EFFECTS OF MUTATION BEFORE AND AFTER OFFSPRING SELECTION IN GENETIC PROGRAMMING FOR SYMBOLIC REGRESSION

Gabriel K. Kronberger[(a)], Stephan M. Winkler[(b)], Michael Affenzeller[(c)], Michael Kommenda[(d)], Stefan Wagner[(e)]

[(a – e)] Upper Austria University of Applied Sciences
School for Informatics, Communications, and Media
Heuristic and Evolutionary Algorithms Laboratory
Josef Ressel Centre for Heuristic Optimization - Heureka!
Softwarepark 11, 4232 Hagenberg, Austria

[(a)] gabriel.kronberger@fh-hagenberg.at, [(b)] stephan.winkler@fh-hagenberg.at ,
[(c)] michael.affenzeller@fh-hagenberg.at , [(d)]michael.kommenda@fh-hagenberg.at ,
[(e)] stefan.wagner@fh-hagenberg.at

ABSTRACT
In evolutionary algorithms mutation operators increase the genetic diversity in the population. Mutations are undirected and have only a low probability to improve the quality of the manipulated solution. Offspring selection determines if a newly created solution is added to the next generation of the population. By definition, offspring selection is applied after mutation and the effects of mutation are directed and quality-driven.

In this paper we propose an alternative variant of genetic programming with offspring selection where mutation is applied to increase genetic diversity after offspring selection. We compare the solution quality achieved by the original algorithm and the new algorithm when applied to a symbolic regression problem. We observe that solutions produced by the new variant have a smaller generalization error and conclude that the proposed variant is better for symbolic regression with linear scaling.

Keywords: Genetic Programming, Symbolic Regression, Mutation Operators

1. INTRODUCTION

In evolutionary algorithms mutation operators are used to manipulate existing solutions. Usually mutations are non-deterministic, local and undirected and have only a minor effect on the quality of the solution candidate (e.g., the single point mutation operator for genetic algorithms with binary encoding changes a single bit in the binary string representing the solution). The probability that a random mutation improves the quality of a solution is rather small. In genetic algorithms the purpose of mutation operators is to increase the genetic diversity of the population which often has a beneficial effect on the overall dynamics of the algorithm and the final solution quality.

For some problem formulations mutation operators or recombination operators are more natural or more efficient, but most evolutionary algorithms described recently use a combination of both, recombination and mutation, to generate new solution candidates from existing ones.

1.1. Offspring Selection

Offspring selection (Affenzeller and Wagner 2005) is an additional selection step in evolutionary algorithms which is applied after parental selection, recombination and mutation. Offspring selection adds a newly created solution to the next generation only if it fulfills a success criterion. Most often the success criterion is that the newly created child solution must have a better quality than its parents. It has been shown that offspring selection can improve the final solution quality found by genetic algorithms on different benchmark problems (Affenzeller, Winkler, Wagner and Beham 2009).

If offspring selection is applied after mutation, mutations become directed and quality-driven. In this configuration the mutation operator can be compared to a local first-improvement optimization step after recombination that is applied to a small percentage of the generated individuals determined by the mutation rate parameter.

The two algorithm variants studied in this paper can be directly related to the two possible perspectives of mutation in the theory of evolutionary algorithms. In the perspective of population genetics as also pursued by (Beyer and Schwefel 2002) mutation affects the genotype before selection. This is also concordant to the original formulation of offspring selection. In the perspective of genetic algorithms the role of mutation is to increase the genetic diversity of the population through random undirected changes of the genotype that are not driven by solution quality (Holland 1975).

1.2. Genetic Programming

Genetic programming (Koza 1992) is a heuristic problem solving method that uses the principles of evolution to evolve programs that solve the original

problem when executed. The core of genetic programming is an evolutionary algorithm that evolves computer programs in generational steps using selection, recombination and manipulation operations, starting from a randomly initialized population. Solution candidates represent computer programs that are most often encoded as symbolic expression trees where the root node is the program entry point. The set of symbols that can be used in the tree and the interpretation of the symbolic expression tree is problem specific.

