Quality-based Software-Selection and Hardware-Mapping as Model Transformation Problem

Sebastian Götz, Johannes Mey, Rene Schöne and Uwe Aßmann
{first.last}@tu-dresden.de, sebastian.goetz@acm.org

Software Technology Group
Technische Universität Dresden

Abstract

In this TTC case, we describe the computation of an optimal mapping from software implementations to hardware components for a given set of user requests as a model transformation problem. Further, contracts specify dependencies between components in terms of non-functional properties. Different solutions of this case can be compared in terms of their validity, performance, scalability and, quality w.r.t. the real optimal deployment.

1 Introduction

One of the goals of software engineering is to enable the development of *efficient* software, i.e., software which provides the best possible utility for the least possible cost. By now, this goal has been addressed at several levels of abstraction: from compiler construction [Kil73] in the 70s to cloud computing [BAB12].

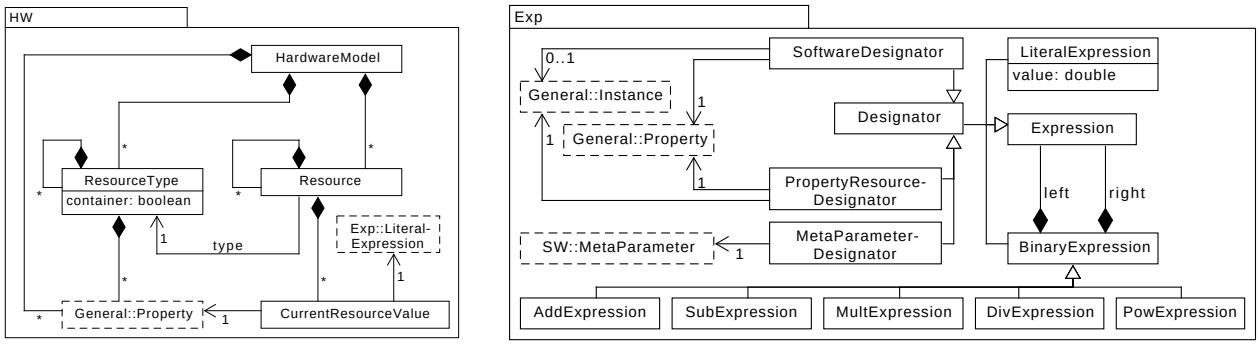
To optimize the tradeoff between utility and cost, at least two interrelated standard problems are typically addressed:

- Resource allocation (aka. register allocation in compilers), i.e., the mapping of software implementations to hardware components leading to the least cost
- Variant selection (aka. instruction selection in compilers), i.e., selecting the software implementation, which provides the best utility

These problems have been investigated for compilers since almost 50 years, where they are called code generator optimizations. However, both problems are strongly intertwined. Together, they form an NP-complete problem with exponential cost, but typically are carried out in phases [WG12]. Considering both problems to be independent from each other allows to use efficient optimization approaches, running in polynomial time [HG06]. But, an immanent problem of solving variant selection and resource allocation in isolation is that invalid solutions might result. Hence, still new approaches to compute solutions for the joint selection and mapping problem, are required.

This case provides a generic metamodel for the combined problem of resource allocation and variant selection. Both problems are interrelated by user requests specifying minimum requirements on the non-functional properties provided (i.e., minimum utility), whilst searching for a selection and mapping, which both maximizes the utility and minimizes the cost. To show progress over state of the art, these approaches need to be evaluated w.r.t. their correctness, performance, solution quality and scalability. Correctness denotes that only solutions, which do not violate the minimum requirements of the users, result. The performance of an approach describes

In: A. Garcia-Dominguez, G. Hinkel, F. Krikava (eds.): Proceedings of the Transformation Tool Contest 2018, Toulouse, France, 29-Jun-2018, published at http://ceur-ws.org



(a) Hardware View.

(b) Expressions View.

