MODELING OF HEURISTIC OPTIMIZATION ALGORITHMS

Stefan Wagner[a], Gabriel Kronberger[b], Andreas Beham[c], Stephan Winkler[d], Michael Affenzeller[e]

[a], [b], [c], [d], [e]Heuristic and Evolutionary Algorithms Laboratory
School of Informatics, Communications and Media – Hagenberg
Upper Austria University of Applied Sciences
Softwarepark 11, A-4232 Hagenberg, Austria

[a]stefan.wagner@heuristiclab.com, [b]gabriel.kronberger@heuristiclab.com, [c]andreas.beham@heuristiclab.com,
[d]stephan.winkler@heuristiclab.com, [e]michael.affenzeller@heuristiclab.com

ABSTRACT
The definition of a generic algorithm model for representing arbitrary heuristic optimization algorithms is one of the most challenging tasks when developing heuristic optimization software systems. As a high degree of flexibility and a large amount of reusable code are requirements that are hard to fulfill together, existing frameworks often lack of either of them to a certain extent. To overcome these difficulties the authors present a generic algorithm model not only capable of representing heuristic optimization but that can be used for modeling arbitrary algorithms. This model can be used as a meta-model for heuristic optimization algorithms, enabling users to represent custom algorithms in a flexible way by still providing a broad spectrum of reusable algorithm building blocks.

Keywords: heuristic optimization, algorithm modeling, software engineering, software tools

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 revealing a processing power one could only dream of a few years ago. This trend opens the door for attacking complex optimization problems from 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 compromise 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 meta-heuristics is a very active field of research as these algorithms represent generic methods that can be used for solving many different optimization problems. Thereby a huge variety of often nature inspired archetypes has been used as a basis for developing new optimization paradigms like evolutionary algorithms, ant systems, particle swarm optimization, tabu search, or simulated annealing. Several publications show successful applications of such meta-heuristics in various problem domains. A recent overview is for example given in (Doerner, Gendreau, Greistorfer, Gutjahr, Hartl, and Reimann 2007).

Today, this broad spectrum of different algorithmic concepts makes it more and more difficult for researchers to compare new algorithms with existing ones to show advantageous properties of some new approach. As most research puts a focus on one particular heuristic optimization paradigm, comparisons with other algorithms are often not quite fair and objective. In many cases a thoroughly optimized algorithm containing cutting-edge concepts is compared with standard and non-optimized algorithms of other paradigms representing research know-how that is several years old.

One of the reasons for these difficulties is that there is no common model for heuristic optimization algorithms in general that can be used to represent, execute and compare arbitrary algorithms. Many existing software frameworks focus on one or a few particular optimization paradigms and miss the goal of providing an infrastructure generic enough to represent all different kinds of algorithms.

In this paper the authors try to overcome this problem by shifting the layer of abstraction one level up. Instead of trying to incorporate different heuristic optimization algorithms into a common model, a generic algorithm (meta-)model is presented that is capable of representing not only heuristic optimization but arbitrary algorithms. By this means the model can be used for developing custom algorithm models for various optimization paradigms. Furthermore, by considering aspects like parallelism or user interaction on different layers of abstraction the presented algorithm model can serve as a basis for development of a next generation heuristic optimization environment that can be used by many researchers to rapidly develop and fairly compare their algorithms.

2. EXISTING SOFTWARE SYSTEMS FOR HEURISTIC OPTIMIZATION

Today, modern concepts of software engineering like 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 (Johnson and Foote 1988). 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 (like word processing, spread sheets, 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 attacking 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 like reusability, flexibility, genericity, documentation and a clean design are not the primer concern of developers. As a consequence, these applications still suffer from a quite low level of maturity seen from a software engineering point of view.

In the last years and with the ongoing success of heuristic algorithms also in 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 identified and discussed in several publications like (Voß and Woodruff 2002; Jones, McKeown, and Rayward-Smith 2002; Gagne and Parizeau 2006). 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 black or white 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 on the one hand a wide range of ready-to-use classical algorithms, solution representations, manipulation operators and benchmark problems which make it easy to jump into the area of heuristic optimization and to start experimenting and comparing various concepts. On the other hand a high degree of flexibility due to a clean object-oriented design makes it easy for users to implement custom extensions like 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. On the one hand 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. On the other hand this model should be very fine granular so that a broad spectrum of existing classical algorithms can be represented in form of algorithm modules. These modules can then serve as building blocks to realize different algorithm variations or completely new algorithms with a high amount of reusable code.