One possible problem that can be solved by genetic programming is symbolic regression. In the case of symbolic regression the goal of genetic programming is to find a functional expression to predict the value of a target variable from values of input variables, given a set of observed or measured values for each input variable and the target variable. The functional expression is encoded as a symbolic expression tree that contains basic arithmetic operations (+, -, /, *) in internal tree nodes. In terminal nodes only two types of symbols are allowed: either a terminal node represents an input variable or a constant value.

The task of symbolic regression is a very constrained and simple form of genetic programming that makes this task ideal as a test-bed for more theoretical research. The scope of problems that can be solved be genetic programming is much larger. If it is possible to formulate a language for problem solutions and a fitness function for such solutions can be defined it is possible to use genetic programming to evolve solutions for the problem using the constructs defined in the language. Genetic programming has been used to find novel and human competitive solutions (Koza 2010).

1.3. Mutation in Genetic Programming

The role of mutation in genetic programming is uncertain (Poli, Langdon and McPhee 2008, Langdon and Poli 2002). Koza showed that mutation is not necessary for GP (Koza 1992, 1994) and crossover is sufficient to search the solution space. However, for certain problems it has been shown that GP with mutation performs better than GP with only crossover (Harries and Smith 1997, Luke and Spector 1997). The effect of mutation often depends on the problem and on the GP system (Luke and Spector 1997). This work is the first in which the effects of mutation in combination with offspring selection are studied.

2. EXPERIMENTAL SETUP

The main aim of this paper is to compare two genetic programming variants using offspring selection. The first algorithm applies mutation before offspring selection (BeforeOS), the second algorithm applies mutation after offspring selection (AfterOS).

We ran a number of experiments to answer the following two questions: How many mutated solution candidates are successful and accepted into the next generation if offspring selection is applied after mutation? Is the solution quality better or worse if the algorithm is changed so that mutation is applied after offspring selection?

Many different mutation operators for genetic programming with symbolic expression encoding have been described in the literature. We picked four different mutation operators and applied both algorithm variants each time with a different manipulation operator on a symbolic regression problem. For each configuration 20 independent runs were executed. The results shown in the next section are based on 160 independent genetic programming runs.

2.1. Symbolic Regression Problem

In the experiments we use a dataset from a real world chemical process at Dow Chemical that contains 57 cheap process measurements, such as temperatures, pressures, and flows (inputs) and noisy lab data of a chemical composition which is expensive to measure (http://dces.essex.ac.uk/research/evostar/competitions.html). This data set has been made public by Arthur Kordon, Dow Chemical for the symbolic regression competition which was a side event of the EvoStar 2010 conference. Unfortunately, no details have been published about the chemical plant and the process from which this dataset was created.

The dataset contains a total of 747 measurements of the 58 variables and was split into three partitions: training set (0-400), validation set (400-600) and test set (600-747). The validation set is used by the algorithm for model selection. The best solution determined by the algorithm in each run is evaluated on the test set in order to get an expected value for the unknown generalization error of the model on unseen data.

Each solution candidate considered by the algorithm was linearly scaled before evaluation (Keijzer 2004). Linear scaling transforms the output values generated by the symbolic regression models to the same offset and scale as the original target values; linear scaling has been shown to improve the final solution quality produced by genetic programming (Keijzer 2004).

2.2. Mutation Operators

The following mutation operators are used in the experiments:

ChangeNodeType: This operator selects a single node of the solution that has been selected for manipulation, and replaces the symbol in the selected node with a random symbol from the function library. Symbols are selected with uniform probability. The original symbol is included in the list of available symbols. When a terminal node is manipulated the replacement symbol must be a terminal symbol (either variable or constant).

OnePointShaker: This operator selects a single terminal node of the solution that has been selected for manipulation, and applies a shaking operation to the parameter values of the terminal node. If the selected node is a constant a normally distributed with ($N(0,1)$)

value is added to the current constant value. If the selected node is a variable node a normally distributed (*N(0,1)*) value is added to the weighting factor of the variable and the referenced input variable is selected randomly from the set of all allowed input variables.

FullTreeShaker: This operator applies the shaking operation executed by the *OnePointShaker* on all terminal nodes of the solution that has been selected for manipulation.

SubsituteSubTree: This operator randomly selects a branch in the solution that has been selected for manipulation and replaces the whole branch starting at the selected node with a new random tree of the same size. Random trees are generated with the PTC2 operator (Luke 2000), the same operator that is used for generation of the initial population.

