which should be incremented."));
16     Parameters.Add(new ValueLookupParameter<IntValue>(
17       "Increment",
18       "The increment which is added to the value.",
19       new IntValue(1)));
20   }
21
22   public override IOperation Apply() {
23     if (ValueParameter.ActualValue == null)
24       ValueParameter.ActualValue = new IntValue();
25     ValueParameter.ActualValue.Value +=
26       IncrementParameter.ActualValue.Value;
27     return base.Apply();
28   }
29 }
```

Listing 10.4. Source code of the Counter

Comparator

The *Comparator* operator is responsible for comparing the values of two variables. It expects two input variables which should be compared and a comparison operation specifying which type of comparison should be applied (e.g., less, equal, greater or equal). After retrieving both variable values and comparing them, Comparator creates a new Boolean variable containing the result of the comparison and writes it back into the scope.

ConditionalBranch

The operator *ConditionalBranch* can be used to model simple binary branches. It retrieves a Boolean input variable from the scope tree. Depending on the value of this variable, an operation collection containing the successor operation and additionally either the first (true branch) or the second sub-operator (false branch) is returned. Note that do-while or repeat-until loops can be constructed without any other specific loop operator by combining the operators ConditionalBranch and any sequence of operations to execute in the body of the loop. Figure 10.10 shows an operator graph for these two loop structures.

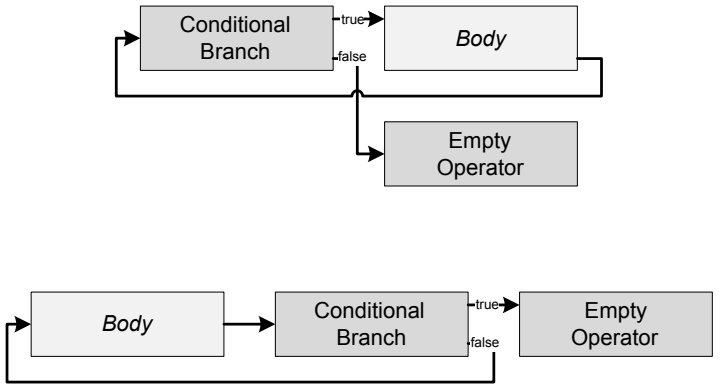


Fig. 10.10. Operator graphs representing a while and a do-while loop

StochasticBranch

In heuristic optimization algorithms it is a common pattern to execute operations with a certain probability (for example mutation of individuals in evolutionary algorithms or post-optimization heuristics in hybrid algorithms). Of course, this could be realized by using an operator creating a random number into a scope in combination with the Comparer and ConditionalBranch operators. However, for convenience reasons the *StochasticBranch* operator performs this task in one step. It expects a double variable as an input specifying the probability and a PRNG. When applying the operator a new random number between 0 and 1 is generated and compared with the probability value. If the random number is smaller, the true branch (first sub-operator) or otherwise the false branch (second sub-operator) is chosen.

UniformSubScopesProcessor

HeuristicLab offers operators to navigate through the hierarchy levels of the scope tree (i.e., to apply an operator on each sub-scope of the current scope). The *UniformSubScopesProcessor* fulfils this task by returning an operation for each sub-scope and its first sub-operator. Additionally this operator has a parameter for enabling parallel processing of all sub-scopes if the engine supports it. This leads to a single program multiple data style of parallel processing.

SubScopesProcessor

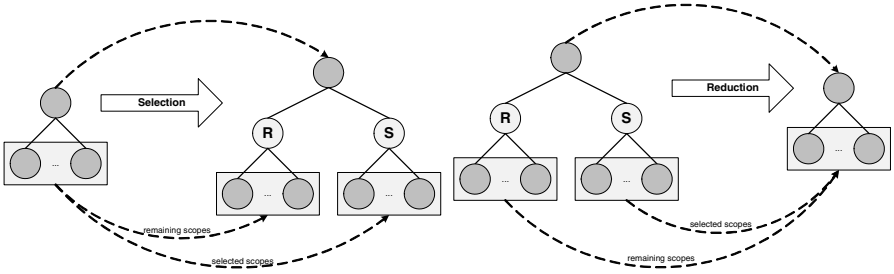
As a generalization of the UniformSubScopesProcessor is the *SubScopesProcessor* that returns an operation not only for the first sub-operator and every sub-scope, but pair sub-operators and sub-scopes together. For each sub-scope there has to be a

sub-operator which is executed on its corresponding sub-scope to enable individual processing of all sub-scopes.

Selection and Reduction

Seen from an abstract point of view, a large group of heuristic optimization algorithms called improvement heuristics follows a common strategy: In an initialization step one or more solutions are generated either randomly or using construction heuristics. These solutions are then iteratively manipulated in order to navigate through the solution space and to reach promising regions. In this process manipulated solutions are usually compared with existing ones to control the movement in the solution space depending on solution qualities. Selection splits solutions into different groups either by copying or moving them from one group to another; replacement merges solutions into a single group again and overwrites the ones that should not be considered anymore.

In the HL algorithm model each solution is represented as a scope and scopes are organized in a hierarchical structure. Therefore, these two operations, selection and replacement, can be realized in a straight forward way: Selection operators split sub-scopes of a scope into two groups by introducing a new hierarchical layer of two sub-scopes in between, one representing the group of remaining solutions and one holding the selected ones as shown in Figure 10.11a. Thereby solutions are either copied or moved depending on the type of the selection operator. Reduction operators represent the reverse operation. A reduction operator removes the two sub-scopes again and reunites the contained sub-scopes as shown in Figure 10.11b. Depending on the type of the reduction operator this reunification step may also include elimination of some sub-scopes that are no longer required.



(a) General principle of selection operators (b) General principle of reduction operators

Fig. 10.11. Selection and reduction operators

Following this simple principle of selection and reduction of solutions, HL provides a set of predefined selection and reduction operators that can be used as a basis for realizing complex selection and replacement schemes.

Selection Operators

The most trivial form of selection operators are the two operators *LeftSelector* and *RightSelector* which select sub-scopes either starting from the leftmost or the rightmost sub-scope. If the sub-scopes are ordered for example with respect to solution quality, these operators can be used to select the best or the worst solutions of a group. If random selection of sub-scopes is required, *RandomSelector* can be used which additionally expects a PRNG as an input variable.

In order to realize more sophisticated ways of selection, *ConditionalSelector* can be used which selects sub-scopes depending on the value of a Boolean variable contained in each sub-scope. This operator can be combined with a selection preprocessing step to create this Boolean variable into each scope depending on some other conditions.

Furthermore, HL also offers a set of classical quality-based selection schemes well-known from the area of evolutionary algorithms, as for example fitness proportional selection optionally supporting windowing (*ProportionalSelector*), linear rank selection (*LinearRankSelector*), or tournament selection with variable tournament group sizes (*TournamentSelector*). Additionally, other individual selection schemes can be integrated easily by implementing custom selection operators.

Reduction Operators

Corresponding reverse operations to LeftSelector and RightSelector are provided by the two reduction operators *LeftReducer* and *RightReducer*. Both operators do not reunite sub-scopes but discard either the scope group containing the selected or the group containing the remaining scopes. LeftReducer performs a reduction to the left and picks the scopes contained in the left sub-scope (remaining scopes) and RightReducer does the same with the right sub-scopes (selected scopes). Additionally, another reduction operator called *MergingReducer* is implemented that reunites both scope groups by merging all sub-scopes.

The following sections show how the described operators are used for designing a genetic algorithm and simulated annealing.

10.4.2 Modeling Genetic Algorithms

