ON THE BENEFITS OF A DOMAIN-SPECIFIC LANGUAGE FOR MODELING METAHEURISTIC OPTIMIZATION ALGORITHMS

Stefan Vonolfen[(a)], Stefan Wagner[(b)], Andreas Beham[(c)] , Michael Affenzeller[(d)]

[(a)(b)(c)(d)] Upper Austria University of Applied Sciences, Campus Hagenberg
School of Informatics, Communication and Media
Heuristic and Evolutionary Algorithms Laboratory
Softwarepark 11, A-4232 Hagenberg, Austria

[(a)]stefan.vonolfen@heuristiclab.com, [(b)]stefan.wagner@heuristiclab.com, [(c)]andreas.beham@heuristiclab.com,
[(d)]michael.affenzeller@heuristiclab.com

ABSTRACT

This work provides a case-study of how metaheuristic optimization algorithms can be developed using a domain-specific language as a separate modeling layer. A separation of the modeling process from the implementation of the algorithmic concepts improves the communication and collaboration of practitioners, optimization experts and programmers. This is achieved by providing a higher level of abstraction compared to a general-purpose programming language. A generic and extensible modeling concept is presented and several example algorithm models are illustrated.

Keywords: metaheuristics, modeling, domain specific language

1. INTRODUCTION

Metaheuristics are general search strategies that can be used to calculate approximate solutions for many different kinds of problems in diverse application areas. They guide the search process and can be seen as an algorithmic framework with the ability to be tailored to different problem environments (cf. Blum and Roli 2003). Many metaheuristic search strategies are inspired by nature. For an overview of metaheuristic techniques see for example Talbi (2009).

However according to Wolpert and Macready (1995) there is no general search strategy that performs well for all kinds of problems. This implies that even though metaheuristics are general search strategies, tailoring has to be done in order to generate good and feasible solutions for a given problem.

According to Talbi (2009) there are three major approaches for the development of metaheuristics: From scratch, code reuse and both design and code reuse. A software framework provides reusable code and also a reusable design (cf. Johnson and Foote 1988). The goal of using a metaheuristic framework is to be able to build on as much existing code and design elements as possible when developing a new optimization solution. There are many different frameworks available in that area - examples are Templar (Jones 2000), HotFrame (Voss and Woodruff 2002), ParadisEO (Talibi 2009) or HeuristicLab (Wagner, Winkler, Braune, Kronberger, Beham, Affenzeller 2007; Wagner 2009).

When using such frameworks - however - the user generally needs an in depth-knowledge of the framework since according to (Talibi 2009) currently a white-box approach is more suited to metaheuristics than a black-box approach. This means that the user often needs to know implementation details of the framework and needs to have programming skills.

White-box reuse means that a very low level of abstraction (i.e. the source code level) has to be provided to the user in order to tailor an algorithm to a certain problem. There are some approaches to build an additional level of abstraction on top of the frameworks including EASEA (Collete, Luttion, Schoenauer and Louchet 2000) and GUIDE (Da Costa and Schoenauer 2009). However these approaches are either limited to a particular optimization paradigm or very close to an actual programming language.

The scope of this work is to raise this level of abstraction by developing a modeling layer for metaheuristic optimization techniques which is independent of the underlying implementation. This is accomplished by providing black-box reuse without losing the flexibility to tailor the algorithm. Black-box reuse means that components (building-blocks) are provided that have a well defined interface and can be combined in a well-defined way.

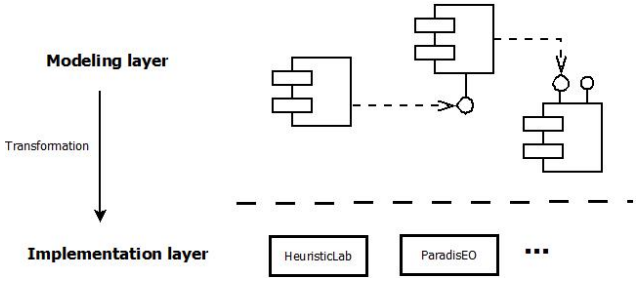

Figure 1: Separation of the modeling layer from the implementation layer

The modeling layer is separated from the implementation layer as illustrated in Figure 1. In the modeling layer algorithm models can be built by combining algorithm building blocks. These building blocks are then executed in the implementation layer, for example by transforming them to code using an underlying framework.