2.3. Parameter Settings

We used the same parameter settings for all experiments, only the manipulation operator was exchanged. These settings are well-proven settings that have been used to generate high quality solutions in a number of real world applications of GP. The upper limit of 500.000 evaluated solutions was used as stopping criterion.

Table 1: GP algorithm parameter values for all experiments.

Parameter	Value
Population size	1000
Max tree size	100
Max tree height	10
Mutation rate	15%
Comparison factor	1
Success ratio	1
Crossover	Sub-tree swapping (Koza 1992)
Initialization	PTC2 (Luke 2000)
Selection	50%: Random 50%: Proportional
Evaluation wrapper	Linear scaling (Keijzer 2004)

3. IMPLEMENTATION

The following tables show the pseudo-code of a genetic programming algorithm with offspring selection. Inside the offspring selection loop solution candidates are evaluated only on the training set. The output of the solution candidate is scaled linearly to match the offset and scale of the original target variables (Keijzer 2004).

Table 2 shows the first algorithm (BeforeOS) where mutation is applied inside the offspring selection loop. Thus only mutations which do not have a negative effect on the solution quality are accepted.

Table 3 shows the proposed algorithm variant (AfterOS). The difference to the first algorithm is that the mutation operator is applied after the next generation has been populated via repeated application of selection and crossover in the offspring selection step. The solution candidates which are manipulated have to be evaluated a second time on the training set.

Table 2: Algorithm I: BeforeOS

```
Initialization:
    i ← 0
    Best-Solution ← Ø
    Pop_i ← Create-Random-Individuals_PTC2 (PopSize)
    Evaluate_training ( Pop_i )
Repeat (Main Loop):
    Pop_i+1 ← Ø
    Repeat (Offspring Selection):
        Parent_male ← Selection_male (Pop_i)
        Parent_female ← Selection_female (Pop_i)
        Child = Crossover (Parent_male Parent_female)
        Conditional on Mutation Rate
            Mutate (Child)
        Quality_child ← Evaluate_training (Child)
        if Quality_child ≤ Min(Quality_male, Quality_female)
            Pop_i+1 ← Pop_i+1 ∪ {Child }
        Else
            Discard Child
    Until |Pop_i+1| = PopSize
    Best-Solution ←
        argmin_solution (Evaluate_validation (solution)),
        where solution ∈ Pop_i+1 ∪ { Best-Solution }
    i ← i + 1
Until Stopping-Criterion = true
Output ( Best-Solution )
```

Table 3: Algorithm II: AfterOS

```
Initialization:
    i ← 0
    Best-Solution ← Ø
    Pop_i ← Create-Random-Individuals_PTC2 (PopSize)
    Evaluate_training ( Pop_i )
Repeat (Main Loop):
    Pop_i+1 ← Ø
    Repeat (Offspring Selection):
        Parent_male ← Selection_male (Pop_i)
        Parent_female ← Selection_female (Pop_i)
        Child = Crossover (Parent_male Parent_female)
        Quality_child ← Evaluate_training (Child)
        if Quality_child ≤ Min(Quality_male, Quality_female)
            Pop_i+1 ← Pop_i+1 ∪ {Child }
        Else
            Discard Child
    Until |Pop_i+1| = PopSize
    For each solution ∈ Pop_i+1
        Conditional on Mutation-Rate
            Mutate (solution)
            Quality_solution ← Evaluate_training (solution)
    Best-Solution ←
        argmin_solution (Evaluate_validation (solution)),
        where solution ∈ Pop_i+1 ∪ { Best-Solution }
    i ← i + 1
Until Stopping-Criterion = true
Output ( Best-Solution )
```

Both algorithms have been implemented using an internal pre-release version of HeuristicLab (http://dev.heuristiclab.com) (Wagner 2009), a generic and paradigm independent framework for heuristic optimization.

4. RESULTS

The results of the experiments show that when offspring selection is applied after mutation the number of mutated individuals that are accepted into the next generation is high in the beginning stages of the algorithm, but drops later. There is no significant difference in the solution quality on the training data, however, the solutions generated by the algorithm with mutation after offspring selection are better on the validation data-set.