Figure 1: First part of the Metamodel. Classes with dashed borders are described in other views, e.g., `General::Property` can be found in Figure 3.

how fast a solution can be computed for a given problem. The solution quality quantifies how close the computed solution is to the optimal solution. Finally, scalability is represented by the size of the largest problem, for which a valid solution can be computed.

We describe a reference implementation using a reference attribute grammar as metamodel [BKWA11] to translate the above described selection and mapping problem into an integer linear program, whose solution is then translated into a deployment description.

2 Case Description

In the following, we first provide a detailed description of the case by specifying the metamodel of source and target model, which in our case share the same metamodel. Next, in Section 2.2, we give an overview of our reference implementation.

2.1 Metamodel description

As mentioned in the previous section, the problem to be solved is selecting variants of software components and a mapping of those to suitable hardware resources based on user requests. To adequately cover this problem, we designed a combined metamodel comprised of four views: hardware, software, expressions, and their combination.

Figure 1a depicts the hardware view, specifying a **HardwareModel**, which is comprised of hierarchically structured **ResourceType**s and **Resource**s as instances of these types. Thus, the hardware model composes static knowledge about resources (types) and runtime knowledge (instances). Resource types have an attribute *container* to specify, if they are able to run software, i.e., are a valid mapping target for software implementations. Moreover, resource types are further characterized by a set of properties. Instances of those types specify concrete values for these properties. As an example, one could define the resource type *CPU* with a **Property** *frequency* having the unit *Mhz* and mark it as a container. An instance of this type would be CPU_0 with a current frequency of 200. A full example is shown in Figure 4a.

Figure 1b depicts the expressions metamodel, which allows to specify simple expressions. The hardware view uses **LiteralExpression**s to specify the current value of a resource property. **BinaryExpression**s are required by the software view to describe formulas for required and provided properties. Designators are variables used in expressions referring either to a meta-parameter, or to a property of an instance. The latter can target both resource and software component instances. An example of such a formula can be seen in Figure 4b at line 12 and is explained later in detail.

Figure 2 shows the **SoftwareModel** with its main element **Component** representing a certain functionality (e.g., sorting). **Implementation**s provide this functionality, requiring other components and/or resources to fulfill their work. For example, a software component *sort* could have the implementations *Radix sort* and *Counting sort*, where the component specifies that it requires a *ReadFile* and a *WriteFile* component to read the unordered list and write the ordered list. The in- and output components have different implementations, too, as sketched in Figure 4b. A **ComponentRequirement** states that the referenced component is called during the execution of this implementation. Requirements for components and resources are specified on type level (i.e.,

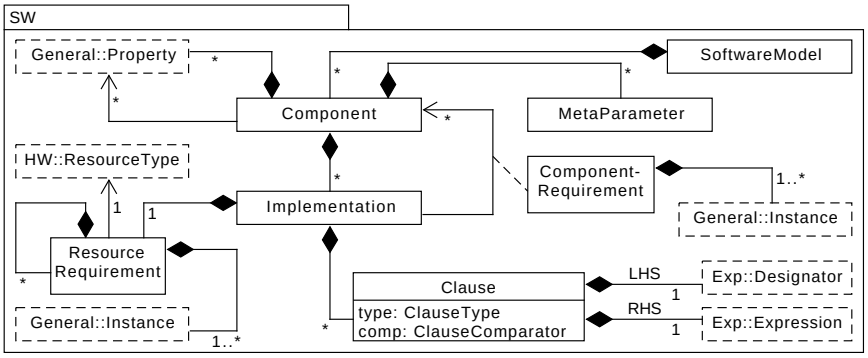


Figure 2: Software part of metamodel. Classes with dashed borders are described in other diagrams.