Genetic algorithms [19] are population-based metaheuristics which apply processes from natural evolution to a population of solution candidates. This includes natural selection, crossover, mutation and a replacement scheme for updating the old population with the newly generated one. Figure 10.12 gives and overview of a genetic algorithm.

The genetic algorithm starts with generating a population of solution candidates. The selection operation then selects individuals based on a certain scheme (e.g., randomly or according to their qualities). In the next two steps the selected individuals

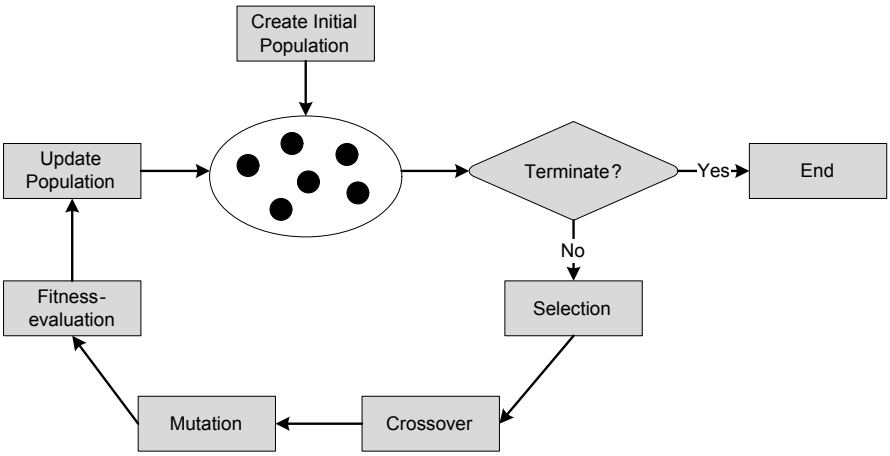


Fig. 10.12. Overview of a genetic algorithm

are crossed and mutated leading to a new population of solution candidates that then get evaluated. In the last step the old population gets replaced with the newly generated one. This procedure is repeated until a termination criterion (e.g., maximum number of generations) is reached.

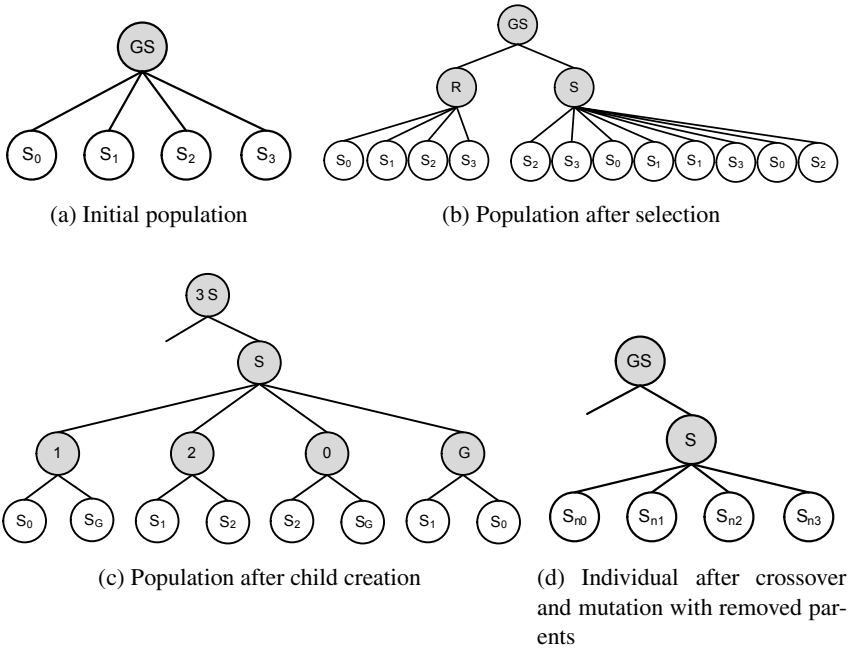
In the following it will be outlined how a genetic algorithm can be implemented using HeuristicLab's operator graph. Based on the above description, the GA needs at least the following operators:

- Operator for creating the initial population
- Crossover operator
- Mutation operator
- Fitness evaluation operator
- Selection operator
- Population update method

The first four operators are problem- or encoding-specific operators. As the genetic algorithm should work with all types of encodings and problems which HeuristicLab offers, place-holders are used for these operators so that the user can configure which operator should be used. The same concept holds for the selection operator, though this operator is not specific to a problem or encoding. HL offers several common selection strategies, crossover and mutation operators which can be configured by the user and be executed by the placeholders. Before discussing the population update method, the representation of individuals and populations is covered in more detail.

Individuals in HL are represented as scopes. Each individual is a subscope containing at least it's genotype and optionally additional values describing the solution

candidate (e.g. the quality). The selection operators create two subscopes from the population. The first subscope contains the old population while the second subscope contains the selected individuals. Crossover operators in HeuristicLab assume that they are applied to a subscope containing the two indivduals to be crossed. Therefore a *ChildrenCreator* is used to divide the selected subscope into groups of two individuals. To these subscopes the crossover and mutation operators can then be easily applied. Additionally the two parents have to be removed after a new individual was generated as they are not needed any more. The population update method has to delete the subscope of the old population and extract the newly generated individuals from the second (selected) subscope. Figure 10.13 outlines how the subscopes change over the course of the selection, crossover, mutation and population update methods.



(a) Initial population

(b) Population after selection

(c) Population after child creation

(d) Individual after crossover and mutation with removed parents

Fig. 10.13. Changes on scope tree during GA execution

Figure 10.13a shows the initial population which is created beneath the root of the scope tree (Global Scope - GS). Figure 10.13b depicts the scope tree after applying a selection operator. It shows that the population is now divided into the remaining (R) and the selected (S) population. The *ChildrenCreator* operator then introduces subscopes in the selected subscope that contain two individuals (Figure 10.13c). After crossover and mutation, the parents are removed with the *SubScopesRemover* and the remaining scopes contain the new solutions (Figure 10.13d). After having

generated a new population the old population is removed and the new population moved back into the global scope to produce the original structure shown in Figure 10.13a. This task is accomplished with the previously described *RightReducer*.

Figure 10.14 shows the operator graph for generating the offspring in the genetic algorithm.

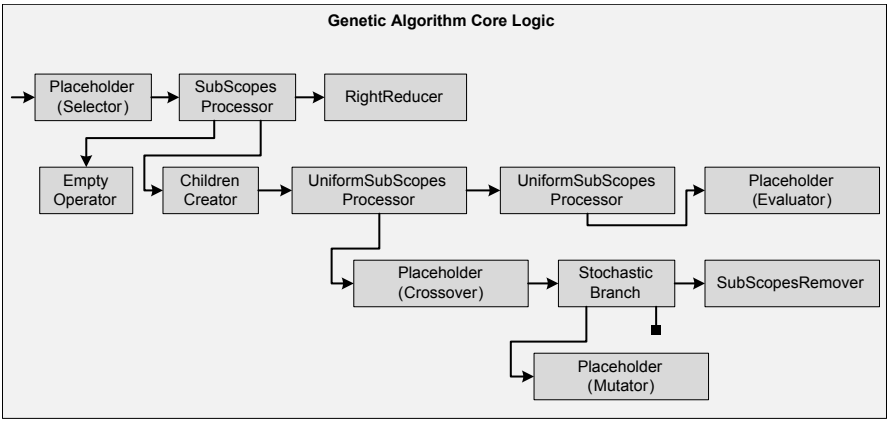


Fig. 10.14. Offspring generation in the genetic algorithm

After the selection operation a *SubScopesProcessor* is used to apply the *ChildrenCreator* to the selected individuals. An *UniformSubScopesProcessor* is used to apply the crossover to each group of parents. After the crossover the *StochasticBranch* chooses based on the mutation rate, if the mutation operator should be applied to the newly generated offspring. After the mutation the parents are removed with the help of the *SubScopesRemover* operator. Next a quality is assigned to the offspring using the *Evaluator*. The last step of the core logic is to remove the old population and move the offspring to the global scope.

The algorithm so far represents the steps that one iteration of the GA is made up of. Figure 10.15 depicts the loop that applies the offspring generation method until the maximum number of generations is reached as well as the initialization of the algorithm.

After initializing the random number generator, a variable *Generations* is created and added to the results collection. It is incremented by the *IntCounter* every generation and should be displayed on the results page of HL to show the progress of the algorithm. After incrementing *Generations*, a *Comparator* is used to check if *Generations* has reached it's allowed maximum. If *MaximumGenerations* is reached, the variable *Abort* is set to true and the *ConditionalBranch* executes the true branch, which is empty and therefore the algorithm stops.

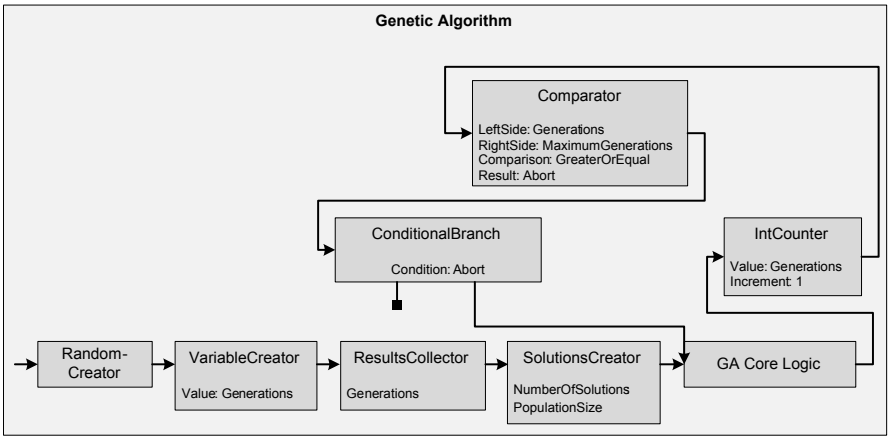


Fig. 10.15. Initialization and main loop of the genetic algorithm

10.4.3 Modeling Simulated Annealing

As stated in [4], simulated annealing (SA) [24] is commonly said to be the oldest among the metaheuristics and one of the first algorithms that contained an explicit strategy to escape from local optima. As greedy local search algorithms head for the optimum located in the attraction basin of the initial solution, they severely suffer from the problem of getting stuck in a local optimum. To overcome this problem many heuristic optimization algorithms use additional strategies to support diversification of the search. One of these algorithms is simulated annealing which additionally introduces an acceptance probability. If a worse solution is selected in the solution space, it is accepted with some probability depending on the quality difference of the actual (better) and the new (worse) solution and on a parameter called temperature; to be more precise, the higher the temperature and the smaller the quality difference, the more likely it is that a worse solution is accepted. In each iteration the temperature is continuously decreased leading to a lower and lower acceptance probability so that the algorithm converges to an optimum in the end. A detailed description of SA is given in Listing 10.5.

SA starts with an initial solution s which can be created randomly or using some heuristic construction rule. A solution s' is randomly selected from the neighborhood of the current solution in each iteration. If this solution is better, it is accepted and replaces the current solution. However, if s' is worse, it is not discarded immediately but is also accepted with a probability depending on the actual temperature parameter t and the quality difference. The way how the temperature is decreased over time is defined by the cooling scheme. Due to this stochastic acceptance criterion the temperature parameter t can be used to balance diversification and intensification of the search.

