SIMULATION-BASED SET-UP TIME OPTIMIZATION USING SIM# AND HEURISTICLAB

Johannes Karder[a], Andreas Scheibenpflug[b], Andreas Beham[c], Stefan Wagner[d], Michael Affenzeller[e]

[a],[b],[c],[d],[e]Heuristic and Evolutionary Algorithms Laboratory
University of Applied Sciences Upper Austria, Softwarepark 11, 4232 Hagenberg, Austria
[b],[c],[e]Institute for Formal Models and Verification
Johannes Kepler University, Altenbergerstraße 69, 4040 Linz, Austria

[a]jkarder@heuristiclab.com, [b]ascheibe@heuristiclab.com, [c]abeham@heuristiclab.com, [d]swagner@heuristiclab.com, [e]maffenze@heuristiclab.com

ABSTRACT
Model building is a fundamental task in simulation-based optimization. In this paper we demonstrate the application of Sim# in combination with HeuristicLab to optimize set-up times of arbitrary machinery. On top of Sim#, custom simulation extensions have been implemented and are used to create a simulation model of real world machinery. These extensions enable the design of simulation components that can be reused within different simulation models. This allows to easily create multiple model implementations that reflect different designs of a machine by using a combination of already existing and adapted components. The resulting model is used as evaluation function for the optimization inside HeuristicLab.

Keywords: simulation-based optimization, set-up time optimization, sim#, heuristiclab

1. INTRODUCTION
Sim# (Beham et al. 2014) is a .NET port of SimPy (Matloff 2008), a process-based discrete event simulation framework. Implemented in the C# programming language, it can easily be integrated in other .NET applications such as HeuristicLab (Wagner et al. 2014). It provides an intuitive way to simulate processes as events. Sim# contains an event queue which allows fast evaluation and computation of processes that make up the simulation model. Simulation-based heuristic optimization methods often require many fitness evaluations that employ a simulator. Therefore, fast simulations have to be implemented so that evaluation phases can be executed in a reasonable amount of time.
HeuristicLab (HL) provides different ways of integrating external software such as simulation models in optimization algorithms, e.g. by communicating with other applications using Google's Protocol Buffers (https://developers.google.com/protocol-buffers/) or by using other plug-ins specifically designed for this purpose (e.g. MATLAB and Scilab problems). In this publication the authors created a new customized plug-in for simulation-based set-up time minimization. It features a simulation model which is created with a new simulation architecture based on Sim#.
The focus is to extend Sim# in such a way that it is possible to model reusable software components for simulation models and use those components to create similar but individual model implementations which represent e.g. different machine designs.
In Section 2, previous related work is presented. Section 3 explains the architecture of a general simulator. In Section 4, insights into the actual implementation of the used simulation model are given. The optimization is explained in Section 5 and the achieved results are presented in Section 6. Section 7 concludes the paper with gained domain knowledge and an outlook about future work.

2. RELATED WORK
Previous work including the use of HeuristicLab has been done by Affenzeller et al. (2007) to tune simulation parameters by applying simulation-based and heuristic optimization methods. Bruzzone et al. (2011) conducted a real case study about short period production planning in manufacturing systems by combining a flexible simulation model, genetic algorithms and dispatching rules. Beham et al. (2012) also worked on alternative ways for information exchange between simulation and optimization processes. In (Beham et al. 2014), Sim# was used to implement a simple supply chain simulation. HeuristicLab was then used to optimize the simulation's parameters. A similar set-up time optimization in the same domain has already been done by Karder et al. (2015) without using an underlying simulation framework such as Sim#. Here the simulator was written and continuously extended with new functionality without reusability in mind. This made it very difficult to implement additional specifications and new features which were added occasionally. A specific optimization problem was also created for HeuristicLab, in which three different optimization aspects, namely the job execution order, the storage organization and different (un-)loading strategies were optimized. Set-up times could be reduced by approximately 20.21 %.