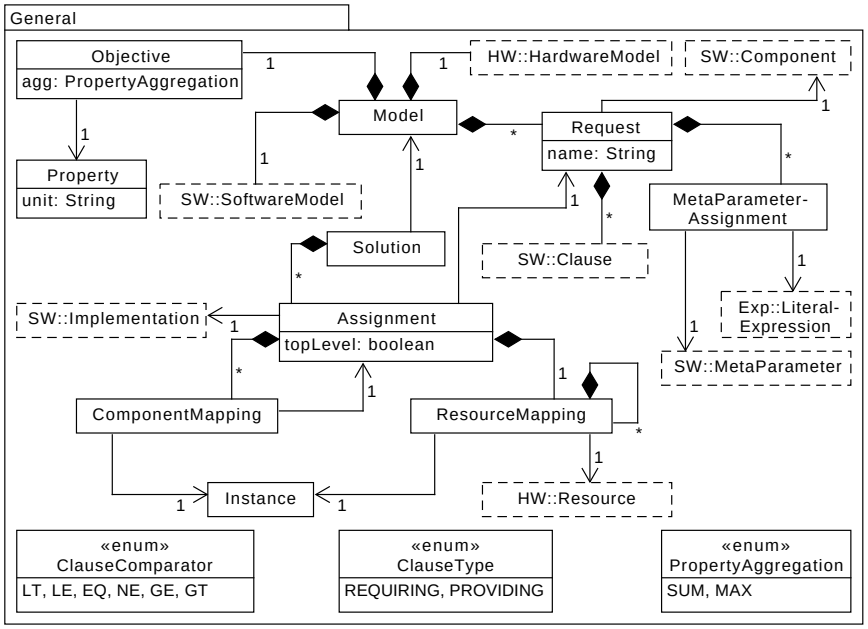


Figure 3: Overview of metamodel. Classes with dashed borders are described in other diagrams.

refer to components and resource types). Their relation to the runtime model is expressed by referencing a set of **Instance**s, which are either **Resource**s or instantiated **Implementation**s. Implementations have a contract specified as a number of **Clause**s describing provisions to and requirements on required instances. A clause comprises left hand side (LHS), which is a designator, comparator (one of $=, \neq, \leq, <, >, \geq$), type (provision or requirement), and right hand side (RHS), which is an expression. For example, the predicted minimum runtime of an implementation can be expressed as a provision clause coupled with a requirement clause, which specifies a certain CPU frequency for which the provision holds. An example contract is shown in Figure 4b. In this example contract, requirements on components as well as resource types are declared (cf. lines 4-7). Later, in line 12, a provision clause is defined with *energy* as the SoftwareDesignator for the left hand side, *EQ* as ClauseComparator and an AddExpression as right hand side.

Figure 3 depicts the complete metamodel showing how the three former parts in Figures 1a, 1b and 2 are connected. Its main concepts are the objective function, user requests, a solution to the problem, and all used enumerations.

The **Objective** specifies how to compute the objective value of a solution, i.e., for which value(s) the problem should be optimized. Therefor, a property to optimize for, and an function how to aggregate all property values are selected.

```
// type and property definitions
container resource type ComputeNode {
  resource type CPU {
    property frequency [Hz]
    property load [%]
  }
  resource type RAM {
    property totalRam [MB]
    property freeRam [MB]
  }
  resource type DISK {
    property totalDisk [MB]
    property freeDisk [MB]
  }
  resource type NETWORK {
    property latency [ms]
    property throughput [kB/s]
  }
}

// resource definitions
resource resource0:ComputeNode {
  resource cpu0_0:CPU { frequency = 200 load = 0 }
  resource ram0:RAM { totalRam = 885 freeRam = 885 }
  resource disk0:DISK { totalDisk = 15472 freeDisk = 15472 }
  resource network0:NETWORK { latency = 14 throughput = 23466 }
}
// ...
```

(a) Hardware model of the scenario generator.

```
component Sort {
  using property quality
  implementation CountingSort {
    requires component input of type ReadFile
    requires component output of type WriteFile
    requires resource compute_resource_0 of type
        ComputeNode
    requires resource cpu_1 of type CPU
    requiring input.quality ≥ 95
    requiring output.quality ≥ 55
    requiring cpu_1.frequency ≥ 2159
    providing quality = 300
    providing energy = ((0.59 * (size ^ 2)) + cpu_1.
        frequency)
  }
  implementation RadixSort { /* ... */ }
}
component ReadFile {
  using property quality
  implementation BufferedInputStream { /* ... */ }
  // ...
}
```

(b) Example of a contract.