```
1  s ← new random solution // starting point
2  evaluate s
3  s_best ← s // best solution found so far
4  i ← 1 // number of evaluated solutions
5  t ← t_i // initial temperature
6
7  WHILE i ≤ maxSolutions DO BEGIN
8      s' ← manipulate s // get solution from neighborhood
9      evaluate s'
10     i ← i + 1
11     q ← quality difference of s and s'
12     IF s' is better than s THEN BEGIN
13         s ← s'
14         IF s is better than s_best THEN BEGIN
15             s_best ← s
16         END IF
17     END
18     ELSE IF Random(0,1) < e^(-|q|/t) BEGIN
19         s ← s'
20     END IF
21     t ← t_i // calculate next temperature
22 END WHILE
23
24 RETURN s_best
```

Listing 10.5. Simulated annealing

Figure 10.16 shows the operator graph for generating solutions in simulated annealing.

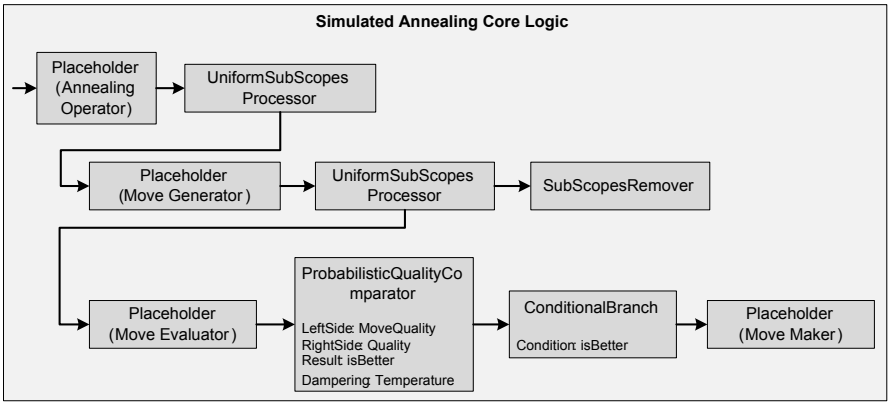


Fig. 10.16. Solution generation in simulated annealing

The annealing operator defines the cooling scheme and is modeled with a placeholder as HL offers different cooling schemes. Concrete operators for placeholders can be provided by plugins and selected by the user through the GUI. In contrast to the GA the global scope of the algorithm contains not a population of solution candidates but only one solution. A *UniformSubScopesProcessor* is used to apply a *Move Generator* to this solution. Move generators are operators that generate a configurable amount of solution candidates for a given solution. A placeholder is used for the *Move Generator* again, so that the user can decide how to generate moves. Each new solution candidate is then evaluated with the *Move Evaluator*. The *ProbabilisticQualityComparator* calculates whether the current quality has gained a better quality than the so far found best solution. The operator additionally considers the current temperature as well as a random number (depicted in Listing 10.5). The following *ConditionalBranch* decides based on the result of the *ProbabilisticQualityComparator* if the global solution has to be updated. If that is the case the update is performed by the *Move Maker*. If all generated solution candidates have been evaluated, a *SubScopesRemover* is used to dispose them.

As with genetic algorithms, simulated annealing's solution generation method is applied repeatedly and the number of iterations can be defined by the user. Figure 10.17 shows the main loop of the SA algorithm and the initialization operators.

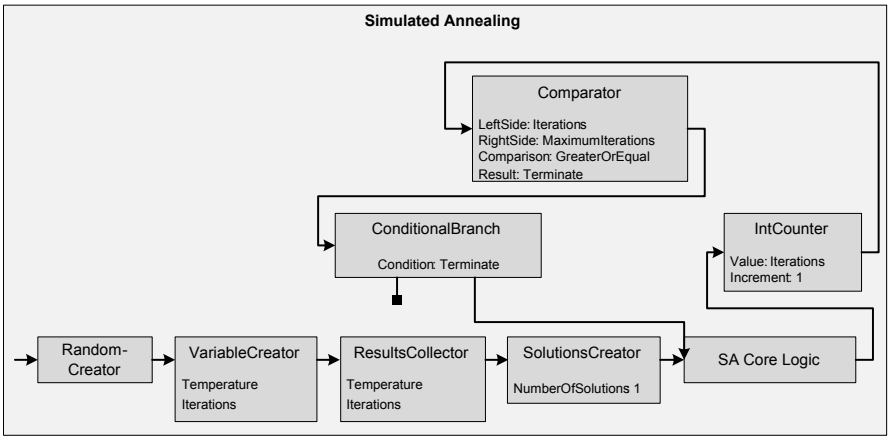


Fig. 10.17. Initialization and main loop of simulated annealing

After initializing the random number generator, two variables, *Temperature* and *Iterations*, are created and added to the results collection. *Temperature* is used in the core algorithm to save the result of the cooling scheme operator as well as for deciding if the quality of solution candidates is better than the global best solution. *Iterations* is incremented after each solution generation phase and is compared to the maximum number of allowed iterations. If *MaximumIterations* is reached, the algorithm is stopped. *SolutionsCreator* creates only one solution which is used for

generating moves and is continuously improved over the course of the algorithm execution.

10.5 Problem Modeling

After having defined a set of generic operators in Section 10.4 which build the basis for every heuristic optimization algorithm, in the next sections the focus is on problem-specific aspects such as solution encoding, quality evaluation and manipulation operators. Figure 10.18 gives an overview of how problems and encodings are organized in HL.

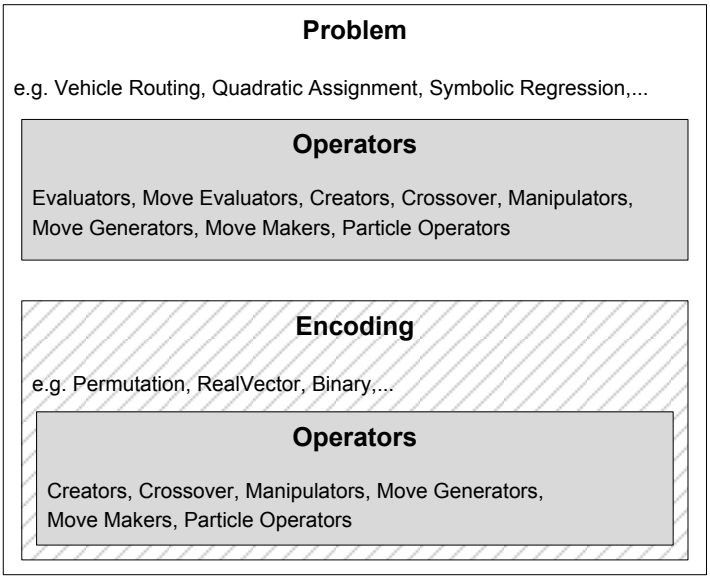


Fig. 10.18. Problems, encodings and problem/encoding-specific operators in HeuristicLab

HeuristicLab offers a range of already implemented problems. These problems use encodings to represent solution candidates. Problems and encodings offer various operators that a user can choose from. As seen in Section 10.4 algorithms usually are modeled to allow configuration of certain operators. These placeholders can be filled with encoding- or problem-specific operators. In addition to the representation of a solution, encodings usually offer operators for creating solution candidates, crossover and mutation (*Manipulator*) operators as well as operators for trajectory-based metaheuristics (*Move Generators* and *Move Makers*) or particle swarm optimization. Problems hold the problem information (e.g. the distance matrix of a TSP) and offer problem-specific operators. Primarily these operators are the *Evaluators*

and *Move Evaluators* for assigning a fitness value to a solution candidate. Of course problems can also offer the same operators as the encodings if there is a need for incorporating problem-specific information.

In the following, three concrete problem implementations in HeuristicLab are presented in more detail, showing how HL can be used to represent the quadratic assignment problem, to optimize simulations, or to do genetic programming.

10.5.1 Quadratic Assignment Problem

The Quadratic Assignment Problem (QAP) was introduced in [26] and is a well-known problem in the field of operations research. It is the topic of many studies, treating the improvement of optimization methods as well as reporting successful application to practical problems in keyboard design, facility layout planning and re-planning as well as in circuit design [18, 7]. The problem is NP hard in general and, thus, the best solution cannot easily be computed in polynomial time. Many different optimization methods have been tried, among them popular metaheuristics such as tabu search [36] and genetic algorithms [10].

The problem can be described as finding the best assignment for a set of facilities to a set of locations so that each facility is assigned to exactly one location which in turn houses only this facility. The problem can also model situations in which there are a greater number of locations available. By introducing and assigning facilities with no flows a solution indicates which locations are to be selected. Likewise the problem can also model situations that contain a larger set of facilities by introducing locations with very high distances. Generally, an assignment is considered better than another when the flows between the assigned facilities run along smaller distances.

More formally the problem can be described by an $N \times N$ matrix W with elements w_{ik} denoting the weights between facilities i and k and an $N \times N$ matrix D with elements d_{xy} denoting the distances between locations x and y. The goal is to find a permutation π with $\pi(i)$ denoting the location that facility i is assigned to so that the following objective is achieved:

$$\min \sum_{i=1}^{N} \sum_{k=1}^{N} w_{ik} \cdot d_{\pi(i)\pi(k)} \tag{10.1}$$

A permutation is restricted to contain every number just once, hence, it satisfies the constraint of a one-to-one assignment between facilities and locations:

$$\forall_{i,k} i \neq k \Leftrightarrow \pi(i) \neq \pi(k) \tag{10.2}$$

The complexity of evaluating the quality of an assignment according to Eq. (10.1) is $O(N^2)$, however several optimization algorithms move from one solution to another through small changes, such as by swapping two elements in the permutation. These moves allow to reduce the evaluation complexity to $O(N)$ and even $O(1)$ if the

previous qualities are memorized [36]. Despite changing the solution in small steps iteratively, these algorithms can, nevertheless, explore the solution space and interesting parts thereof quickly. The complete enumeration of such a "swap" neighborhood contains $N*(N-1)/2$ moves and, therefore, grows quickly with the problem size. This poses a challenge for solving larger instances of the QAP.

HeuristicLab offers an implementation of the QAP in the form of the *QuadraticAssignmentProblem* plugin that is based on the *PermutationEncoding* plugin. The *Permutation* is used as a representation to the QAP. Besides the solution vector a QAP solution additionally holds the quality assigned by the *QAPEvaluator* operator.

HeuristicLab also provides access to all instances of the Quadratic Assignment Problem Library (QAPLIB) [5] which is a collection of benchmark instances from different contributors. According to the QAPLIB website[6], it originated at the Graz University of Technology and is now maintained by the University of Pennsylvania, School of Engineering and Applied Science. It includes the instance descriptions in a common format, as well as optimal and best-known solutions or lower bounds and consists of a total of 137 instances from 15 contributing sources which cover real-world as well as random instances. The sizes range from 10 to 256 although smaller instances are more frequent. Despite their small size these instances are often hard to solve so a number of different algorithms have emerged [36, 35, 10].

The *QuadraticAssignmentProblem* class in HeuristicLab extends from the *SingleObjectiveHeuristicOptimizationProblem* class provided by the HeuristicLab 3.3 framework. Besides support for single and multi objective problems HeuristicLab also distinguishes between problems and heuristic optimization problems. *IProblem* specifies that each problem has to hold a list of operators that can be queried. Problems are therefore responsible for discovering encoding and problem specific operators that algorithms can retrieve and use. *IHeuristicOptimizationProblem* defines that a heuristic optimization problem has to contain at least parameters for an evaluator and a solution creator that an algorithm can use to create and evaluate solution candidates. *ISingleObjectiveHeuristicOptimizationProblem* further specifies that the evaluator has to be an *ISingleObjectiveEvaluator* in contrast to an *IMultiObjectiveHeuristicOptimizationProblem* that requires an *IMultiObjectiveEvaluator* which generates multiple fitness values.

The QAP itself holds the matrices for the distances between the locations and the weights between the facilities. These are parameters that are used by the QAPEvaluator to compute the fitness value of a solution candidate. The plugin also provides move evaluators such as the *QAPSwap2MoveEvaluator* that evaluates the quality of a swap move. This move evaluator has a special method that allows to compute the move quality in O(1) as described in [36]. It requires that the complete swap neighborhood is evaluated each iteration and that the move qualities from the previous iteration, as well as the previously selected move are remembered. The *QuadraticAssigmentProblem* class also implements interfaces which allow to parameterize it with benchmark instances from various libraries. As described above the QAPLIB is