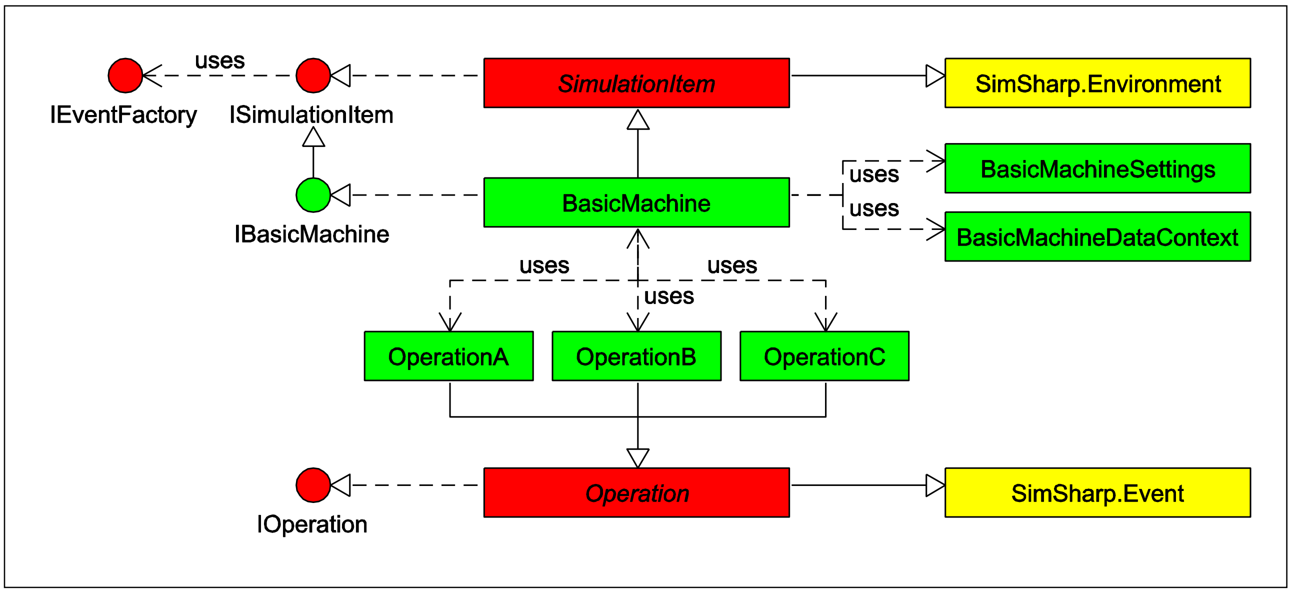

Figure 2: The simulator's architecture showing parts of Sim# (yellow), classes from the implemented extensions (red) and from the implemented simulation (green).

3. SIMULATOR ARCHITECTURE

The simulator is built with modularity in mind. With Sim# as underlying simulation framework, the focus of the implementation is not the creation of a simulation framework itself, but rather the application and extension of already existing simulation infrastructure and the design of modular components which can be reused in different versions of a simulation that belong to a certain kind of simulation domain. Instead of creating three separate implementations, a basic version of the machine can be modeled and extended with different functionality by deriving or adding new operations. This leads to a three-layered architecture, where a concrete simulation builds upon specific Sim# extensions, which again build upon the simulation framework itself. This architecture is shown in Figure 1. Using this concept, the changes in the machine's designs should only be minor, since atomic operations are often implemented for a specific kind of machinery.

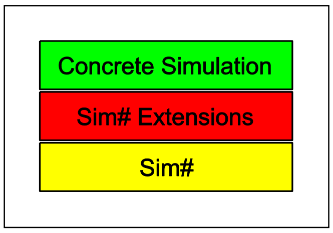

Figure 1: The three simulation layers.