4.1. Effective Mutation Rates

Figure 1 shows the effective mutation rate for all mutation operators. In the first generations of the run almost all mutations improve the solution quality; in the later stages the probability sinks, but is still rather high. All mutation operators except for the *FullTreeShaker* have a probability greater than 30% to produce successful solutions even at end of the run.

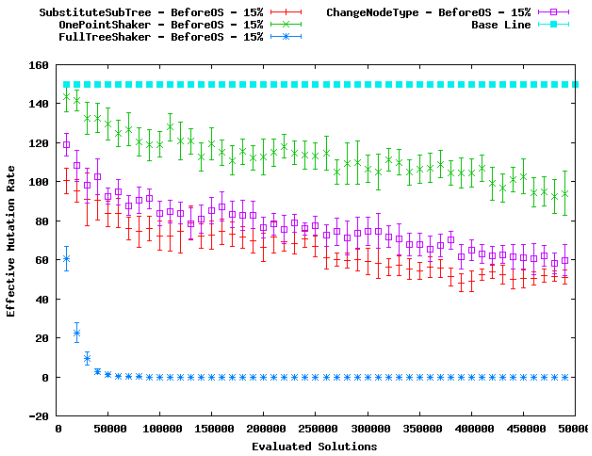

Figure 1: Number of mutated individuals in the population of manipulation operators when applied before offspring selection (averages over 20 runs for each mutation operator; error bars indicate the 95% confidence interval.)

4.2. Training Error

In Figure 2 the best training and validation quality of solutions generated by genetic programming with mutation before offspring selection are shown. Each data point is the best training/validation quality (y-axis) found after the number of evaluated solutions (x-axis) averaged over 20 independent runs (error bars indicate the 95% confidence interval). After the first few generations the algorithm generations solutions that are increasingly worse on the validation set, while the quality on the training set steadily increases. In Figure 3 the same effect can be observed, however the effect is not so strong when mutation is applied after offspring selection. The overfitting effect can be attributed to the linear scaling operation applied to all solutions. Only the training data-set is considered for the scaling operation, this leads to bad performance on the validation set and also on the test set. Based on this observation linear scaling of solution candidates should always be used in combination with an internal validation data-set for model selection. Figures 2 and 3 also show that there is no significant difference in the final solution quality of the two algorithm variants if only the training quality is considered.

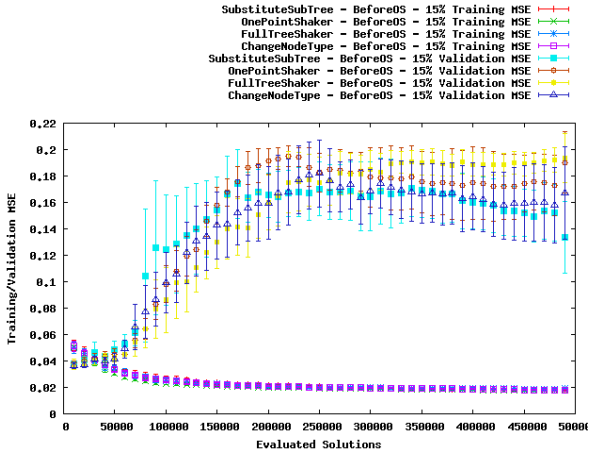

Figure 2: Best training and validation quality averaged over 20 independent runs for each mutation operator when applied before offspring selection (error bars indicate the 95% confidence interval.)

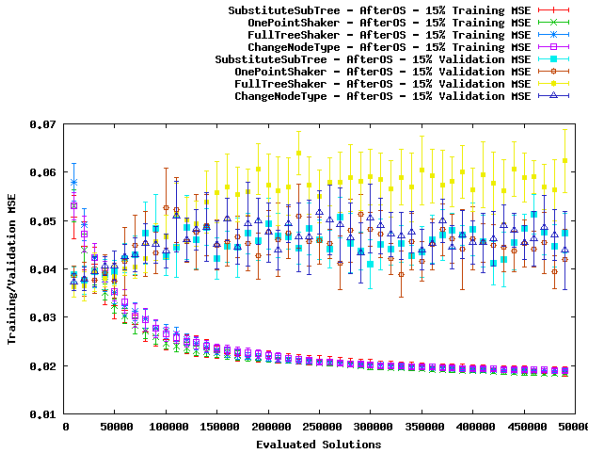

Figure 3: Best training and validation quality averaged over 20 independent runs for each mutation operator when applied after offspring selection (error bars indicate the 95% confidence interval).