One main 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.

Taking a look at several existing frameworks for heuristic optimization, it can be seen that this question has been answered in quite different ways. For example, in the Templar framework developed by Martin Jones and his colleagues at the University of East Anglia (Jones, McKeown, and Rayward-Smith 2002) a very high level of abstraction has been realized. In Templar each algorithm is represented as so-called engines. Although, the framework supports distribution, hybridization and cooperation of engines, no more fine granular representation of algorithms is considered. Therefore, when a new algorithm with just a slight modification of an existing one is required, the engine of the existing algorithm has to be copied and modified leading to code duplication and less maintainability.

As another example the HotFrame framework developed by Andreas Fink and his colleagues at the University of Hamburg (Fink and Voß 2002) provides a very low level of abstraction. HotFrame contains a large amount of generic C++ classes that can be put together to represent an algorithm. However, in that way the basic algorithm model is more strictly predefined forcing users to fit their custom algorithms into that class structure. Furthermore, due to the complexity of the model the framework also suffers from a quite steep learning curve.

Obviously neither a high nor a low level of abstraction is able to fulfill both, a high degree of flexibility and reusability of code, as these two requirements can be considered as mutually exclusive.

3. GENERIC ALGORITHM MODEL

In order to overcome this problem the authors decided to use a different approach. Instead of trying to develop an algorithm model representing all different kinds of heuristic optimization algorithms, a generic algorithm (meta-)model inspired by classical programming languages is presented in this paper. This model is generic enough to represent not only heuristic optimization techniques but all different kinds of algorithms in general.

On top of this generic algorithm model more specific models for representing heuristic optimization algorithms can be defined. These specific models do not have to be hard-coded in a framework though, but can be defined on the user level. A large variety of different algorithm models can be realized, opening the door for each user to either reuse an existing one or to create an own model if necessary. By shifting the model one layer up, users do not need to fit custom algorithms into a single fixed model but can fit the model itself to their needs in order to be able to represent their algorithms.

From an abstract point of view an algorithm is a sequence of *steps* (operations, instructions, statements) describing manipulation of *data* (variables) that is finally executed by a *machine* (or human). Consequently, these three aspects (data, operators and execution) represent the core components that have to be represented by the model and are considered in the following sections.

3.1. Data Model

In classical programming languages variables are used to represent data values 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 the statements and instructions manipulating the data. This concept is also taken up in the data model. A variable object is a simple key-value-pair containing a name (represented as a string) and a value (an arbitrary object). The data type of a variable's value doesn't have to be fixed explicitly but is given by the type of the value itself.

In a typical heuristic optimization algorithm a lot of different data values and consequently also variables are used. Hence, in addition to data values and variables special objects called scopes are needed for variable management to keep a clear structure. Each scope can hold an arbitrary number of variables. To access a variable in a scope the variable name is used as an identifier, so each variable has to have a unique name in each scope it is contained.

In the domain of heuristic optimization hierarchical structures are very common. For example, in terms of evolutionary computation, an environment contains several populations, each population contains individuals (solutions) and these solutions may consist of different solution parts. Furthermore, hierarchical structures are not only very suitable in the area of heuristic optimization but in general are used to assemble complex data structures by combining simple ones. As a consequence it is quite reasonable to combine scopes in a hierarchical way to represent such layers of abstraction. Each scope may contain any number of sub-scopes leading to an n-ary tree structure. For example one scope representing a set of solutions (population) may contain several other (sub-)scopes representing the solutions themselves.

When retrieving a variable from a scope this hierarchical structure of scopes is also taken into account. If a variable (identified by its name) is not found in a scope, the variable lookup mechanism continues searching for the variable in 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). Consequently, each variable in a scope is also "visible" in all sub-scopes of that scope. However, if another variable with the same name is added in one of the sub-scopes, it hides the original one (due to the lookup procedure). Note that this behavior is very similar to scopes in classical programming languages. That is also the reason why the name "scope" was chosen.

Based on this abstract representation of data, the next section describes operators which are applied on scopes to manipulate data. Therefore, operators represent the fundamental building blocks of algorithms. Due to the hierarchical nature of scopes operators may be applied on different abstraction levels leading to several essential benefits concerning parallelization discussed later on.

3.2. Operator Model

Regarding to the definition of an algorithm, the next topic to be defined are steps. Each algorithm is a sequence of clearly defined, unambiguous and executable instructions. These atomic building blocks of algorithms are called operators and are of course also considered as objects in a generic algorithm model. In analogy to classical programming languages these operators can be seen as statements that represent instructions or procedure calls.