[6] `http://www.seas.upenn.edu/qaplib/`

already integrated in HeuristicLab, but as the QAP is a generalization of the Traveling Salesman Problem (TSP) it can be parameterized with problems from the according benchmark instance library TSPLIB[33]. For this purpose the problem class acts as an *IProblemInstanceConsumer* for *QAPData* and *TSPData*. Figure 10.19 shows a screenshot of the QAP implementation in HL with the chr12a problem instance loaded from the QAPLIB.

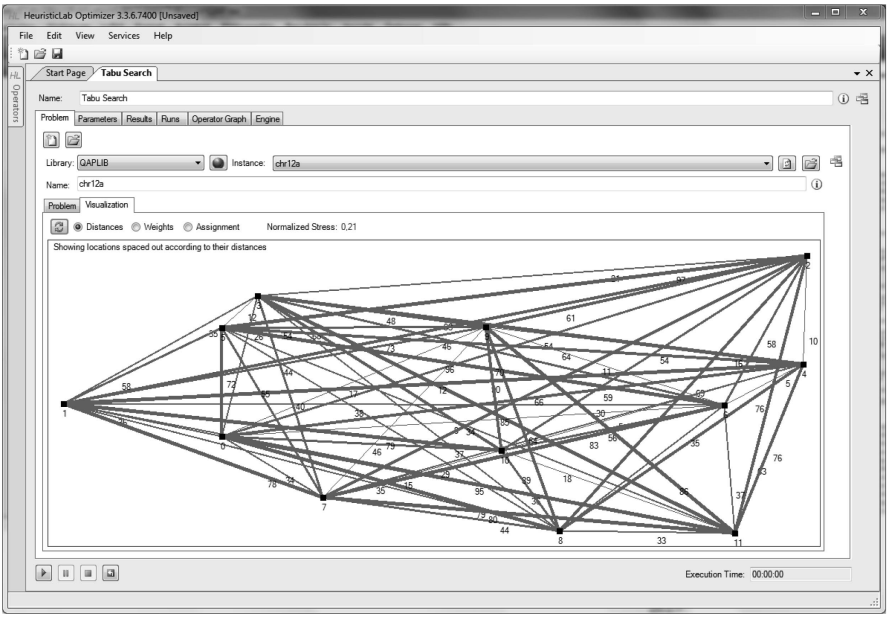


Fig. 10.19. QAP in HeuristicLab showing a QAPLIB instance. The layout of the locations is obtained by performing multi-dimensional scaling on the distances matrix.

10.5.2 Simulation-Based Optimization

The optimization of simulation parameters has been discussed before [12, 11], however there is still a lack of widely available ready-to-use optimization systems that can be employed to perform the optimization task. Some simulation frameworks come with optimizers such as OptQuest [16] included, but even in this case the included optimizer might not be well suited to solve the given problem. The cause for this can be found in the *no free lunch theorem* [50] which states that no algorithm can outperform any other algorithm on all problems. Due to the high heterogeneity of the actual simulation models and the meaning of their parameters, the problems can be of many different natures. In one case there might be only a single optimal solution that can be obtained through a basic local search algorithm, but in other cases the model response might be more complex and different strategies are

required with different levels of diversification and intensification. In any case, the more solutions that need to be evaluated, the longer the optimization process will last. The selection of a suited optimization method and suited algorithm parameters is therefore crucial to find good parameters for the given simulation model.

Generally, two different cases of simulation-based optimization can be identified. In the first case the simulation model acts as a fitness function, it will take a number of parameters and calculate the resulting fitness value. In the second case, the optimization problem occurs within the simulation model itself. For example, the simulation of a production facility might require to solve a scheduling problem to determine the best order of job executions which in turn requires the integration of an optimization approach. HeuristicLab has been used in both cases [32, 39] successfully, but while the first case has been generalized and abstracted as shown in this work, the second case still requires a tighter coupling with the simulation model. The generalization and abstraction of the second case is a topic for future work.

External Evaluation Problem

Simulation-based optimization in HeuristicLab has been integrated in the form of the *ExternalEvaluationProblem*. As the name implies it assumes the evaluation is taking place in another application. This problem has no predefined representation or operators, instead the user can customize the problem according to his or her needs. If the actual simulation-based optimization tasks can be represented by a set of real-valued parameters, the *RealVectorEncoding* plugin and its operators can be added to the problem. If instead the parameters are integer values, the *IntegerVectorEncoding* plugin can be used to create and modify the solutions. Both encodings can also be combined if the problem parameters are mixed. A screenshot of the problem configuration view is given in Figure 10.20. In the following the parameters of this problem are explained.

BestKnownQuality and *BestKnownSolution:* These parameters are used and updated by certain analyzers and remember the best quality that has been found so far as well as the corresponding best solution.
Cache: Together with the appropriate evaluation operator this parameter can be used to add an evaluation cache. The cache stores already seen configurations and their corresponding quality so that these need not be simulated again.
Clients: Contains a list of clients in the form of communication channels. At least one must be specified, but the list can contain multiple channels if the simulation model is run on multiple machines.
Evaluator: This operator is used to collect the required variables, packs them into a message, and transmits them to one of the clients. If the cached evaluator is used the cache will be filled and the quality of previously seen configurations will be taken directly from the cache.
Maximization: This parameter determines if the received quality values should be maximized or minimized.

Operators: This list holds all operators that can modify and process solutions such as for example crossover and mutation operators. Any operator added to the list can be used in an algorithm and certain algorithms might require certain operators to work.
SolutionCreator: This operator is used to create the initial solution, typically it randomly initializes a vector of a certain length and within certain bounds.

Interoperability

The representation of solutions as scope objects which contain an arbitrary number of variables and the organization of scopes in a tree provides an opportunity for integrating a generic data exchange mechanism. Typically, an evaluator is applied on the solution scope and calculates the quality based on some variables that it would expect therein. The evaluator in the ExternalEvaluationProblem will however collect a user specified set of variables in the solution scope, and adds them to the *SolutionMessage*. This message is then transmitted to the external application for evaluation. The evaluation operator then waits for a reply in form of the *QualityMessage* which contains the quality value and which can be inserted into the solution scope again. This allows to use any algorithm that can optimize single-objective problems in general to optimize the ExternalEvaluationProblem. The messages in this case are protocol buffers[7] which are defined in a .proto file. The structure of these messages is shown in Listing 10.6.

The protocol buffer specification in form of the message definitions is used by a specific implementation to generate a tailored serializer and deserializer class for each message. The format is designed for very compact serialized files that do not impose a large communication overhead and the serialization process is quick due to the efficiency of the specific serialization classes. Implementations of protocol buffers are provided by Google for Java™, C++, and Python, but many developers have provided open source ports for other languages such as C#, Clojure, Objective C, R, and many others[8]. The solution message buffer is a so called "union type", that means it provides fields for many different data types, but not all of them need to be used. In particular there are fields for storing Boolean, integers, doubles, and strings, as well as arrays of these types, and there is also a field for storing bytes. Which data type is stored in which field is again customizable. HeuristicLab uses a *SolutionMessageBuilder* class to convert the variables in the scope to variables in the solution message. This message builder is flexible and can be extended to use custom converters, so if the user adds a special representation to HeuristicLab a converter can be provided to store that representation in a message. By default, if the designer of an ExternalEvaluationProblem would use an integer vector, it would be stored in an integer array variable in the solution message. The simulation model can then extract the variable and use it to set its parameters. The protocol buffer

[7] `http://code.google.com/p/protobuf`
[8] `http://code.google.com/p/protobuf/wiki/ThirdPartyAddOns`

is also extensible in that new optional fields may be added at a later date. Finally, transmission to the client is also abstracted in the form of *channels*. The default channel is based on the transmission control protocol (TCP) which will start a connection to a network socket that is opened by the simulation software. The messages are then exchanged over this channel.

Parallelization and Caching

If the required time to execute a simulation model becomes very long, users might want to parallelize the simulation by running the model on multiple computers. In HeuristicLab this is easily possible through the use of the *parallel engine*. The parallel engine allows multiple evaluation operators to be executed concurrently which in turn can make use of multiple channels defined in the *Clients* parameter. It makes use of the *TaskParallelLibrary* in Microsoft® .NET which manages the available threads for efficient operations. To further speed up the optimization the user can add aforementioned *EvaluationCache* and the respective evaluator. The cache can be persisted to a file or exported as a comma-separated-values (CSV) file for later analysis [32].

10.5.3 Genetic Programming

HeuristicLab includes an implementation of tree-based genetic programming and and extensive support for symbolic regression and classification. This implementation will be described in the following sections.

10.5.3.1 Symbolic Expression Tree Encoding

The central part of our implementation of tree-based GP is the symbolic expression tree encoding. It defines the structure of individuals and provides problem-independent classes that can be reused for GP problems. All standard methods for tree creation, manipulation, crossover and compilation are located in the encoding and therefore a newly implemented problem just has to provide concrete symbols and methods to evaluate solution candidates. For example, in the case of a symbolic regression problem, the evaluator calculates the error of the model predictions and uses an interpreter to calculate the output of the formula for each row of the dataset.

Any algorithm that uses recombination and mutation operators to generate new solution candidates, for instance a genetic algorithm (GA), can be used to solve any problem using the symbolic expression tree encoding for instance a symbolic regression problem. A specialized algorithm for genetic programming with reproduction and crossover probability is not yet provided but is planned to be added soon.