Figure 4: Model specification

A **Request** represents a requirement of a user, who specifies what algorithm with which parameters and requirements should be executed. Requests contain their functional requirement by referencing a target software component, constraints on non-functional requirements using clauses, and an abstract characterization of the input (i.e., parameters) using **MetaParameter**s. A meta-parameter describes possible values of the actual parameter. As an example, when passing a list of items to a sort component, a possible meta-parameter for this list is its length.

The class **Solution** is to be computed by the solvers. It contains a number of **Assignment**, each selecting one implementation of the given problem model and assigning every instance in the contract of the implementation to either another assignment or a resource. With this, a concrete implementation is selected for each required component along with their required resources. The value of *topLevel* is *true* if the assignment corresponds to a requested component. An example solution is shown in Figure 6.

A solution is valid, if for each request a) an implementation for the target component is deployed, b) an implementation is deployed for each required component, c) all required (non-functional) property clauses are fulfilled (including the request constraints), and d) at most one implementation is deployed on each resource. A solution is optimal, if it is valid and no other solution has a better objective value.

To preserve readability, the following inheritance relation was not included in the Figures 1a, 1b, 2 and 3: All of Instance, ResourceType, Resource, Component, Implementation, Property, MetaParameter inherit from the class **ModelElement**, which defines a single attribute "Name" of type String, to uniquely identify those model elements, e.g., while parsing the input model.

2.2 Reference Implementation using Reference Attribute Grammars

This section describes the reference implementation for the presented problem based on an attribute grammar-based model-to-text transformation and integer linear programming (ILP). The approach is a variant of Multi-Quality Auto-Tuning [GKP+14, SGAB16] which directly uses the metamodel presented in the previous subsection.

The general approach of our reference solution to the case is depicted in Figure 5. We first transform the problem model into an integer linear program, whose solution is computed using a standard solver. Then, we insert the solution to the problem into the model by interpreting the solution of the solver.

However, there are restrictions on the complexity of the model: Only $\leq, \geq, =$ are permitted as expression clause comparators, resource types can only be nested to a depth of one, and every component may not be

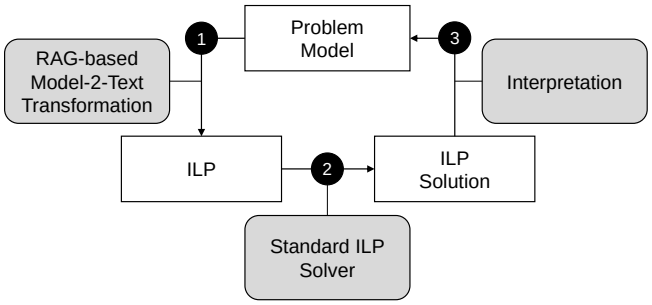


Figure 5: Architecture of reference solution using ILP for solving.

required more than once per request. Models generated for this case as described in Section 3.1 adhere to those restrictions.

2.3 ILP Generation

The textual representation of the selection and mapping problem as an ILP is comprised of an objective function, a set of linear constraints and a set of variables used both in the objective function and the constraints [Man87].

Each combination of a request, resource and implementation is represented by one binary variable in the ILP stating whether or not this implementation is deployed on that resource for the respective request.

The objective function is represented as minimization of the weighted sum of all variables. The weights for each variable are the effect the decision represented by the respective variable has on a selected non-functional property. If, for example, the objective is to optimize the performance, the weights represent the effect on the total runtime.