Each layer's most important classes/interfaces and their relations are shown in Figure 2. The base for any simulated machine is a SimulationItem, which represents the execution environment and is shown in Listing 1.

```
public abstract class SimulationItem
 : SimSharp.Environment, ISimulationItem {
 private readonly IEventFactory eventFactory;
 public IEventFactory EventFactory
  { get { return eventFactory; } }

 protected SimulationItem() {
  eventFactory = new EventFactory();
 }

 public SimulationResult Run() {
  var op = CreateSimulationOperation();
  Process(op);
  base.Run();

  var result = new SimulationResult {
   Duration = Now - StartDate
  };
  return result;
 }

 public abstract IEnumerable<SimSharp.Event>
  CreateSimulationOperation();
}
```

Listing 1: The abstract base class SimulationItem from which all simulated environments should derive.

It implements the ISimulationItem interface, which provides access to an EventFactory and a Run method. The EventFactory can be used to register to different events and allows to execute according event handlers after an operation, its dependencies or its sub-operations have been executed. This concept allows to easily extend a simulator with debugging, logging or visualization features. The Run method is called to execute the simulation.

Any machine deriving from SimulationItem owns a context and has specific settings. Contexts represent the current state of a machine and its components and are composites of multiple properties and sub-contexts, which are accessible and can be modified by the machine and its operations. Operations are always created with the specific context they modify. Each operation also has access to the global context of the environment. Settings define different properties of the machine, e.g. sizes and speeds. Unlike the context, settings are read-only and therefore cannot be modified during a simulation run.

Simulated operations are always designed for a specific environment. By implementing (and deriving from) an ISimulationItem, all operations from the simulated environment have to be defined. In the presented architecture it is possible to reuse operations defined for environment A in another environment B. The limiting factor here is that an operation can only be reused in B, if B is a subtype of A. This constraint has to be satisfied, because the operation was originally designed for A and possibly requires contexts and settings that have also been designed for A.

Each machine operation is represented by a separate class that derives from Operation which implements IOperation. All operations have to declare their dependencies (i.e. operations which have to be executed before the actual operation) and sub-operations (i.e. all operations that are executed after the actual operation). By calling Operation.Execute, only the actual operation is executed and its dependencies and sub-operations are ignored. Operation.ExecuteAll calls all dependencies, the actual operation and all sub-operations (in that order).

Different information such as the start time, the duration and the description are set by all operations and passed to the EventFactory in form of OperationInformation instances. Each SimulationItem defines an abstract CreateSimulationOperation method that represents the entry point of the simulation. This method is called within the Run method and builds the operation tree, i.e. yields all necessary operations as well as its dependencies and sub-operations. Listing 2 shows the Operation class.

```
public abstract class Operation
 : SimSharp.Event, IOperation {
 private readonly IDataContext ctx;

 protected SimulationItem Env
  { get { return (SimulationItem)Environment; } }

 public IDataContext DataContext
  { get { return ctx; } }

 public virtual IEnumerable<SimSharp.Event>
  Dependencies { get { yield break; } }
 public virtual IEnumerable<SimSharp.Event>
  SubOperations { get { yield break; } }

 public OperationInformation
  DependencyOperationInformation { get; set; }
 public OperationInformation
  OperationInformation { get; set; }
 public OperationInformation
  SubOperationInformation { get; set; }

 protected Operation(SimSharp.Environment env,
                     IDataContext ctx = null)
  : base(env) {
  this.ctx = ctx;
  DependencyOperationInformation = new ...
  OperationInformation = new ...
  SubOperationInformation = new ...
 }

 public virtual IEnumerable<SimSharp.Event>
  Execute() { yield break; }

 public virtual IEnumerable<SimSharp.Event>
  ExecuteAll() {
  var depOpStart = Env.Now;
  yield return Env.Process(Dependencies);

  DependencyOperationInformation.Duration =
   Env.Now - depOpStart;
  Env.EventFactory
     .FireOperationsDependenciesExecuted(this);

  OperationInformation.StartTime = Env.Now;
  yield return Env.Process(Execute());

  OperationInformation.Duration =
   Env.Now - OperationInformation.StartTime;
  Env.EventFactory.FireOperationExecuted(this);

  var subOpStart = Env.Now;
  yield return Env.Process(SubOperations);

  SubOperationInformation.Duration =
   Env.Now - subOpStart;
  Env.EventFactory
     .FireOperationsSubOperationsExecuted(this);
 }
}

public abstract class Operation<TEnvironment>
 : Operation
 where TEnvironment : SimulationItem {
 protected new TEnvironment Env
  { get { return (TEnvironment)base.Env; } }

 protected Operation(TEnvironment env,
                     IDataContext ctx = null)
  : base(env, ctx) { }
}
```