```
message SolutionMessage {
  message IntegerVariable {
    required string name = 1;
    optional int32 data = 2;
  }
  message IntegerArrayVariable {
    required string name = 1;
    repeated int32 data = 2;
    optional int32 length = 3;
  }
  //... similar sub-messages omitted for brevity ...
  message RawVariable {
    required string name = 1;
    optional bytes data = 2;
  }
  required int32 solutionId = 1;
  repeated IntegerVariable integerVars = 2;
  repeated IntegerArrayVariable integerArrayVars = 3;
  repeated DoubleVariable doubleVars = 4;
  repeated DoubleArrayVariable doubleArrayVars = 5;
  repeated BoolVariable boolVars = 6;
  repeated BoolArrayVariable boolArrayVars = 7;
  repeated StringVariable stringVars = 8;
  repeated StringArrayVariable stringArrayVars = 9;
  repeated RawVariable rawVars = 10;
}

message QualityMessage {
  required int32 solutionId = 1;
  required double quality = 2;
}
```

Listing 10.6. Definition of the generic interface messages

10.5.3.2 Symbolic Expression Trees

The most important interfaces of the symbolic expression tree encoding are: *ISymbolicExpressionTree*, *ISymbolicExpressionTreeNode*, and *ISymbol*.

The structure of a tree is defined by linked nodes, and the semantic is defined by symbols attached to these nodes. The *SymbolicExpressionTree* represents trees and provides properties for accessing the root node, getting its length and depth, and iterating all tree nodes. Every node of a tree can be reached beginning with the root node as the *ISymbolicExpressionTreeNode* provides properties and methods to manipulate its parent, its subtrees and its symbol. A GP solution candidate is

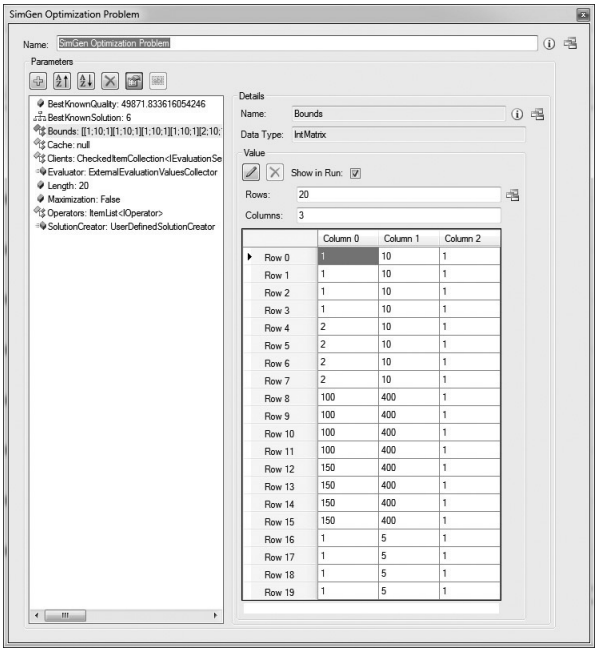


Fig. 10.20. Example of a simulation-based optimization problem configuration in HeuristicLab 3.3 with multiple bounds shown for each dimension of an integer parameter vector. The first column denotes the minimum value, the second column the maximum value, and the step size can be given in the third column.

therefore simply created by building a tree by linking tree nodes beginning with the root node.

The root node of a tree always contains the *ProgramRootSymbol* and must have a child node with the *StartSymbol*. This convention is necessary to support ADFs which are defined as additional sub-trees of the root node. The root node of ADF definitions contains a *DefunSymbol* (cf. Section 10.5.3.4).

10.5.3.3 Symbols and Grammars

In addition to the structure of a tree, symbols and grammars are necessary for individual creation. A symbol defines the semantic of a tree node (how it is interpreted) and specifies a minimum and maximum arity; terminal symbols have a minimum and maximum arity of zero. The set of available symbols must be defined with a grammar from which the set of valid and well-formed trees can be derived. We have chosen to implement this by defining which *Symbols* are allowed as child symbol of

other symbols, and at which position they are allowed. For example, the first child of a conditional symbol must either be a comparison symbol or a Boolean function symbol.

Problem-specific grammars should derive from the base class for grammars which is provided by the encoding and define the rules for allowed tree structures. The initialization of an arithmetic expression grammar is shown in Listing 10.8. The statements in lines 1–10 create the symbols and add them to lists for easier handling. Afterwards the created symbols are added to the grammar (lines 12–13) and the number of allowed subtrees is set to two for all function symbols (lines 14–15). Terminal symbols do not have to be configured, because the number of allowed subtrees is automatically set to zero. The last lines define which symbols are allowed at which position in the tree. Below the *StartSymbol* all symbols are allowed (lines 16 and 17) and in addition, every symbol is allowed under a function symbol (Lines 18–21).