The created models are written according to a specification - an extensible domain specific language (DSL) for metaheuristic optimization. This language consists of all building blocks that are available to the modeler and can be extended by adding new components. In that way for example new algorithmic concepts or problem-specific parts can be incorporated. The models can be validated according to the specification.

There are various benefits a separate modeling layer yields which are examined in this work. First of all additional layers of abstraction can be built where the implementation details can be abstracted and the requirement of programming skills can be removed. Second of all different frameworks or framework versions can be used to transform the models into an executable representation. Apart from that various artifacts can be generated from the models including for example textual descriptions.

When building this modeling layer both, top-down and bottom-up, approaches are possible. Top-down means that the modeler starts by defining abstract components that the algorithm is built of. Then by using stepwise refinement these components are mapped to the according implementation concepts. Bottom-up means that more abstract components are defined starting from an existing implementation layer.

2. BENEFITS

The level of abstraction that is provided by current frameworks is raised by providing a separate modeling layer above existing frameworks and consequently the implementation layer is separated from the modeling layer. This yields several benefits:

- **Communication:** In heuristic optimization projects there are usually different types of stakeholders. When implementing a solution, an expert in the field of heuristic optimization algorithms usually needs to have programming skills since most of the available frameworks are whitebox oriented. A separate modeling layer could improve the communication between experts in the field of optimization and programmers.

- **Collaboration:** By proving a common model, algorithm designers and programmers can work together more efficiently because both, top-down and bottom-up, modeling approaches are possible. On the one hand, in a top-down approach an algorithm designer would first specify the abstract algorithm building blocks and model the algorithm, then the programmer maps these building blocks to an existing framework or implements them from scratch. On the other hand, in a bottom-up approach the programmer first maps existing implementations to modeling building blocks that the algorithm designer can then use to design new algorithmic variants. Of course, also mixed approaches are possible where certain building blocks are provided and new ones are added later on by the programmer.

- **Abstraction:** Algorithms can be modeled independently and the implementation details are abstracted. As a result, no internal knowledge of the underlying framework is necessary and no programming skills are required. During the modeling process the technical details are hidden. Furthermore higher layers of abstraction can be created, than it is possible to realize within a programming language. This can be achieved for example by using coarse grained components.

- **Generation:** Various artifacts can be generated from the models. For example different frameworks or framework versions can be supported for the actual implementation and execution of the algorithmic model, since it is independent of the implementation. Apart from that for example textual descriptions of the algorithm or graphical representations can be generated. In terms of communication and collaboration it is also important that the generated executable representations can be easily executed and algorithm runs can be easily analyzed by the domain expert. The generated executable representations of the algorithms are all unified, which means that for example coding standards are followed and the design is standardized. This is often very difficult to achieve when writing hand-crafted code, especially when parts are written by different developers.

- **Model validation:** Since the model is defined according to a meta-model, various validations can be included. This validation is often only performed on the syntactical level in current frameworks. An example would be that the operators actually fit together and work on the same type of problem representation.

- **Flexibility:** Apart from providing the before mentioned benefits, the modeling layer should provide full flexibility. This means that it should not be restricted to a certain paradigm and the algorithmic structure should be fully modifiable.

3. EXISTING MODELING LAYERS

There are already some approaches of providing a modeling layer for heuristic optimization. For example some domain specific languages for local search strategies have been developed which are mostly based on constraint programming according to Fink and Voss (2001). Other examples of modeling layers are EASEA (Collete, Luttion, Schoenauer and Louchet 2000), GUIDE (Collete, Luttion, Schoenauer and Louchet 2000) and the HotFrame configurator (Fink and Voss 2001) which are examined in the following. Also the HeuristicLab environment (Wagner, Winkler, Braune, Kronberger, Beham, Affenzeller 2007; Wagner 2009) can be regarded as a modeling tool.

In this section the existing modeling layers are evaluated according to the benefits that are discussed in section 2. Table 1 gives an overview of the evaluation criteria. A √ symbol means that the criterion is fulfilled; a o symbol means that a criterion is partially met and a × symbol means that the criterion is not met. The − symbol means that a certain criterion is not applicable to a modeling layer.