Furthermore, the ILP is comprised of three types of constraints:

Architectural constraints ensure that each request is fulfilled, exactly one implementation per component is chosen and not more than one implementation is deployed on one resource.

Request constraints ensure that components for each request are chosen such that the requested non-functional properties are provided.

Negotiation constraints guarantee that non-functional requirements are met by depending implementations

The reference implementation uses non-terminal attributes [VSK89] to allow caching of the computed representation of the ILP containing the objective function, used variables and all constraints. Already at the time of generation of those constraints, only valid deployments of configuration on hardware resources w.r.t. contracts of implementations are considered. Variables of invalid combinations are set to be ignored when solving.

We used GLPK [1] to solve the resulting ILP. Calling GLPK was done in two different ways: The first way comprises pretty-printing the subtree to a file, calling the solver via command line, and parsing the result from the created solution file. As a second way, we used the Java-Binding of GLPK[2] to programmatically create constraints, solve the problem and read the solution.

In both cases, the solution can be easily interpreted by iterating over all variables having value one. By this, we can recursively reconstruct the assignments for each request.

The reference implementation is publicly available online[3].

3 Task Description

The general task is to solve the interleaved problems of software variant selection and hardware mapping. To ease implementing an approach to solve this problem, we provide a problem generator (cf. Section 3.1), a concrete set of example problems to be solved (cf. Section 3.2) and a benchmark environment (cf. Section 3.3).

[1] https://www.gnu.org/software/glpk/
[2] http://glpk-java.sourceforge.net/
[3] https://git-st.inf.tu-dresden.de/stgroup/ttc18

Table 1: Parameters of the benchmark generator

Parameter		Description	Value Range
Software variants	s	Number of variants per software component	$1, \ldots, c_{max}$
Number of requests	q		$1, \ldots, n$
Component tree depth	d	depth of the component dependency tree	$0, \ldots, d_{max}$
Available hardware resources	r	Resources as a ratio of the minimally required resources	$1, 1.1, \ldots, h_{max}n$

3.1 Problem Generator

For the presented case, we developed a generator that is able to create random models of arbitrary size. However, the software and hardware component structures are fixed to ensure comparability and some level of realism. The resource types defined in the hardware model are designed to resemble regular computer hardware. A compute node consists of one or more CPUs, RAM memory, disk, and a networking interface. Each of the these components has properties that specify their capabilities. Figure 4a shows the resource types used in the case and their respective properties. The number of resources generated can be specified as a parameter represented as the ratio of the minimal required resources.

The software model has a simple tree structure. That is, all requests have a single meta-parameter (*size*), refer to the same software component and each software component may require other components.

While the generator is able to create sparse trees, for simplicity and comparability reasons, we only generate dense trees with a fixed branching factor of two, such that all possible solutions fulfilling a request require the same amount of software components. For all software components, one property, *quality*, is specified. Requirements and provisions of this property are randomly generated for each implementation.

Finally, for each software component the requirement of exactly one container is generated. Clauses in every implementation use the meta-parameter size as part of a randomly generated formula (i.e., expression).

3.2 Scaling the Problem

To test the scalability of our approach, we generate a set of different problems. The difficulty of a problem is scaled using a set of parameters as shown in Table 1. Naturally, the number of software variants (per software component) and the number of requests have a lower impact on the problem complexity than depths of the component tree and the number of available resources. Since the number of problems that can be generated within the given parameter ranges is large, we propose a set of example problems, which are listed in Table 2.

In addition to the four above mentioned parameters, the generator takes a seed as a fifth parameter to ensure that for the same seed and same values for the other four parameters always the exact same problem is generated.

3.3 The Benchmark Environment

To simplify benchmarking, an easily adoptable process has been defined. Our framework can be used to run the scenarios presented in Section 3.2. If the framework is used, results and timing data are automatically taken and stored for comparison.