In general, operators fulfill two major tasks: On the one hand an operator can access and manipulate a scope's variables or sub-scopes and on the other hand an operator may define the further execution flow (i.e. which operators are executed next). To support genericity of operators and to enable reuse, operators have to be decoupled from concrete variables. For that reason a mechanism is used that is similar to procedure calls.

As an example consider a simple increment operator that increases the value of an integer variable by one. Inside the operator it is defined that the operator is expecting a variable of a specific type (in our case an integer) and how this variable is going to be used. When implementing an operator, formal names are used to identify variables but these formal names do not correspond to any real variable name. The concrete variable remains unknown until the operator is applied on a scope. By this means an increment operator can be used to increment any arbitrary integer variable. When adding an operator to an algorithm the user has to define a mapping between the formal variable names used inside the operator and the real variable names that should be used when the operator is finally executed. When a variable is accessed by the operator the variable's formal name is automatically translated into the actual name, which is then used to retrieve the variable from the scope. As a consequence meta-information has to be provided by an operator to declare on which variables the operator is going to work on. This information is represented by objects called variable infos that can be added to each operator. Additionally, the user can access these variable infos to set the actual variable names. In analogy to classical procedure calls variable infos can therefore be interpreted as parameter lists of operators.

In order to build complex algorithms, operators are combined to a sequence of operations. Each operator contains arbitrary many references to other operators (sub-operators) representing the static structure of an algorithm. When an operator is executed it can decide which operators have to be executed next. In that way designated control operators can be built that do not manipulate data but dynamically define the execution flow. For example, a sequence of operators can be specified using an operator that just returns all its sub-operators as the next operators to be executed. Another example would be a branch operator that is choosing one of its sub-operators as the next operator depending on the value of some variable contained in the scope the operator is applied on (cf. an if- or switch-statement in classical programming languages). In contrast to scopes, operators do not form a hierarchical structure (although contained operators are called sub-operators) but are combined in a graph. In other words an operator that has already been used in some upper level can be added as a sub-operator again leading to cycles in operator references. In combination with sequences and branches this concept can be easily used to build loops or any other form of control structures known from classical programming languages. For example, a do-while-loop can be realized as a sequence operator containing a branch operator as its last sub-operator. This branch operator can contain a reference back to the sequence operator as its sub-operator defining the branch executed if the condition holds.

As a result, it is possible to represent concepts known from classical (procedural) programming languages in the operator model (sequences, branches, loops). It is therefore capable of representing arbitrary algorithms and of course especially heuristic optimization algorithms.

3.3. Execution Model

The last aspect to be considered is execution of algorithms. Represented as operator graphs algorithms are executed step by step by virtual machines called engines. In each iteration an engine performs an operation which is applying an operator on a scope. Therefore, an operation represents a tuple of an operator and the scope the operator should be applied on. At the beginning of each algorithm execution an engine is initialized with a single operation containing the initial (root) operator of the algorithm and an empty scope (i.e. the global scope).