Table 1: Evaluation

Criterion	EASEA	GUIDE	HotFrame Configurator	HeuristicLab
Communication	×	o	o	√
Collaboration	×	o	o	√
Abstraction	o	√	o	o
Generation	√	√	√	−
Model validation	×	−	−	−/×
Flexibility	o	×	×	√

EASEA is a scripting language for evolutionary computation and is close to a programming language. This means that even though it abstracts the actual programming language and framework, the communication and collaboration between implementers and domain experts is not supported. This is because programming skills are required to use EASEA efficiently. However, one benefit of using EASEA is certainly that code generation for different frameworks is possible. Model validation is not supported. In terms of flexibility, EASEA is designed for evolutionary computation. However, since EASEA is very close to an actual programming language, one could argue that additional paradigms can be incorporated.

GUIDE provides a graphical user interface for building evolutionary algorithms based on an algorithm template. Its graphical user interface is intuitive and reflects the terms used in the evolutionary computation community. This enables communication between programmers and domain experts. In terms of collaboration building blocks based on a certain framework can be defined and offered to the modeler. However, in terms of communication and collaboration the analysis of algorithm runs is not supported directly. All framework and implementation details are abstracted and code can be generated for various frameworks. Model validation is not required, since the tool does not allow the construction of invalid models. However in terms of flexibility GUIDE is limited to evolutionary algorithms and the underlying algorithm structure cannot be modified.

Similar to GUIDE, the HotFrame configurator is designed as a configuration tool. It hides all implementation details by providing a configuration language to the user which improves the communication. In terms of collaboration, programmers can provide configurable building blocks to the domain expert. From the configuration an executable representation is generated. However, the analysis of algorithm runs is not supported. In terms of abstraction, the HotFrame configurator is limited to the HotFrame framework. There is no need for model validation since it only allows the user to create valid models. When it comes to flexibility, the HotFrame configurator is limited to the configuration of the algorithms; the algorithm structure cannot be modified.

Regarding HeuristicLab, one great benefit is certainly the communication and collaboration between domain experts and programmers. In the current release (version 3.3) communication and collaboration between programmers and domain experts is supported using a model-driven approach which is described by Wagner (2009). Apart from building the algorithms using pre-defined building blocks, algorithm runs can be executed and analyzed within the HeuristicLab environment. The details of the underlying programming language are abstracted completely by providing a generic modeling concept. However, this layer of abstraction is tied to the HeuristicLab framework and right now it cannot be used to generate code for other frameworks. This means that the framework details are not abstracted. As a result the modeling and execution layer are not separated and no generation is needed to execute the models. In terms of flexibility, there exist two different types of algorithms in the HeuristicLab framework: pre-defined and user-defined algorithms. Pre-defined algorithms can be parameterized (similar to the HotFrame configurator) and thus there is no need for model validation, since the user cannot construct invalid models. User-defined algorithms provide a full degree of flexibility. However, for these algorithms no fully-fledged meta-layer exists right now. This means that the control and data flow cannot be validated when constructing custom algorithms.

4. MODELING CONCEPT

As stated in section 1, the developed modeling concept is a black-box oriented approach. This allows the definition and combination of well defined building blocks. These building blocks are components that represent a specific encapsulated functionality which is part of a metaheuristic optimization algorithm. This functionality can be realized on different levels of abstraction.

The elementary components that the algorithm is built of are operators, similar to the modeling concept proposed by Wagner (2009). However the level of abstraction is higher and less focused on

implementation and execution than the algorithm models of the examined frameworks. The modeling layer is separated from the implementation/execution layer.

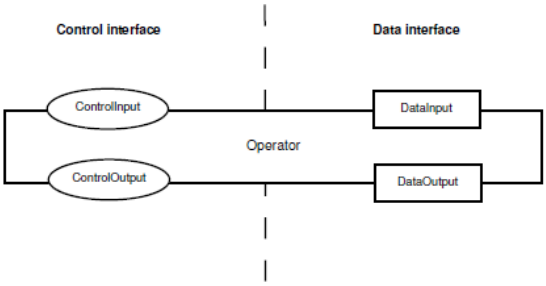

Figure 2: Interface of an operator

Following a component-based approach, these building blocks have a well defined interface. This interface determines how building blocks can be combined. As illustrated in Figure 2 an operator has a control interface and a data interface. The control interface can be used to model the control flow, the data interface to model the data flow of the algorithm. This separation allows a high level of flexibility. The control and data interfaces consist of input and output ports that can be connected to each other.