Listing 2: Implementation of the base class for each specific operation.

The resulting operation tree is traversed and executed by the environment. Each operation is defined in its own class. This allows operations to be used in more than one environment. Each environment (i.e. each machine) contains methods to create the operations it can execute.

This architecture allows developers to build simulations with single components in a modular way. The idea is to separate simulation definition (i.e. what operations are available) and operation definition (i.e. what is done when executing an operation). In Figure 2 a sample machine is described by the IBasicMachine interface, which defines all operations, the context and the settings that can be used. Each operation (A, B, C) represents actions and their necessary dependencies/sub-operations. The BasicMachine class implements IBasicMachine and is responsible for creating operation instances. By extending from BasicMachine, each operation creation can be overridden and alternative operations that represent different features can be included.

4. MODEL IMPLEMENTATION

The real world machinery is transformed into a static, deterministic, discrete simulation model. All machine operations as well as their dependencies and sub-operations are implemented to provide a detailed model of the real world system. The operations are built with reusability in mind. It should be possible to easily create alternative

model implementations with different features by extending a basic version of the simulation and adding specialized operations.
A new BasicMachine derives from SimulationItem and implements the IBasicMachine interface. Listing 3 shows the BasicMachine class.

```
public class BasicMachine
 : SimulationItem, IBasicMachine {
 // ...
 public override IEnumerable<SimSharp.Event>
  CreateSimulationOperation() {
  return new SimulationOperation<BasicMachine>
   (this).ExecuteAll();
 }
 // ...
 public virtual IEnumerable<SimSharp.Event>
  MoveShuttleHome(
   BasicComponentDataContext dataContext) {
    return new MoveShuttleHome<BasicMachine>
     (this, dataContext).ExecuteAll();
 }
 // ...
}
```

Listing 3: An excerpt of the BasicMachine class showing the creation of a MoveShuttleHome operation.

The interface specifies methods to create all Operations that are available in the basic implementation of the machine, e.g. the MoveShuttleHome operation that is shown in Listing 4.

```
public class MoveShuttleHome<TEnvironment>
 : Operation<TEnvironment>
 where TEnvironment : BasicMachine {
 protected new
  BasicComponentDataContext DataContext {
   get { return
    (BasicComponentDataContext)base.DataContext;
   } }

 public MoveShuttleHome(TEnvironment env,
  BasicComponentDataContext ctx)
  : base(env, ctx) { }

 public override IEnumerable<Event> Dependencies {
  get { yield return Env.Process(
   Env.SwitchMagnetState(DataContext,
                         MagnetState.MagnetOff));
  } }

 public override IEnumerable<SimSharp.Event>
  Execute() {
  var shuttle = DataContext.Shuttle;

  if (shuttle.Position.IsAlmost(0.0)) {
   shuttle.Position = 0.0;
   yield break;
  }

  yield return Env.Timeout(
   Env.Settings.ShuttleMove.CalculateMovementTime(
     shuttle.Position));
  OperationInformation.Distance = shuttle.Position;
  shuttle.Position = 0.0;
 }
}
```

Listing 4: Implementation of the MoveShuttleHome operation that shows how dependencies and operation logic can be added.