As the program flow is dynamically defined by operators themselves, each operator may return one or more operations after its execution that have to be executed next. As a consequence engines have to keep track of all operations waiting for execution. These pending operations are kept in a stack. In each iteration an engines pops the next operation from the top of its stack, executes the operator on the scope and pushes the returned successor operations in reverse order back on the stack again (reversing the order is necessary to maintain the execution sequence as a stack is a last-in-first-out queue). By this means engines perform a depth-first expansion of operators. A pseudo-code representation of the main loop of engines is shown below:

```
clear global scope
clear operations stack
push initial operation

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

As a summary of the generic algorithm model consisting of the three parts described in the previous sections (data model, operator model, execution model), figure 1 gives the main identified components and shows the corresponding interactions.

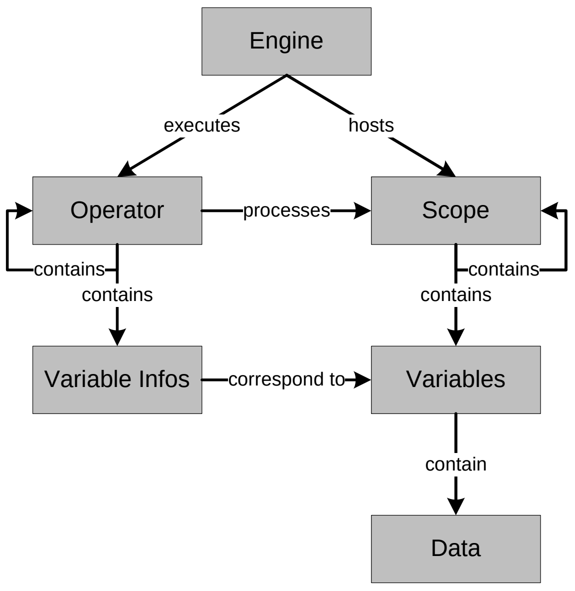

Figure 1: Generic algorithm model

4. PARALLELISM

In many real world applications of heuristic optimization performance is of crucial importance. Therefore, concepts of parallel and distributed computing have to be used frequently to utilize multiple cores or even computers (clusters) and to distribute the work load. In the parallel heuristic optimization community several models of parallelization have been developed reflecting different strategies. In general these models can be categorized in two main approaches:

On the one hand quality calculation can be considered for parallelization. For many optimization problems the computational effort required for calculating the quality of a single solution is much higher than the effort needed by solution manipulation operations. Consider for example heuristic optimization in the area of production planning or logistics. In that case evaluating a solution is done by building a schedule of all jobs or vehicles available whereas manipulation of solutions is usually reduced to twisting of permutations (this depends on the solution encoding but variations of permutation-based encoding are frequently used for combinatorial optimization problems and have been successful in many applications). As another example heuristic optimization of data representation or simulation models can be mentioned as in these applications solution evaluation 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) executing the evaluation of solutions in parallel is a helpful approach (usually called *global parallelization*) (Alba 2005). However, the heuristic algorithm performing the optimization by creating new and hopefully better solutions remains a sequential one.

On the other hand parallelization can also be considered for heuristic optimization algorithms directly (Alba 2005). By splitting solution candidates into distinct sets, these sets can be optimized independently from each other and therefore in parallel. For example parallel multi-start heuristics are simple representatives of that concept. In that case multiple optimization runs are executed with different initial solutions to achieve a broader coverage of the search space. No information is exchanged between solution sets until the end of the optimization. In more complex approaches exchange of information from time to time is additionally used to keep the search process alive and to support diversification of the search (coarse- or fine-grained parallel genetic algorithms, e.g.). In general, population-based heuristic optimization algorithms are very well suited for this kind of parallelization as multiple populations can be used as distinct sets and no additional splitting of solutions is necessary.

By separating the definition of parallelism in algorithms from the concrete way how a parallel algorithm is executed, users of heuristic optimization software systems can focus on algorithm development without having to rack their brains on how parallelization is actually done. If the basic algorithm model already supports parallelism, all different kinds of parallel algorithms can be modeled enabling also the implementation of these different parallelization strategies used in heuristic optimization discussed above.

The generic algorithm model discussed in this paper follows a strict separation of data, operations and algorithm execution. As a consequence introducing parallelism can be done quite easily by grouping operations into sets that are allowed to be executed in parallel. As an operator may return several operations to be executed next, it can mark this group of successor operations as a parallel group. This signals the engine that some operations are independent from each other and the engine is now free to decide which kind of parallel processing should be used for their execution. How parallelization is actually done depends on the engine only. For example, one engine can be developed that doesn't care about parallelism at all and executes an algorithm still in a sequential way (which is especially helpful for testing algorithms before they are really executed in parallel). Another engine might use multiple threads to execute operations of a parallel group (exploiting multi-core CPUs) or an even more sophisticated engine might distribute parallel operations to several nodes in a network following either a client-server-based or a peer-to-peer based approach (utilizing cluster or grid systems). Also meta-engines are possible that use other engines for execution which enables hybrid parallelization on different levels (for example distributing operations to different cluster nodes on a higher level and using shared-memory parallelization on each node on a lower level). As a consequence the parallelization concept used for executing parallel algorithms can simply be specified by the user by choosing an appropriate engine the algorithm is executed on. The algorithm itself doesn't have to be modified at all.