4.1. Control Flow

By connecting the control ports of the operators, the control flow of the algorithm can be modeled. Each operator has exactly one control input and zero or more control outputs. A connection between the control output of an operator A and a control input of an operator B means that operator B is called by operator A. Each algorithm has a start operator that has no control input and one control output that triggers the execution of the algorithm.

Whenever multiple control output ports are connected to one control input port, the control flows are joined at that point. The semantic of a join is that the execution of all previous operators has to be finished before the called operator is executed. Finished means that all previous operators either have finished execution or are not executed.

4.2. Data Flow

Whenever an operator is executed, the parameters from the input data ports are processed and the results are written to the output data ports. An operator can have zero or more data input and zero or more data output ports. The data input ports store the current value until a new value is set. The value of each data input port is set to empty at the beginning of the execution of an algorithm. A data output port can be connected to an arbitrary number of data input ports. The value of the data output port is written to all connected data input ports.

5. IMPLEMENTATION

To provide means of formulating the models, a domain specific language (DSL) for metaheuristic optimization algorithms is developed. The design and implementation of the modeling layer is based on the general modeling concept that is outlined in the previous section. Several design choices were made to create a modeling system that meets the benefits described in section 2.

First of all it was decided that an external graphical DSL should be developed. The main reason for that is raising the level of abstraction and hiding the implementation details. Furthermore, to enhance the communication between domain experts and programmers, a graphical concrete syntax was chosen.

To achieve a high level of flexibility the vocabulary of the DSL should not be fixed but definable by the user. The user should be able to define new building blocks and that way add a new vocabulary to the modeling language. The developed DSL should be highly extensible.

As a result, actually two graphical editors are developed: One editor to define the algorithmic building blocks and one editor to model the algorithms. That way, new building blocks for the modeling of algorithms can be created and the vocabulary can be easily extended.

The DSL to define algorithmic building blocks provides a specification of the algorithmic components. It is used to create toolbox models which provide a set of components that can be used in an algorithm. In principal these two DSLs can be used by the same or by different users. For example one user could create certain building blocks and then provide them to another user. The first user that defines the components could be a programmer and the second user that builds the algorithm models using the second editor could be a domain expert in heuristic optimization. That way the communication between these two user groups could be enhanced.

Both languages were implemented as graphical editors using the Microsoft Visual Studio Visualization and Modeling SDK[1]. Both designers are integrated into the Visual Studio environment.

6. RESULTS

Using the developed modeling tools, several example algorithm models were created. Figure 3 shows a genetic algorithm model solving the traveling salesman problem using a permutation encoding. In that case, high-level building blocks which are defined in the corresponding toolboxes are combined with each other.