3.4 Task Description

The task is to solve the given set of scenarios listed in Table 2. Since the size of the scenarios varies, it might not be possible to find the optimal solution, so any valid solution suffices.

The solution is to be represented within the original problem model by instantiating the respective classes (i.e., Assignment, ComponentMapping and HardwareMapping).

A possible textual representation of the solutions is shown in Figure 6. The depicted exemplary solution is to be interpreted as follows. For the request `request0`, the CountingSort implemententation of Sort shall be used. This configuration is comprised of one hardware mapping (lines 4-9) and two variant selections (lines 11-26). The hardware mapping specifies that the designator (i.e., the variable) `compute_resource_0` is to be mapped to the physical resource named `resource0`. Moreover, the designators like `ram_1` defined within the scope of `compute_resource_0` are mapped to the respective physical subresources. The variant selections specify that for `request_0` the `BufferedInputStream` implementation of `ReadFile` and the `RandomAccessFile` implementation

Table 2: Parameters of the benchmark generator

Scenario	Variants	Requests	Tree Depth	Resources	Impl's	Compute Resources	Description
0	1	1	1	1.0	1	1	minimal
1	2	1	2	1.5	6	5	small
2	10	15	2	1.5	30	68	medium
3	20	20	2	1.5	60	90	large
4	50	50	2	1.5	150	225	huge
5	2	1	2	5.0	6	15	small; much hardware
6	10	15	2	5.0	30	225	medium; much hardware
7	20	20	2	5.0	60	300	large; much hardware
8	50	50	2	5.0	150	750	huge; much hardware
9	2	1	5	1.5	62	47	small; complex software
10	5	10	5	1.5	155	465	medium; complex software
11	10	20	5	1.5	310	930	large; complex software
12	20	50	5	1.5	620	2325	huge; complex software

of `WriteFile` shall be used. For both variant selections, the respective mapping to resources is described analogously to the example in lines 4-9.

Note, that the solution not only contains the information which implementation is assigned to which hardware resources, but also how the software instances from the contract are mapped; thus, the solution describes a tree structure.

In addition to the found solution, the approach is to be evaluated as described in the following section.

4 Evaluation Criteria

To evaluate solutions for the above described case, the following criteria shall be used: correctness, performance, solution quality, and scalability. While correctness, and solution quality are measured on an ordinal scale, performance and scalability are measured on a rational scale. To avoid differences between the measures taken for evaluation due to different hardware, we encourage the authors to evaluate their solution in the SHARE environment[4]. In the following, we describe the criteria to be investigated for a solution and specify the metrics to be used.

4.1 Solution Validity (Correctness)

The most important criteria to be met by a solution to our case is the validity of the computed solution(s). For this, the computed solutions have to be checked against the constraints described in the problem model. The value of the criteria computes as follows:

- 1, if there is at least one invalid problem solution computed by the approach or
- 2, if all solutions computed by the approach are valid.
- 3, if all solutions computed by the approach are optimal.

4.2 Solution Time (Performance)

To assess the performance of a presented approach, the time required to compute a solution for a given problem model shall be measured. Our benchmark framework offers the ability to implement visitor methods, which are called by the problem generation and, thus, can be used to create the problem model representation in the respective solution. Approaches translating the problem model to another technical space, must include the times required to transform the problem and to interpret the solution. As an example, for the reference

[4]https://fmt.ewi.utwente.nl/redmine/projects/grabats/wiki

```
solution {
  request0 -> CountingSort {

    compute_resource_0 -> resource0 {          // assign a compute resource
      ram_1 -> ram0                            // and all contained resources
      network_1 -> network0
      cpu_0 -> cpu0_0
      disk_1 -> disk0
    }

    ReadFile -> BufferedInputStream {          // assign a required component to a resource
      compute_resource_0 -> resource1 {
        disk_1 -> disk1
        ram_1 -> ram1
        network_1 -> network1
        cpu_0 -> cpu1_0
      }
    }

    WriteFile -> RandomAccessFile {
      compute_resource_0 -> resource2 {
        ram_1 -> ram2
        disk_1 -> disk2
        network_1 -> network2
        cpu_0 -> cpu2_0
      }
    }

  }
}
```

Figure 6: Example solution

implementation, the times needed to transform the model to an ILP representation, to solve the ILP and to interpret the ILP solution are summed up.