All methods are implemented in BasicMachine and return an according instance of the required operation class. This creates two possibilities for extending the machine design. On the one hand it is possible to derive from BasicMachine and override particular methods to return new operations. Those extended operations should derive from the original operation and can introduce new dependencies, sub-operations and logic by overriding as required. On the other hand new interfaces can be added to a new class derived from BasicMachine which introduce new methods for creating new operations.
The operation sequence of a simulated environment is represented by the operation tree. This tree is constructed by the operations and their dependencies/sub-operations themselves. The entry point of the simulation is the Run method defined in the ISimulation interface. It creates a SimulationOperation (root operation) and generates the operation tree by calling ExecuteAll on that operation. The operation tree is traversed when the environment processes all resulting operations.
An improved machine could for example require additional steps before the shuttle is moved to its home position and therefore, an AdvancedMoveShuttleHome operation could be derived. The new operation would then be returned by the respective environment method and yield additional dependency operations.

5. OPTIMIZATION

The optimization focuses on the set-up time minimization of a machine that sequentially processes multiple jobs. Each job requires a certain set of tools that are stored in a storage area and must be set up in the machine's working area before processing can be started. The machine automatically loads and unloads required components.
A new HeuristicLab problem which is targeted to work with already existing evolutionary algorithms such as genetic algorithms is used to optimize the job sequence and the storage organization. The problem derives from Heuristiclab's BasicProblem, which makes it easy to implement the necessary evaluation and analysis logic without the need to create custom operators. The required problem data is stored in a TJOData instance. A TJOData object represents the scenario of the machine, i.e. available tools and jobs as well as the machine settings. This TJOData instance is then passed on to each solution evaluation. Solutions are represented by a multi-encoding that represents the identified optimization aspects. It consists of three permutations that are used in combination with the TJOData to construct the actual solution. The first permutation represents the order of jobs the machine has to process. Depending on the processing order of the jobs, the simulated execution time can increase or decrease. The other two permutations are used to create the storage layout of the machine. The way that components are sorted plays an important role when executing job after job. The indices of the tools are fixed within the TJOData of the problem instantiation. Each storage rack is assigned exactly one tool (or none). The actual storage

configuration can be created with the respective permutation by rearranging the tools by their index in the permutation. Figure 3 shows how the actual storage configuration is constructed by combining the respective permutation and the data that is available within the TJOData object. In this example tool 0 (T_0) was originally assigned to rack 0 (index 0 contains 0). The upper rack assignments permutation now dictates that in rack 0 tool 3 can be found (index 0 contains 3). If there are less tools (n) than racks (m), all numbers greater than n represent void tools that are used to create empty racks.

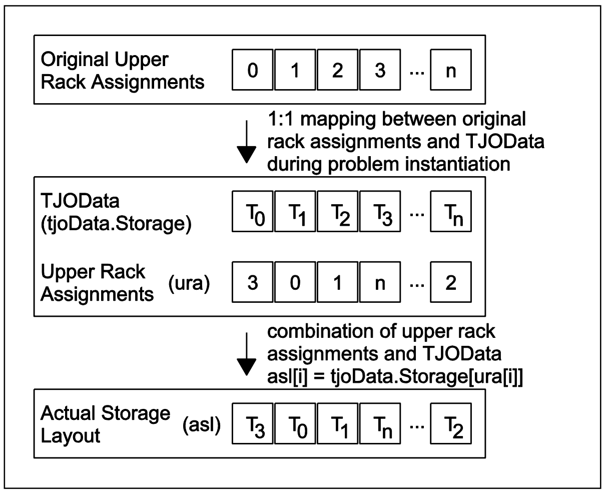
Original Upper Rack Assignments
0 1 2 3 ... n
1:1 mapping between original rack assignments and TJOData during problem instantiation
TJOData (tjoData.Storage)
T0 T1 T2 T3 ... Tn
Upper Rack Assignments (ura)
3 0 1 n ... 2
combination of upper rack assignments and TJOData asl[i] = tjoData.Storage[ura[i]]
Actual Storage Layout (asl)
T3 T0 T1 Tn ... T2

Figure 1: The procedure for creating the actual storage configuration.