[1] http://code.msdn.microsoft.com/vsvmsdk

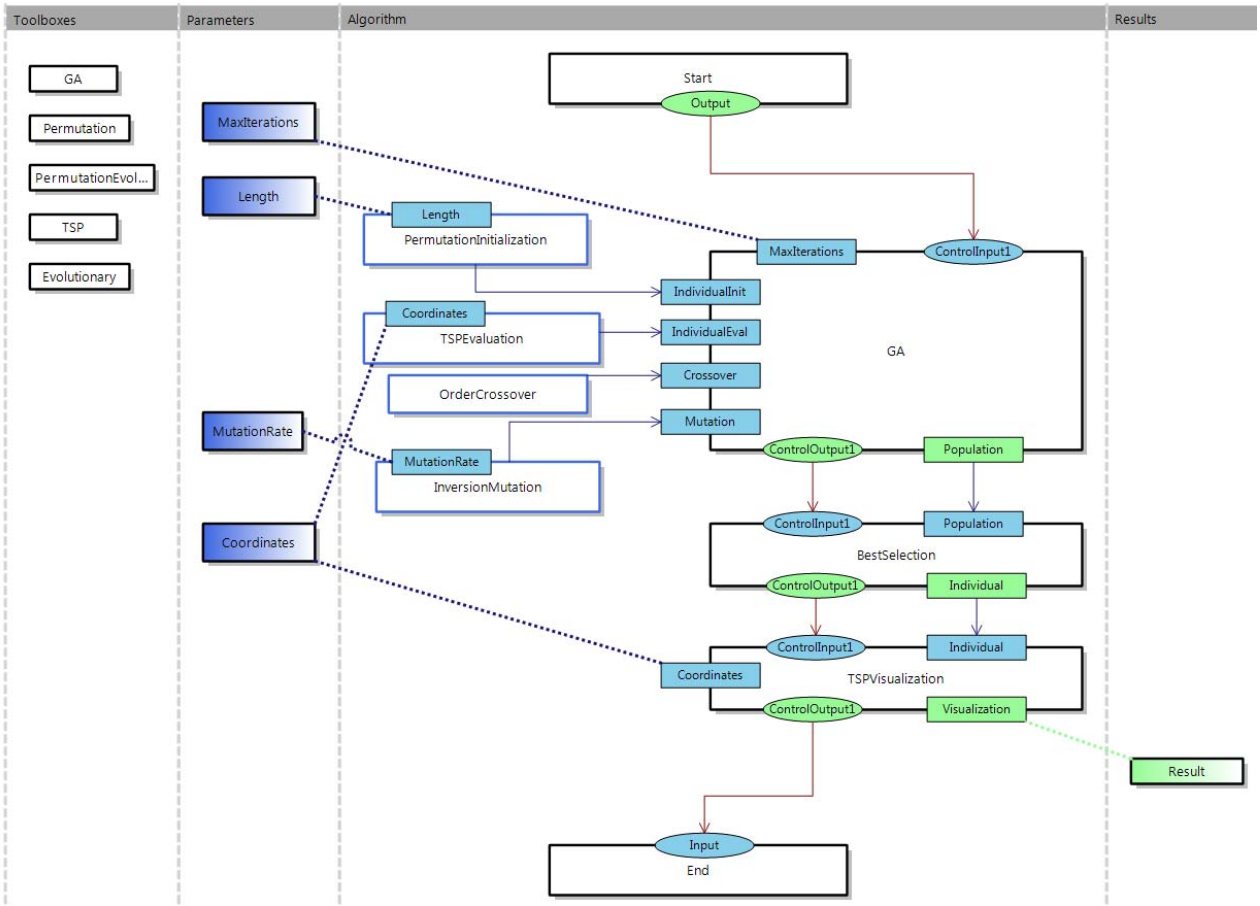

Figure 3: Genetic algorithm model solving the TSP

These high-level building blocks are defined in toolboxes. Some can be directly mapped to a framework, others are defined as finer-grained algorithm models. For example the *GA* building block is defined by a finer-grained GA building block that includes operations such as evaluation, selection and reproduction following the abstract GA specification.

In the example model, the problem-specific *TSP* toolbox is combined with the encoding-specific *Permutation* toolboxes and the general *Evolutionary* and *GA* toolbox.

Due to data flow validation, only building blocks that fit together can be combined; for example no real-vector operations can be used in conjunction with permutation operations.

These algorithm models can be converted to executable HeuristicLab algorithms using the HeuristicLab code generator. The targeted HeuristicLab version is 3.3.0 (http://dev.heuristiclab.com).

The runtime and quality of generated solutions for different problem instances retrieved from the TSPLIB [Rei91] has been measured and analyzed. These results are compared to the standard GA implementation included in the HeuristicLab 3.3.0 release which is hand-crafted.

All test runs where done using a population size of 100, a mutation rate of 5% and one-elitism. For all instances 10 independent test runs were done and the mean value and standard deviation was calculated.

The quality of the best solution after 100 iterations is examined to make sure that the generated code produces correct results. The results are listed in table 3. All deltas between the results of the hand-crafted GA and the generated GA are in the range of the standard deviation which shows the correctness of the approach.