The value for this criteria is to be represented as a measured runtime. For each measured runtime, its median and standard deviation have to be shown.

4.3 Solution Quality

The selection and mapping problem often has multiple valid solutions, if the objective function is not taken into account, i.e., only the constraints are fulfilled by the respective solutions. To compare different approaches, the quality of computed valid solutions shall be investigated by comparing them to the optimal solution computed using an exact approach, which is available as reference implementation.

The score for this criteria is computed as the minimum of the score of all individual solutions (if more than one solution is returned by the approach), which computes as follows:

- 1, if the solution is the worst solution among all valid solutions,
- 2, if the solution is not the best, but also not the worst among all valid solutions and
- 3, if the solution is the best among all valid solutions

4.4 Scalability

Quality-based Selection and Mapping is a combinatorial optimization problem and, hence, computing the best solution with an exact approach leads to a combinatorial explosion, i.e., does not scale. Approximate approaches can trade solution quality for performance and scalability, i.e., by not ensuring that the best solution is found, but maybe a near-optimal one, the solution can be found faster and larger problems can be solved at all.

To compare different approaches, their scalability shall be assessed in terms of the above described generator parameters (cf. Table 1). For this, the problem generator is to be used with increasing values for the generator parameters in an intelligent way. A table, showing for each parameter set the respective runtime required, is the result. Using this table, the largest solvable problem size, i.e., the largest product of the generator parameters, can be determined.

This criteria is to be represented by the four generator parameters, whose product is the largest among all parameter sets, for which the respective approach could compute a solution.

Acknowledgements

This work has been funded by the German Research Foundation within the Collaborative Research Center 912 Highly Adaptive Energy-Efficient Computing, within the Research Training Group Role-based Software Infrastructures for continuous-context-sensitive Systems (GRK 1907) and the research project "Rule-Based Invasive Software Composition with Strategic Port-Graph Rewriting" (RISCOS) and by the German Federal Ministry of Education and Research within the project "OpenLicht".

References

[BAB12] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future Generation Computer Systems*, 28(5):755 – 768, 2012. Special Section: Energy efficiency in large-scale distributed systems.

[BKWA11] Christoff Bürger, Sven Karol, Christian Wende, and Uwe Aßmann. Reference attribute grammars for metamodel semantics. In Brian Malloy, Steffen Staab, and Mark van den Brand, editors, *Software Language Engineering*, pages 22–41, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[GKP+14] Sebastian Götz, Thomas Kühn, Christian Piechnick, Georg Püschel, and Uwe Aßmann. A models@run.time approach for multi-objective self-optimizing software. In *Proceedings of the 3rd International Conference on Adaptive and Intelligent Systems (ICAIS)*, pages 100–109. Springer, 2014.

[HG06] Sebastian Hack and Gerhard Goos. Optimal register allocation for ssa-form programs in polynomial time. *Information Processing Letters*, 98(4):150 – 155, 2006.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[Man87] CPLEX Users Manual. IBM ILOG CPLEX Optimization Studio, 1987.

[SGAB16] René Schöne, Sebastian Götz, Uwe Aßmann, and Christoff Bürger. Incremental runtime-generation of optimisation problems using rag-controlled rewriting. In *Proceedings of the 11th International Workshop on Models@run.time (MRT16)*, volume 1742, pages 26–34. CEUR-WS.org, 2016.

[VSK89] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. *SIGPLAN Not.*, 24(7):131–145, June 1989.

[WG12] William M. Waite and Gerhard Goos. *Compiler construction*. Springer Science & Business Media, 2012.