4.3. Generalization Error

For the symbolic regression task the quality on unseen data is more relevant than the solution quality achieved while training because it is a better estimate for the quality of the model when applied to unseen data.

Table 4 shows the generalization error of the models selected as final result (best model on the

validation set over the whole run) for all mutation operators and different mutation rates (5%, 10%, 15%). The values are averages over 20 independent GP runs. Statistical significance was tested with the two-tailed non-parametric Mann-Whitney-U test (null hypothesis: the medians are equal). Significant results are given in bold face (p-Value < 0.01). Overfitting on the validation set occurs in both configurations for all mutation operators. This can be seen by the fact that the generalization error is very high in comparison to the average training MSE of the best on validation solutions which is 0.08 and the average validation MSE which is 0.03 over all runs.

Table 4: Generalization error (mean squared error on test set) of models generated by the GP variants averaged over 20 independent runs.

Operator	Rate	Test MSE (beforeOS)	Test MSE (afterOS)
ChangeNodeType	0.05	0.240	0.252
	0.15	0.253	0.235
	0.25	0.276	**0.121**
FullTreeShaker	0.05	0.265	0.219
	0.15	0.220	0.208
	0.25	0.205	0.216
OnePointShaker	0.05	0.263	0.208
	0.15	0.221	0.170
	0.25	0.281	**0.165**
SubstituteTree	0.05	0.237	0.209
	0.15	0.251	**0.141**
	0.25	0.204	0.165

Table 5: Best solution (on validation set) generation (median value of 20 independent runs).

Operator	Rate	Generation (beforeOS)	Generation (afterOS)
ChangeNodeType	0.05	4	**10.5**
	0.15	4	**31.5**
	0.25	5	**28**
FullTreeShaker	0.05	4	**6**
	0.15	4	4.5
	0.25	3	**11**
OnePointShaker	0.05	5	**26**
	0.15	3.5	**23.5**
	0.25	4	**22**
SubstituteTree	0.05	4	**22.5**
	0.15	3	**28.5**
	0.25	3.5	**23.5**

In Table 5 the median value of the generation when the best solution on the validation set was found is shown. From the table it can be seen that if mutation is applied before offspring selection the best solution is found in the first few generations and no improved solution is found over the rest of the GP run. If mutation is applied after OS this effect is reduced.

These observations strongly suggest that mutation should be applied after offspring selection in order to reduce overfitting if linear scaling is used. The results also raise the following questions which should be pursued in further experiments. What causes the observed overfitting effect with linear scaling and which countermeasures effectively prevent it? Does linear scaling cause a loss of genetic diversity and premature convergence? Further experiments should include an analysis of the genetic diversity in order to get an insight about the effects of linear scaling on the genetic diversity.

5. CONCLUSIONS

In this paper we have analyzed a variant of genetic programming with offspring selection where mutation is applied after offspring selection. In the original formulation of offspring selection it is applied after mutation. Both variants were used to solve a symbolic regression problem. In the experiments we observed that even when offspring selection is applied after the mutation operator a large percentage of mutated individuals are accepted into the next generation. This means that mutation has a high probability to improve solution quality. This is a bit surprising because mutations are usually undirected random changes and should intuitively have a low probability to improve solution quality. We attribute this behavior to the linear scaling operator which is a local optimization step before evaluation and improves the model quality on the training data-set. We also observed that the *FullTreeShaker* manipulation operator almost never improves the solution quality and thus has no effect when it is applied before offspring selection.

The results show that linear scaling leads to overfitting on the training data-set. This is clearly seen when the solution candidates are evaluated on the test data-set. Thus linear scaling should be used in combination with a model selection step to make sure that the final solution is not overfit on the training, for instance by tracking the best solution on an internal validation set.

Interestingly the overfitting effect of the linear scaling operation is less problematic if mutation is applied after offspring selection. In some configurations the application of mutation after offspring selection produced significantly better results regarding the generalization error.

The results of our experiments suggest that the overfitting effect is related to a kind of premature convergence to highly fit but small solutions. The cause for this is unfortunately not directly apparent from our experiments. In further research the experiments should also include analysis of the genetic diversity in order to get an insight about the effects of linear scaling on the genetic diversity.