Table 3: Best solution quality analysis

Instance	Handcrafted μ	Handcrafted σ	Generated μ	Generated σ
eil51	936.8	46.2	941.0	54.3
ch130	30438.1	670.4	30952.7	1082.3
rl5934	39008794.3	106313.8	39122368.0	246800.6
rl11849	82537969.2	235147.2	82545883.2	299712.9
d18512	56920526.6	151996.7	56985640.1	80526.1

7. CONCLUSION

Concluding, a prototype of the modeling environment has been developed and several benefits have been met and also some drawbacks were identified:

- **Communication**: It has been shown that these algorithm models are close to the abstract algorithmic specification found in the literature. This illustrates that a domain-specific language can enhance the communication between domain experts and

programmers in the field of heuristic optimization. However, graphical models can get quite complex and difficult to change for complex algorithmic concepts and it is crucial to choose a sufficient level of abstraction.

- **Collaboration:** Using the prototype, top-down and bottom-up modeling approaches are possible which advances the collaboration between the different stakeholders.

- **Abstraction:** The algorithm models are designed independently of the underlying framework and programming language. However, it has yet to be shown that these models can be mapped to frameworks other than HeuristicLab.

- **Generation:** The generated code turned out to be less efficient than the handcrafted code. Furthermore, the generated code is very different from the way a domain expert would model the algorithms in the HeuristicLab environment. This has a negative impact on the maintainability.

- **Model validation:** Data flow validation has been implemented and prevents the user from creating invalid models.

- **Flexibility:** The algorithm model provides full flexibility. This is shown by the fact that several algorithmic concepts have been used in the example models.

In the future, additional code generators could be developed that produce more optimal code or produce code that is closer to the algorithm model of HeuristicLab. Also additional code generators could be developed that produce code for other frameworks such as ParadisEO or HotFrame. This would be especially beneficial for the comparison of different frameworks.

In terms of validation, control flow validation could be implemented in addition to the existing data flow validation.

Concluding, apart from being beneficial as a case-study, the work described in this paper could also have impact on the further development of HeuristicLab. Especially aspects concerning abstraction, generation and model validation could be transfered to the HeuristicLab algorithm model.

ACKNOWLEDGMENTS

The work described in this paper was done within the Josef Ressel Centre for Heuristic Optimization *Heureka!* (http://heureka.heuristiclab.com) sponsored by the Austrian Research Promotion Agency (FFG).

REFERENCES

Blum, C.; Roli, A., 2003. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. In: *ACM Computing Surveys 35(3)*, 268-308.

Collete, P., Luttion, E., Schoenauer, M. And Louchet, J., 2000. Take It EASEA. *PPSN VI: Proceedings of the 6th International Conference on Parallel Problem Solving from Nature*, pp. 891-901. London (UK).

Da Costa, L.; Schoenauer, M., 2009. Bringing Evolutionary Computation to Industrial Applications with GUIDE. *GECCO 2009*. Montreal, Quebec (Canada).

Fink, A; Voss, S.; 2001. *Reusable metaheuristic software componentsand their application via software generators.* In: Proceedings of the 4th Metaheuristics International Conference, Porto,pages 637–642.

Johnson, R.E.; Foote, B., 1988. Designing reusable classes. In: *Journal of object-oriented programming 1(2)*, 22-35.

Jones, M.S., 2000. *An Object-Oriented Framework for the Implementation of Search Techniques*. Thesis (PhD). University of East Anglia.

Talbi, E.-G.., 2009. *Metaheuristics: from design to implementation*. John Wiley & Sons.

Voss, S. And Woodruff, D., 2002. *Optimization Software Class Libraries*. Kluwer Academic Publishers.

Wagner, S.; Winkler, S.; Braune, R.; Kronberger, G.; Beham, A. 2007. Benefits of plugin-bases heuristic optimization software systems. *Computer Aided Systems Theory - EUROCAST Conference*, pp. 747-754.

Wagner, S. 2009. *Heuristic optimization software systems – Modeling of heuristic optimization algorithms in the HeuristicLab software environment*. Thesis (PhD). Johannes Kepler Univiersity, Linz, Austria.

Wolpert, D.H.; Macready, W.G., 1995. *No Free Lunch Theorems for Search*. Santa Fe Institute.

AUTHORS BIOGRAPHY

The web-pages of the authors as well as further information about HeuristicLab and related scientific work can be found at http://heal.heuristiclab.com.