```
1 <expr> := <expr> <op> <expr> | <terminal>
2 <op>   := + | - | / | *
3 <terminal>  := variable | constant
```

Listing 10.7. Backus-Naur Form of an arithmetic grammar defining symbolic expression trees to solve a regression problem.

Default grammars are implemented and pre-configured for every problem which can be solved by GP. These grammars can be modified within the GUI to change the arity of symbols or to enable and disable specific symbols.

A typical operator for tree creation first adds the necessary nodes with the *RootSymbol* and the *StartSymbol* and afterwards uses one of the allowed symbols returned by the *Grammar* as the starting point for the result producing branch. This procedure is recursively applied to extend the tree until the desired size is reached. In addition, the crossover and mutation operators also adhere to the rules defined by the grammar so during the whole algorithm run only valid and well-formed trees are produced.

10.5.3.4 Automatically Defined Functions

The GP implementation of HeuristicLab also supports automatically defined functions (ADFs). ADFs are program subroutines that provide code encapsulation and reuse. They are not shared between individuals but have to be evolved separately in individuals, either by crossover or mutation events and are numbered according to their position below the tree root. The *Defun* tree node and symbol define a new

```
var add = new Addition();
var sub = new Subtraction();
var mul = new Multiplication();
var div = new Division();
var constant = new Constant();
var variableSymbol = new Variable();
var allSymbols = new List<Symbol>()
 {add,sub, mul,div,constant,variableSymbol};
var funSymbols = new List<Symbol>()
 {add,sub,mul,div};

foreach (var symb in allSymbols)
  AddSymbol(symb);
foreach (var funSymb in funSymbols)
  SetSubtreeCount(funSymb, 2, 2);
foreach (var symb in allSymbols)
  AddAllowedChildSymbol(StartSymbol, symb);
foreach (var parent in funSymbols) {
  foreach (var child in allSymbols)
    AddAllowedChildSymbol(parent, child);
}
```

Listing 10.8. Source code for the configuration of the *ArithmeticGrammar* formally defined in Listing 10.7.

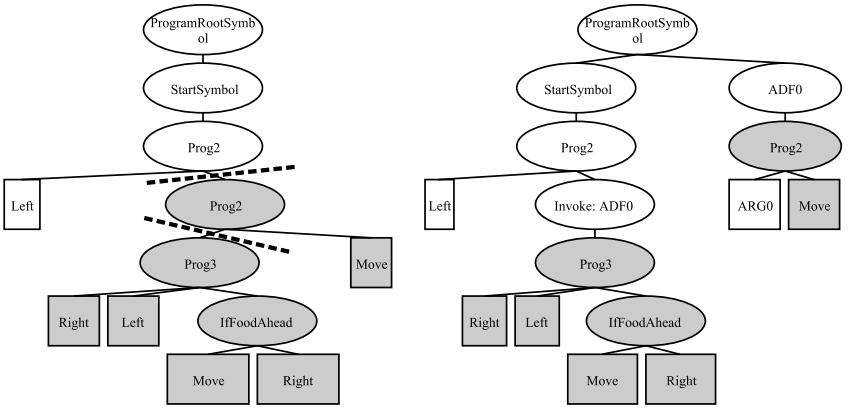


Fig. 10.21. Creation of an ADF in an artificial ant program. The dashed lines indicate the cut points in the tree from which the ADF is created.

subroutine and are used next to the *StartSymbol* directly below the *ProgramRoot-Symbol*. ADFs can be called through the *InvokeFunction* symbol from point in the symbolic expression tree, except from ADFs with a lower index to prevent infinite recursions and non-stopping programs.

ADFs are created during algorithm execution either by the subroutine creator or by the subroutine duplicator. The subroutine creator moves a subtree of an individual into a subroutine and inserts an *InvokeFunctionTreeNode* instead of the original subtree. Furthermore, ADFs can have an arbitrary number of arguments that are used to parameterize the subroutines. An example for creating an ADF with one argument is shown in Figure 10.21. On the left hand side the original tree describing an artificial ant program is displayed. Additionally, two cut points for the ADF extraction are indicated by dashed lines. The subtree between the cut points is added beneath a *DefunTreeNode* displayed as the newly defined ADF *ADF0* and as a replacement an *InvokeFunctionTreeNode* is inserted. The subtree below the second cut point is left unmodified and during interpretation its result is passed to *ADF0* as the value of *ARG0*.

Architecture altering operators for subroutine and argument creation, duplication, and deletion are provided by the framework. All of these work by moving parts of the tree to another location, either in the standard program execution part (below the *StartTreeNode*) or into an ADF (below the *DefunTreeNode*). For example, the subroutine deletion operator replaces all tree nodes invoking the affected subroutine by the body of the subroutine itself and afterwards deletes the subroutine from the tree by removing the *DefunTreeNode*. All architecture altering operators can be called in place of mutation operators as described by Koza, however in contrast to mutation operators, architecture altering operators preserve the semantics of the altered solution.

The combination of architecture altering operators and tree structure restrictions with grammars is non-trivial as grammars must be dynamically adapted over time. Newly defined ADFs must be added to the grammar; however, the grammar of each single tree must be updated independently because ADFs are specific to trees. This has led to a design where tree-specific grammars contain dynamically extended rules and extend the initially defined static grammar. The combination of the tree-specific grammar and the static grammar defines the valid tree structures for each solution and also for its child solutions because grammars must be inherited by child solutions. If ADFs are not allowed the tree-specific grammar is always empty because no symbols are dynamically added during the run.

Architecture manipulating operators automatically update the tree-specific grammar correctly by altering the allowed symbols and their restrictions. This mechanism allows to implement crossover and mutation operators without special cases for ADFs.

10.5.3.5 Symbolic Regression

In this section the implementation for evaluating symbolic regression models represented as symbolic expression trees is described. Symbolic regression is frequently used as a GP benchmark task for testing new algorithmic concepts and ideas. If symbolic regression is applied to large real-world datasets with several thousand data rows, performance as well as memory efficiency becomes an important issue.

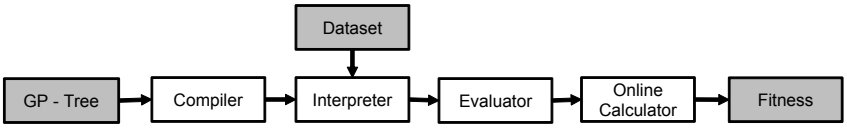


Fig. 10.22. Workflow for calculating the fitness of symbolic regression models

The main concepts for the symbolic regression evaluation in HeuristicLab are streaming and lazy evaluation provided by constructs of the Microsoft® .NET framework. Figure 10.22 depicts how the symbolic expression trees are passed through different operators for fitness value calculation, which is explained in the following sections.

Interpretation of Trees

Evaluators for symbolic regression calculate the error of the predicted values of the model and the actual target values. To prevent the allocation of large double arrays we implemented an interpreter for symbolic regression models that yields a lazy sequence of predicted values for a given model, a dataset and a lazy sequence of row indexes. As a preparatory step the interpreter first compiles the model represented as a symbolic expression tree down to an array of instructions. This preparation can be done in a single pass over all nodes of the tree so the costs are rather small and the linear instruction sequence can be evaluated much faster. First of all the nodes of the original tree are scattered on the heap, while the instruction array is stored in a continuous block of memory. Additionally, the instructions have a small memory footprint as they consist of a single byte for the operation code (opcode), a byte for the number of arguments of the function and an object reference which can hold additional data for the instruction. As a result the instruction array is much more cache friendly and the number of cache misses of tree interpretation can be reduced. Another benefit of the compilation step is that simple static optimizations for instance constant folding can be applied.

The interpretation of the instruction array is implemented with a simple recursive *Evaluate* method containing a large switch statement with handlers for each opcode. Listing 10.9 shows an excerpt of the evaluation method with the handlers for the opcodes for addition, division and variable symbols. The recursive evaluation method is rather monolithic and contains all necessary code for symbol evaluation. This goes against fundamental OO design principles, however, the implementation as a single monolithic switch loop with recursive calls is very efficient as no virtual calls are necessary, the switch statement can be compiled down to a relative jump instruction, and the arguments are passed on the runtime stack which again reduces cache misses.

```
double Evaluate(Dataset ds, State state) {
  var curInstr = state.NextInstruction();
  switch (curInstr.opCode) {
    case OpCodes.Add: {
      double s = Evaluate(dataset, state);
      for (int i = 1; i < curInstr.nArgs; i++) {
        s += Evaluate(dataset, state);
      }
      return s;
    }
    // [...]
    case OpCodes.Div: {
      double p = Evaluate(dataset, state);
      for (int i = 1; i < curInstr.nArgs; i++) {
        p /= Evaluate(dataset, state);
      }
      if (curInstr.nArgs == 1) p = 1.0 / p;
      return p;
    }
    // [...]
    case OpCodes.Variable: {
      if (state.row < 0 || state.row >= dataset.Rows)
        return double.NaN;
      var varNode = (VariableTreeNode)curInstr.dynamicNode;
      var values = ((IList<double>)curInstr.iArg0)
      return values[state.row];
    }
  }
}
```

Listing 10.9. Excerpt of the evaluation method for symbolic regression models showing handlers for the addition, division and variable opcodes.

An alternative design would be to implement a specific evaluation method in each symbol class. This would be the preferable way regarding readability and maintainability. However, with this alternative design a costly indirect virtual call would be necessary for each node of the tree and for each evaluated row of the dataset.

In addition to the recursive interpreter, HeuristicLab also provides an interpreter implementation that compiles symbolic expression trees to linear code in intermediate language (IL) using the *Reflection.Emit* framework which can be executed directly by the Microsoft® .NET CLR. This interpreter is useful for large datasets with more than 10,000 rows as the generated IL code is further optimized and subsequently compiled to native code by the framework JIT-compiler. The drawback is that the JIT-compiler is invoked for each evaluated tree and these costs can be amortized only when the dataset has a large number of rows.

Interpretation of ADFs

The interpreter must be able to evaluate trees with ADFs with a variable number of arguments. The instruction for calling ADF uses the *Call* opcode and contains the index of the called ADF and the number of arguments of the ADF. The code fragment for the interpretation of ADFs and function arguments is shown in Listing 10.10. First the interpreter evaluates the subtrees of the *Call* opcode and stores the results. Next the interpreter creates a stackframe which holds the current program counter and the argument values. Stackframes are necessary to support ADFs that subsequently call other ADFs and recusive ADFs. After the stackframe has been created the interpreter jumps to the first instruction of the ADF. When an argument symbol is encountered while interpreting the ADF instructions the interpreter accesses the previously calculated argument values which are stored in the top-most stackframe and returns the appropriate value. This approach to ADF interpretation using precalculated argument values is only possible because symbolic regression expressions do not have side effects. Otherwise, the interpreter would have to jump back to the subtrees of the call symbol for each encountered *ARG* opcode. At the end of the ADF definition the interpreter deletes the top-most stackframe with *RemoveStackFrame* and continues interpretation at the point after the subtrees of the just evaluated *Call* opcode.

Online Evaluation of Programs

The first step for the evaluation of programs is to obtain the dataset on which the trees have to be evaluated on and to calculate the rows that should be used for fitness evaluation. If all samples are to be used, the rows are streamed as an *Enumerable* beginning with the start of the training partition until its end. Otherwise, the row indices to evaluate the tree on, are calculated and yielded by the selection sampling technique [25].

The row indices, together with the dataset and the individual are passed to the interpreter that in fact returns a sequence of numbers. Until now no memory is allocated (except the space required for the iterators) due to the streaming capabilities of the interpreter and the way of calculating row indices. But the whole streaming approach would by pointless if the estimated values of the interpreter were stored in a data structure for fitness calculation. Therefore, all fitness values must be calculated on the fly which is done by *OnlineCalculators*. Such calculators are provided for the mean and the variance of a sequence of numbers and for calculation metrics between two sequences such as the covariance and the Pearson's R^2 coefficient. Further error measures are the mean absolute and squared error, as well as scaled ones, the mean absolute relative error and the normalized mean squared error. *OnlineCalculators* can be nested; for example the *MeanSquaredErrorOnlineCalculator* just calculates the squared error between the original and estimated values and then passes