From the results described in this work we conclude that if linear scaling is used in OSGP then mutation should be applied after offspring selection to reduce the probability of overfitting.

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

Affenzeller, M. and Wagner, S., 2005. *Offspring selection: A new self-adaptive selection scheme for genetic algorithms*. Proceedings of ICANNGA 2005, pp. 218–221. 21st–23rd March 2005, Coimbra, Portugal.

Affenzeller, M., Winkler, S.M., Wagner, S. and Beham, A., 2009. *Genetic algorithms and genetic programming – Modern concepts and practical applications*. Boca Raton: CRC Press.

Beyer, H. G., and Schwefel, H.-P., 2002. Evolution Strategies: A Comprehensive Introduction. In Natural Computing, 1:1, pp. 3 – 52.

Harries, K. and Smith, P., 1997. Exploring alternative operators and search strategies in genetic programming. In J. R. Koza, et al., editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pp. 147 – 155, Stanford University, CA, USA, 1997.

Holland, J. H., 1975. *Adaption in Natural and Artificial Systems*, University of Michigan Press, Ann Harbor.

Keijzer, M., 2004. *Scaled Symbolic Regression*. Genetic Programming and Evolvable Machines 5:3 (September 2004), pp. 259 – 269.

Koza, J. R., 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press.

Koza, J. R., 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press.

Koza, J. R., 2010. Human-competitive results produced by genetic programming. In J. Miller, et al., editors, *Tenth Anniversary Issue: Progress in Genetic Programming and Evolvable Machines*, Vol. 11, No. 3-4, pp. 251 – 284. Springer Netherlands.

Langdon, W. B. and Poli, R., 2002. *Foundations of Genetic Programming*. Springer-Verlag.

Luke, S. 2000. *Two fast tree-creation algorithms for genetic programming.* In IEEE Transactions on Evolutionary Computation 4:3 (September 2000), 274-283. IEEE.

Luke, S. and Spector L., 1997. A comparison of crossover and mutation in genetic programming. In J. R. Koza, et al., editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pp. 147 – 155, Stanford University, CA, USA, 1997.

Poli, R., Langdon, W. B., and McPhee N., 2008. *A field guide to genetic programming.* Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk, (with contributions by J. R. Koza).

Wagner, S., 2009. *Heuristic Optimization Software Systems - Modeling of Heuristic Optimization Algorithms in the HeuristicLab Software Environment*. Thesis (PhD), Johannes Kepler University Linz, Austria.

AUTHORS BIOGRAPHIES

GABRIEL KRONBERGER is a research associate at the UAS Research Center Hagenberg. His research interests include genetic programming, machine learning, and data mining and knowledge discovery. Currently he works on practical applications of data-based modeling methods for complex systems within the Josef Ressel Centre Heureka!.

STEPHAN M. WINKLER received his PhD in engineering sciences in 2008 from Johannes Kepler University (JKU) Linz, Austria. His research interests include genetic programming, nonlinear model identification and machine learning. Since 2009, Dr. Winkler is professor at the Department for Medical and Bioinformatics at the Upper Austria University of Applied Sciences (UAS), Campus Hagenberg.

MICHAEL AFFENZELLER has published several papers, journal articles and books dealing with theoretical and practical aspects of evolutionary computation, genetic algorithms, and meta-heuristics in general. In 2001 he received his PhD in engineering sciences and in 2004 he received his habilitation in applied systems engineering, both from the Johannes Kepler University of Linz, Austria. Michael Affenzeller is professor at the Upper Austria University of Applied Sciences, Campus Hagenberg, and head of the Josef Ressel Center *Heureka!* at Hagenberg.

MICHAEL KOMMENDA finished his studies in bioinformatics at Upper Austria University of Applied Sciences in 2007. Currently he is a research associate at the UAS Research Center Hagenberg working on data-based modeling algorithms for complex systems within *Heureka!*.

STEFAN WAGNER his PhD in engineering sciences in 2009 from Johannes Kepler University (JKU) Linz, Austria; he is professor at the Upper Austrian University of Applied Sciences (Campus Hagenberg). Dr. Wagner's research interests include evolutionary computation and heuristic optimization, theory and application of genetic algorithms, machine learning and software development.