Based on this parallelization concept the generic algorithm model allows development of special control operators for parallel algorithms. For example, parallel execution of operators can be realized by an operator very similar to the sequence operator already discussed in the previous section. The only difference is that in the parallel case the operator has to mark its successor operations containing all its sub-operators and the actual scope as a parallel group. Furthermore, the hierarchical structure of scopes enables data partitioning in a very intuitive way. As an example consider a sub-scopes processor which returns a parallel group of operations, containing an operation for each of its sub-operators being executed on one of the sub-scopes of the current scope. By this means parallelization can be applied on any level of scopes leading to global, fine- or coarse-grained parallel heuristic algorithms.

5. LAYERS OF USER INTERACTION

As the generic algorithm model described in the previous sections is not dedicated to heuristic optimization but can represent arbitrary algorithms, it offers a very high degree of flexibility. Operators representing a broad spectrum of actions ranging from trivial increments or variable assignments to complex selection or manipulation techniques can be used as building blocks for algorithm development leading to a high degree of code reuse. Furthermore, also custom operators can be integrated easily, if the set of predefined operators provided by a framework is not sufficient.

However, such a low level of abstraction is not reasonable for many users as even the representation of simple algorithms results in quite large and complex operator graphs. Therefore, several layers of user interaction are required that represent different degrees of abstraction.

Such layers can be realized on top of the generic algorithm model by using combined operators that encapsulate operator graphs (i.e. algorithms) and represent more complex operations. In that case the generic algorithm model serves as a meta-model for heuristic optimization algorithms. An important aspect is that combined operators are not hard-coded in a framework but can be developed and shared on the user instead of the development level.

For example, users can provide combined operators representing ready-to-use heuristic optimization algorithms (like a canonical genetic algorithm, simulated annealing, hill climbing, tabu search, or particle swarm optimization) that can be used as black box solvers. By this means other users can start working with specific algorithms right away without having to worry about how an algorithm is structured in detail.

In between predefined solvers and the generic algorithm model, arbitrary other layers can be realized representing various (user-specific) heuristic optimization models. For example, generic models of specific heuristic optimization algorithm flavors (evolutionary algorithms, local search algorithms, etc.) can be represented by a set of operators useful to enable experimenting with these paradigms without putting the burden of the whole complexity and genericity of the basic algorithm model on users.

In figure 2 this layered structure of user interaction is shown schematically. As all these layers use the same algorithm model as their basis, users are free to decide which level of abstraction is adequate for their needs.

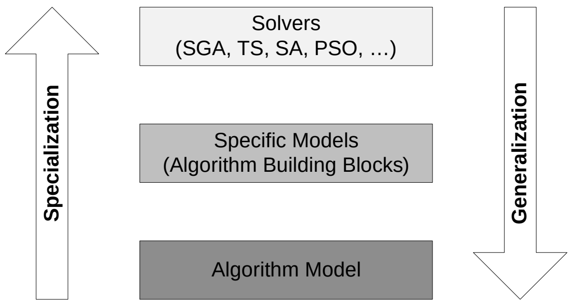

Figure 2: Layers of iser interaction

6. CONCLUSION

In this paper the authors presented a generic model for heuristic optimization algorithms. Compared to other models known from existing heuristic optimization software frameworks, the main advantage of the proposed solution is a higher level of abstraction. By considering the three main aspects of algorithms in general (data, operators and execution), a generic model was developed that is not only capable of representing heuristic optimization techniques but can be used for modeling arbitrary algorithms.

By this means the model can act as a meta-model which enables users to incorporate custom heuristic optimization paradigms and algorithms in a flexible way. As the burden of a single and fixed representation trying to cover all different kinds of heuristic optimization concepts is removed, users are free to realize custom models built on top of the generic algorithm model that exactly fit their needs.

Furthermore, aspects like parallelism or user interaction on different layers of abstraction have been considered, showing that the described model is suitable for developing a new generation of heuristic optimization software systems.

REFERENCES

Alba, E., 2005. *Parallel Metaheuristics: A New Class of Algorithms*. Wiley.

Doerner, K. F., Gendreau, M., Greistorfer, P., Gutjahr, W., Hartl, R. F., Reimann, M., 2007. *Metaheuristics: Progress in Complex Systems Optimization*. Springer.

Fink, A., Voß, S., 2002. HotFrame: A Heuristic Optimization Framework. In: *Optimization Software Class Libraries*. Kluwer.

Gagne, C., Parizeau, M., 2006. Genericity in Evolutionary Computation Software Tools: Principles and Case-Study. *International Journal on Artificial Intelligence Tools*, 15, 173-194.

Johnson, R., Foote, B., 1988. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1 (2), 22-35.

Jones, M. S., McKeown, G. P., Rayward-Smith, V. J., 2002. Distribution, Cooperation, and Hybridization for Combinatorial Optimization. In: *Optimization Software Class Libraries*. Kluwer.

Voß, S., Woodruff, D. L., 2002. Optimization Software Class Libraries. In: *Optimization Software Class Libraries*. Kluwer.