For all experiments, a genetic algorithm with offspring selection (Affenzeller and Wagner 2005) is used. Selection scheme and replacement strategy are set to linear rank selection (LR) and 1-elitism, respectively. For all permutations, crossover and mutation operators have been selected carefully. For recombination, the OrderCrossover2 (OX2) (Davies 1985; Affenzeller et al. 2009) was chosen since it is known to preserve order rather than position within a permutation. For mutation, the InsertionManipulator (IM) was used.
Each time a solution has to be evaluated, a new simulation instance is created specifically for this evaluation to allow parallel execution of optimization algorithms. Therefore, efficient algorithm usage is assured. Quality is measured in total seconds of simulation time, i.e. the simulated time the machine took to load and unload tools for all jobs.

6. RESULTS

Three different problem configurations were tested. In the first configuration, five different jobs were defined. This scenario can be found in rather small manufacturers or manufacturers with low fluctuation in production cycles. The second configuration represents middle class manufacturers and contains ten different jobs together with a medium filled storage. This scenario should simulate a daily production cycles of medium sized manufacturers that are not restricted to just five different jobs. The last configuration features 20 different jobs and a storage that contains a large variety of different tools. This scenario represents large manufacturers or manufacturers with a high fluctuation in production. In each problem configuration, only the least amount of tools and the machine's default settings were used. Each problem instance was tested within an experiment containing a batch run with 10 repetitions. The results of the best parameter combinations are shown in Table 1.

Table 1: Selected algorithm parameters and achieved results for each problem instance.

Parameter	**Prob. 1**	**Prob. 2**	**Prob. 3**
Population Size	1000	1000	1000
Max. Generations	100	100	100
Elites	1	1	1
Selection Operator	LR	LR	LR
Crossover Operator	OX2	OX2	OX2
Mutation Operator	IM	IM	IM
Mutation Prob.	5 %	5 %	5 %
Comp. Factor	1	1	1
Success Ratio	1	1	1
Max. Sel. Press.	100	200	200
Result			
Worst Quality	546.041	944.004	2629.342
Best Quality	476.568	832.955	2354.767
Avg. Best Quality	476.621	833.449	2356.019
Avg. Eval. Sol.	248700	611490	830640

The results shown in Table 1 were selected based on the number of runs that achieved the best found solution in each configuration and the number of evaluated solutions. The only difference between the configurations that is worth mentioning is the change in the number of evaluated solutions. The achieved quality values were quite similar in each algorithm setup. The simulated processing times for problem instance 1, 2 and 3 could be reduced by approximately 12.72 %, 11.76 % and 10.44 %, respectively.

7. CONCLUSION AND OUTLOOK

Despite the machine's current constraints, set-up time improvements of more than 10 % have been achieved. By extending the machine in its design, more optimization potential can be created and set-up times could be reduced even more. Currently, only one tool can be stored in one storage rack. This limits the number of different storage layouts and the number of tools that can be used. A redesign could e.g. allow multiple tools to be stored within one storage rack. Adding new unloading strategies would allow to move the tools in different ways, e.g. back to their original location or to other positions. This approach can be extended by using priority rules to select a specific rack for each tool that needs to be unloaded.

ACKNOWLEDGMENTS

The work described in this paper was conducted within the NPS (Sustainable Production Steering) project and funded by the Austrian Research Promotion Agency (FFG).

REFERENCES

Affenzeller, M., Wagner, S., 2005. Offspring Selection: A New Self-Adaptive Selection Scheme for Genetic Algorithms. *Adaptive and Natural Computing Algorithms*, 218–221. Springer.