```
// [...]
case OpCodes.Call: {
  // evaluate subtrees
  var argValues = new double[curInstr.nArgs];
  for (int i = 0; i < curInstr.nArgs; i++) {
    argValues[i] = Evaluate(dataset, state);
  }
  // push on argument values on stack
  state.CreateStackFrame(argValues);

  // save the pc
  int savedPc = state.ProgramCounter;
  // set pc to start of function
  state.PC = (ushort)curInstr.iArg0;
  // evaluate the function
  double v = Evaluate(dataset, state);

  // delete the stack frame
  state.RemoveStackFrame();

  // restore the pc => evaluation will
  // continue at point after my subtrees
  state.PC = savedPc;
  return v;
}
case OpCodes.Arg: {
  return state.GetStackFrameValue(curInstr.iArg0);
}
// [...]
```

Listing 10.10. Code fragment for the interpretation of ADFs and function arguments

the result to the *MeanAndVarianceOnlineCalculator*. The code of the *MeanAndVarianceOnlineCalculator* is presented in Listing 10.11 and in the *Add* method it can be seen how the mean and variance are updated, when new values are added.

The source for calculating the mean squared error of an individual is shown in Listing 10.12, where all the parts described are combined. First the row indices for fitness calculation are generated and the estimated and original values obtained (lines 1-3). Afterwards these values are enumerated and passed to the *OnlineMeanSquaredErrorEvaluator* that in turn calculates the actual fitness.

```
public class OnlineMeanAndVarianceCalculator {
  private double oldM, newM, oldS, newS;
  private int n;

  public int Count { get { return n; } }
  public double Mean {
    get { return (n > 0) ? newM : 0.0; }
  }
  public double Variance {
    get { return (n > 1) ? newS / (n-1) : 0.0; }
  }

  public void Reset() { n = 0; }
  public void Add(double x) {
    n++;
    if(n == 1) {
      oldM = newM = x;
      oldS = newS = 0.0;
    } else {
      newM = oldM + (x - oldM) / n;
      newS = oldS + (x - oldM) * (x - newM);

      oldM = newM;
      oldS = newS;
    }
  }
}
```

Listing 10.11. Source code of the *MeanAndVarianceOnlineCalculator*.

```
var rows = Enumerable.Range(0,trainingEnd);
var estimated = interpreter.GetExpressionValues(tree,
    dataset, rows).GetEnumerator();
var original = dataset.GetDoubleValues(targetVariable,
    rows).GetEnumerator();
var calculator = new OnlineMSECalculator();

while(original.MoveNext() & estimated.MoveNext()) {
  calculator.Add(original.Current, estimated.Current);
}
double meanSquaredError = calculator.MeanSquaredError;
```

Listing 10.12. Source code for calculating the mean squared error between the original values and the estimated values of an individual.

10.6 Conclusion

The main goal of this chapter was to describe the architecture and design of the HeuristicLab optimization environment which aims at fulfilling the requirements of three heterogeneous user groups, namely practitioners, heuristic optimization experts, and students. Three versions of HeuristicLab, referred to as HeuristicLab 1.x, HeuristicLab 2.x, and HeuristicLab 3.x, have been implemented by the authors since 2002 which were discussed in this chapter. By incorporating beneficial features of existing frameworks as well as several novel concepts, especially the most recent version, HeuristicLab 3.x, represents a powerful and mature framework which can be used for the development, analysis, comparison, and productive application of heuristic optimization algorithms. The key innovations of HeuristicLab can be summarized as follows:

- **Plugin-Based Architecture**
  The concept of plugins is used as the main architectural pattern in HeuristicLab. In contrast to other monolithic frameworks, the HeuristicLab main application just provides a lightweight plugin infrastructure. All other parts are implemented as plugins and are loaded dynamically at runtime. This architecture offers a high degree of flexibility. Users can easily integrate custom extensions such as new optimization algorithms or problems by developing new plugins. They do not need to have access to all the source code of HeuristicLab or to recompile the whole application. Furthermore, the modular nature of the plugin-based architecture simplifies the integration into existing software environments, as only the plugins required in a specific optimization scenario have to be deployed.
- **Generic Algorithm Model**
  As there is no unified model for all different heuristic optimization techniques in general, a generic algorithm model is implemented in HeuristicLab that is not restricted to a specific heuristic optimization paradigm. Any kind of algorithm can be represented. To achieve this level of flexibility, algorithms are not implemented as static blocks of code but are defined as operator graphs which are assembled dynamically at runtime. Users can define custom algorithms by combining basic operators for different solution encodings or optimization problems provided by several plugins, or they can add custom operators to integrate specific functionality. Consequently, not only standard trajectory-based or population-based heuristic optimization algorithms but also generic or problem-specific extensions as well as hybrid algorithms can be easily realized.
- **Graphical User Interface**
  As practitioners, students, and also in some cases heuristic optimization experts might not have comprehensive programming skills, a suitable user interface is required to define, execute, and analyze algorithms. Consequently, a graphical user interface (GUI) is integrated in HeuristicLab. According to the model-view-controller pattern, each HeuristicLab object (e.g., operators, variables, or data values) can provide a view to present itself to the user. However, as graphical visualization of objects usually is a performance critical task, these views are

shown and updated on demand. Furthermore, the GUI reduces the required learning effort significantly. Similarly to standard software products, it enables users to apply heuristic optimization algorithms immediately.

- **Parallelism**
  Last but not least parallel execution of algorithms is also respected in HeuristicLab. Dedicated control operators can be used to define parts of an algorithm that should be executed in parallel. These operators can be used anywhere in an algorithm which enables the definition of parallel heuristic optimization methods, as for example global, coarse-grained, or fine-grained parallel GAs. However, how parallelization is actually done does not depend on the operators but is defined when executing an algorithm by choosing an appropriate execution engine. Several engines are provided in HeuristicLab to execute parallel algorithms for example using multiple threads on a multi-core CPU or multiple computers connected in a network.

Since 2002 all versions of HeuristicLab have been extensively used in the research group "Heuristic and Evolutionary Algorithms Laboratory (HEAL)" of Michael Affenzeller for the development of enhanced evolutionary algorithms as well as in several research projects and lectures. The broad spectrum of these applications is documented in numerous publications and highlights the flexibility and suitability of HeuristicLab for the analysis, development, test, and productive use of metaheuristics. A comprehensive description of the research activities of the group can also be found in the book "Genetic Algorithms and Genetic Programming - Modern Concepts and Practical Applications" [1].

However, the development process of HeuristicLab has not come to an end so far. As it is very easy to apply and compare different algorithms with HeuristicLab, it can be quickly identified which heuristic algorithms and which corresponding parameter settings are effective for a certain optimization problem. In order to systematically analyze this information, the authors plan within the scope of the research laboratory "Josef Ressel-Centre for Heuristic Optimization (Heureka!)"[9] to store the results of all algorithm runs executed in HeuristicLab in a large database. The ultimate goal of this optimization knowledge base is to identify correlations between heuristic optimization algorithms and solution space characteristics of optimization problems. This information will provide essential clues for the selection of appropriate algorithms and will also encourage the development of new enhanced and hybrid heuristic optimization algorithms in order to solve problems for which no suitable algorithms are known yet.

Acknowledgements. The work described in this chapter was done within the Josef Ressel-Centre HEUREKA! for Heuristic Optimization sponsored by the Austrian Research Promotion Agency (FFG). HeuristicLab is developed by the Heuristic and Evolutionary Algorithm Laboratory (HEAL)[10] of the University of Applied Sciences Upper Austria. It can be

[9] `http://heureka.heuristiclab.com`

[10] `http://heal.heuristiclab.com/`

downloaded from the HeuristicLab homepage[11] and is licensed under the GNU General Public License.

References

1. Affenzeller, M., Winkler, S., Wagner, S., Beham, A.: Genetic Algorithms and Genetic Programming - Modern Concepts and Practical Applications. In: Numerical Insights. CRC Press (2009)
2. Alba, E. (ed.): Parallel Metaheuristics: A New Class of Algorithms. Wiley Series on Parallel and Distributed Computing. Wiley (2005)
3. Arenas, M.G., Collet, P., Eiben, A.E., Jelasity, M., Merelo, J.J., Paechter, B., Preuß, M., Schoenauer, M.: A framework for distributed evolutionary algorithms. In: Guervós, J.J.M., Adamidis, P.A., Beyer, H.-G., Fernández-Villacañas, J.-L., Schwefel, H.-P. (eds.) PPSN 2002. LNCS, vol. 2439, pp. 665–675. Springer, Heidelberg (2002)
4. Blum, C., Roli, A., Alba, E.: An introduction to metaheuristic techniques. In: Alba, E. (ed.) Parallel Metaheuristics: A New Class of Algorithms, Wiley Series on Parallel and Distributed Computing, ch. 1, pp. 3–42. Wiley (2005)
5. Burkard, R.E., Karisch, S.E., Rendl, F.: QAPLIB – A quadratic assignment problem library. Journal of Global Optimization 10(4), 391–403 (1997), http://www.opt.math.tu-graz.ac.at/qaplib/
6. Cantu-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms. Kluwer (2001)
7. de Carvalho Jr., S.A., Rahmann, S.: Microarray layout as quadratic assignment problem. In: Proceedings of the German Conference on Bioinformatics (GCB). Lecture Notes in Informatics, vol. P-83 (2006)
8. Cox, B.J.: Planning the software industrial revolution. IEEE Software 7(6), 25–33 (1990), http://www.virtualschool.edu/cox/pub/PSIR/
9. DeJong, K.A.: Evolutionary Computation: A Unified Approach. In: Bradford Books. MIT Press (2006)
10. Drezner, Z.: Extensive experiments with hybrid genetic algorithms for the solution of the quadratic assignment problem. Computers & Operations Research 35(3), 717–736 (2008), Part Special Issue: New Trends in Locational Analysis, http://www.sciencedirect.com/science/article/pii/S0305054806001341, doi:10.1016/j.cor.2006.05.004
11. Fu, M., Glover, F., April, J.: Simulation optimization: A review, new developments, and applications. In: Proceedings of the 2005 Winter Simulation Conference, pp. 83–95 (2005)
12. Fu, M.C.: Optimization for simulation: Theory vs. practice. Informs J. on Computing 14(3), 192–215 (2002), http://www.rhsmith.umd.edu/faculty/mfu/fu_files/fu02.pdf
13. Gagné, C., Parizeau, M.: Genericity in evolutionary computation software tools: Principles and case-study. International Journal on Artificial Intelligence Tools 15(2), 173–194 (2006)
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
15. Giffler, B., Thompson, G.L.: Algorithms for solving production-scheduling problems. Operations Research 8(4), 487–503 (1960)

[11] http://dev.heuristiclab.com/

16. Glover, F., Kelly, J.P., Laguna, M.: New advances for wedding optimization and simulation. In: Farrington, P.A., Nembhard, H.B., Sturrock, D.T., Evans, G.W. (eds.) Proceedings of the 1999 Winter Simulation Conference, pp. 255–260 (1999), http://citeseer.ist.psu.edu/glover99new.html
17. Greenfield, J., Short, K.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley (2004)
18. Hahn, P.M., Krarup, J.: A hospital facility layout problem finally solved. Journal of Intelligent Manufacturing 12, 487–496 (2001)
19. Holland, J.H.: Adaption in Natural and Artificial Systems. University of Michigan Press (1975)
20. Johnson, R., Foote, B.: Designing reusable classes. Journal of Object-Oriented Programming 1(2), 22–35 (1988)
21. Jones, M.S.: An object-oriented framework for the implementation of search techniques. Ph.D. thesis, University of East Anglia (2000)
22. Jones, M.S., McKeown, G.P., Rayward-Smith, V.J.: Distribution, cooperation, and hybridization for combinatorial optimization. In: Voß, S., Woodruff, D.L. (eds.) Optimization Software Class Libraries. Operations Research/Computer Science Interfaces Series, vol. 18, ch. 2, pp. 25–58. Kluwer (2002)
23. Keijzer, M., Merelo, J.J., Romero, G., Schoenauer, M.: Evolving Objects: A general purpose evolutionary computation library. In: EA 2001, Evolution Artificielle, 5th International Concerence in Evolutionary Algorithms, pp. 231–242 (2001)
24. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science 220, 671–680 (1983)
25. Knuth, D.E.: The Art of Computer Programming, 3rd edn. Seminumerical Algorithms, vol. 2. Addison-Wesley (1997)
26. Koopmans, T.C., Beckmann, M.: Assignment problems and the location of economic activities. Econometrica, Journal of the Econometric Society 25(1), 53–76 (1957), http://cowles.econ.yale.edu/P/cp/p01a/p0108.pdf
27. Krasner, G.E., Pope, S.T.: A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. Journal of Object-Oriented Programming 1(3), 26–49 (1988)
28. Lenaerts, T., Manderick, B.: Building a genetic programming framework: The added-value of design patterns. In: Banzhaf, W., Poli, R., Schoenauer, M., Fogarty, T.C. (eds.) EuroGP 1998. LNCS, vol. 1391, pp. 196–208. Springer, Heidelberg (1998)
29. McIlroy, M.D.: Mass produced software components. In: Naur, P., Randell, B. (eds.) Software Engineering: Report of a conference sponsored by the NATO Science Committee, pp. 138–155 (1969)
30. Nievergelt, J.: Complexity, algorithms, programs, systems: The shifting focus. Journal of Symbolic Computation 17(4), 297–310 (1994)
31. Parejo, J.A., Ruiz-Cortes, A., Lozano, S., Fernandez, P.: Metaheuristic optimization frameworks: A survey and benchmarking. Soft Computing 16(3), 527–561 (2012)
32. Pitzer, E., Beham, A., Affenzeller, M., Heiss, H., Vorderwinkler, M.: Production fine planning using a solution archive of priority rules. In: Proceedings of the IEEE 3rd International Symposium on Logistics and Industrial Informatics (Lindi 2011), pp. 111–116 (2011)
33. Reinelt, G.: TSPLIB - A traveling salesman problem library. ORSA Journal on Computing 3, 376–384 (1991)
34. Ribeiro Filho, J.L., Treleaven, P.C., Alippi, C.: Genetic-algorithm programming environments. IEEE Computer 27(6), 28–43 (1994)
35. Stützle, T.: Iterated local search for the quadratic assignment problem. European Journal of Operational Research 174, 1519–1539 (2006)

36. Taillard, E.D.: Robust taboo search for the quadratic assignment problem. Parallel Computing 17, 443–455 (1991)
37. Voß, S., Woodruff, D.L.: Optimization software class libraries. In: Voß, S., Woodruff, D.L. (eds.) Optimization Software Class Libraries. Operations Research/Computer Science Interfaces Series, vol. 18, ch. 1, pp. 1–24. Kluwer (2002)
38. Voß, S., Woodruff, D.L. (eds.): Optimization Software Class Libraries. Operations Research/Computer Science Interfaces Series, vol. 18. Kluwer (2002)
39. Vonolfen, S., Affenzeller, M., Beham, A., Wagner, S., Lengauer, E.: Simulation-based evolution of municipal glass-waste collection strategies utilizing electric trucks. In: Proceedings of the IEEE 3rd International Symposium on Logistics and Industrial Informatics (Lindi 2011), pp. 177–182 (2011)
40. Wagner, S.: Looking Inside Genetic Algorithms. Schriften der Johannes Kepler Universität Linz, Reihe C: Technik und Naturwissenschaften. Universitätsverlag Rudolf Trauner (2004)
41. Wagner, S.: Heuristic optimization software systems - Modeling of heuristic optimization algorithms in the HeuristicLab software environment. Ph.D. thesis, Johannes Kepler University, Linz, Austria (2009)
42. Wagner, S., Affenzeller, M.: HeuristicLab Grid - A flexible and extensible environment for parallel heuristic optimization. In: Bubnicki, Z., Grzech, A. (eds.) Proceedings of the 15th International Conference on Systems Science, vol. 1, pp. 289–296. Oficyna Wydawnicza Politechniki Wroclawskiej (2004)
43. Wagner, S., Affenzeller, M.: HeuristicLab Grid. - A flexible and extensible environment for parallel heuristic optimization 30(4), 103–110 (2004)
44. Wagner, S., Affenzeller, M.: HeuristicLab: A generic and extensible optimization environment. In: Ribeiro, B., Albrecht, R.F., Dobnikar, A., Pearson, D.W., Steele, N.C. (eds.) Adaptive and Natural Computing Algorithms, pp. 538–541. Springer, Heidelberg (2005)
45. Wagner, S., Affenzeller, M.: SexualGA: Gender-specific selection for genetic algorithms. In: Callaos, N., Lesso, W., Hansen, E. (eds.) Proceedings of the 9th World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI 2005), vol. 4, pp. 76–81. International Institute of Informatics and Systemics (2005)
46. Wagner, S., Kronberger, G., Beham, A., Winkler, S., Affenzeller, M.: Modeling of heuristic optimization algorithms. In: Bruzzone, A., Longo, F., Piera, M.A., Aguilar, R.M., Frydman, C. (eds.) Proceedings of the 20th European Modeling and Simulation Symposium, pp. 106–111. DIPTEM University of Genova (2008)
47. Wagner, S., Kronberger, G., Beham, A., Winkler, S., Affenzeller, M.: Model driven rapid prototyping of heuristic optimization algorithms. In: Quesada-Arencibia, A., Rodrígue, J.C., Moreno-Diaz Jr., R., Moreno-Diaz, R. (eds.) 12th International Conference on Computer Aided Systems Theory EUROCAST 2009, vol. 2009, pp. 250–251. IUCTC Universidad de Las Palmas de Gran Canaria (2009)
48. Wagner, S., Winkler, S., Pitzer, E., Kronberger, G., Beham, A., Braune, R., Affenzeller, M.: Benefits of plugin-based heuristic optimization software systems. In: Moreno Díaz, R., Pichler, F., Quesada Arencibia, A. (eds.) EUROCAST 2007. LNCS, vol. 4739, pp. 747–754. Springer, Heidelberg (2007)
49. Wilson, G.C., McIntyre, A., Heywood, M.I.: Resource review: Three open source systems for evolving programs - Lilgp, ECJ and Grammatical Evolution. Genetic Programming and Evolvable Machines 5(1), 103–105 (2004)
50. Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation 1(1), 67–82 (1997)