Affenzeller, M., Kronberger, G., Winkler, S., Ionescu, M., Wagner, S., 2007. Heuristic Optimization Methods for the Tuning of Input Parameters of Simulation Models. *Proceedings of I3M 2007*, 278–283. October 4-6, Bergeggi, Italy.

Affenzeller, M., Winkler, S., Wagner, S., Beham, A., 2009. *Genetic Algorithms and Genetic Programming - Modern Concepts and Practical Applications*. Chapman & Hall/CRC. ISBN 978-1584886297.

Beham, A., Pitzer, E., Wagner, S., Affenzeller, M., Altendorfer, K., Felberbauer, T., Bäck, M., 2012. Integration of Flexible Interfaces in Optimization Software Frameworks for Simulation-Based Optimization. *Companion Publication of the 2012 Genetic and Evolutionary Computation Conference, GECCO'12 Companion*, 125–132. July, Philadelphia, PA, USA.

Beham, A., Kronberger, G., Karder, J., Kommenda, M., Scheibenpflug, A., Wagner, S., Affenzeller, M., 2014. Integrated Simulation and Optimization in HeuristicLab. *Proceedings of I3M 2014*, 418–423. September 10-12, Bordeaux, France.

Bruzzone, A., Longo, F., Nicoletti, L., Diaz, R., 2011. On the short period production planning in industrial plants: A real case study. *Proceedings of the 23rd European Modeling and Simulation Symposium EMSS 2011*, 782–791. September 12-14, Rome, Italy.

Davis, L., 1985. Applying Adaptive Algorithms to Epistatic Domains. *Proceedings of the International Joint Conference on Artificial Intelligence*, 162–164. August, Los Angeles, CA, USA.

Karder, J., Scheibenpflug, A., Wagner, S., Affenzeller, M., 2015. Optimizing Set-up Times using the HeuristicLab Optimization Environment. Accepted to be published in *Proceedings of the Computer Aided Systems Theory – Eurocast 2015*, February 8-13, Las Palmas de Gran Canaria, Spain.

Matloff, N., 2008. Introduction to Discrete-Event Simulation and the SimPy Language. *Davis, CA. Dept of Computer Science. University of California at Davis.* Available from: http://heather.cs.ucdavis.edu/~matloff/156/PLN/DESimIntro.pdf [accessed 13 July 2015]

Wagner, S., Kronberger, G., Beham, A., Kommenda, M., Scheibenpflug, A., Pitzer, E., Vonolfen, S., Kofler, M., Winkler, S., Dorfer, V., Affenzeller, M., 2014. Architecture and Design of the HeuristicLab Optimization Environment. *Advanced Methods and Applications in Computational Intelligence, Topics in Intelligent Engineering and Informatics Series*, 197–261. Springer.

AUTHORS BIOGRAPHIES

JOHANNES KARDER received his Master in software engineering in 2014 from the University of Applied Sciences Upper Austria and is a research associate at the Research Center Hagenberg. His research interests include algorithm theory and development as well as production planning and logistics optimization. He is a member of the HeuristicLab development team.

ANDREAS SCHEIBENPFLUG received his Master in software engineering in 2011 from the University of Applied Sciences Upper Austria and is a research associate at the Research Center Hagenberg. His research interests include parallel and distributed computing. He is a member of the HeuristicLab architects team.

ANDREAS BEHAM received his Master in computer science in 2007 from from the Johannes Kepler University Linz, Austria, and is a research associate at the Research Center Hagenberg. His research interests include metaheuristic methods applied to combinatorial and simulation-based problems. He is a member of the HeuristicLab architects team.

STEFAN WAGNER received his PhD in technical sciences in 2009 from the Johannes Kepler University Linz, Austria. He is a professor at the University of Applied Sciences Upper Austria, Campus Hagenberg. He is the project manager and head developer of the HeuristicLab optimization environment.

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 University of Applied Sciences Upper Austria, Campus Hagenberg.