Towards Pattern-Based Optimization of Cloud Applications*

Martin Fleck, Javier Troya, Philip Langer, and Manuel Wimmer

Vienna University of Technology, Business Informatics Group, Austria
{lastname}@big.tuwien.ac.at

Abstract. With the promise of seemingly unlimited resources and the flexible pay-as-you-go business model, more and more applications are moving to the cloud. However, to fully utilize the features offered by cloud providers, the existing applications need to be adapted accordingly. To support the developer in this task, different cloud computing patterns have been proposed. Nevertheless, selecting the most appropriate patterns and their configuration is still a major challenge. This is further complicated by the costs usually associated with deploying and testing an application in the cloud.
In this paper, we encode the pattern selection problem as a model-based optimization problem to automatically compute good solutions of configured pattern applications. Particularly, we propose a two-phased approach, which is guided by user-defined constraints on the non-functional properties of the application. In the first phase, a preliminary set of promising solutions is computed using a genetic algorithm. In the second phase, this set of solutions is evaluated in more detail using model simulation. We demonstrate the proposed approach and show its feasibility by an initial case study.

Keywords: Cloud Computing, Goal Modeling, Model Simulation, Genetic Algorithm, Cloud Computing Patterns

1 Introduction

The seemingly unlimited resource offerings and the flexible pay-as-you-go business model are, amongst others, the main driver of the adoption of the cloud computing paradigm. As a result, many different cloud providers have emerged. This has also sparked a major interest in the migration of existing applications to the cloud [17]. Besides the cloud provider selection, adapting the application to make the best out of the cloud provider offerings is often very challenging. Cloud computing patterns [6, 13, 18] have been introduced as cloud provider-independent solutions to reoccurring problems in cloud computing. Developers can use these patterns in their design decisions and operationalize them in the context of a specific cloud provider. This step, however, requires detailed insight of the software architecture, the cloud computing paradigm, the offerings of specific cloud providers, and the usage of the given application. Furthermore, the developers have to deal with a possibly infinite search space of pattern applications and a solution has to satisfy multiple, probably conflicting, objectives [11].

* This work is co-funded by the European Commission under the ICT Policy Support Programme, grant no. 317859.

In this paper, we present a model-based approach aimed to support developers in selecting the most appropriate cloud patterns and their configurations. Particularly, the approach consists of two phases and is guided by user-defined constraints on the non-functional properties of the application. In the first phase, a preliminary set of promising solutions is computed using a multi-objective genetic algorithm which uses estimates to determine the fitness of a solution due to the huge search space. In the second phase, this set of solutions is evaluated in more detail using model simulation to better support the final decision by the user, i.e., selecting the most appropriate solution.

The rest of the paper is organized as follows. In Section 2, we describe our proposed approach as well as the necessary input from the stakeholders. Section 3 showcases the applicability of our approach in a case study, while Section 4 discusses related work. The paper concludes in Section 5 with an outlook on future work.

2 Approach

The central aim of our approach is to find a configuration of patterns that best satisfies the needs of the application stakeholder, i.e., the reason why the application is moved to the cloud in the first place. We therefore provide the stakeholder with a goal modeling language that is capable to express these needs in terms of non-functional properties (NFPs). Based on these goals, we approach the pattern selection problem with two subsequent steps, as shown in Figure 1. In the first step, a multi-objective evolutionary algorithm is used to calculate a preliminary set of good solutions. A solution is a set of configured cloud optimization patterns and evaluated based on estimates on how certain patterns impact properties of the applications. In the second step, each solution returned by the evolutionary algorithm is additionally ranked based on the more detailed analysis performed by model simulation. The resulting ranked set of solutions together with their approximated success to fulfill the goals is then presented to the stakeholders.

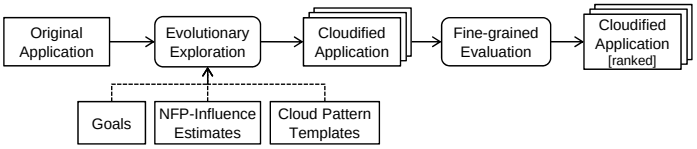

Fig. 1. Approach Overview

2.1 Goal Modeling

Goal modeling originally stems from early phases of requirements engineering, where a goal is an objective for the system from the perspective of a stakeholder. In the goal modeling language we provide, the goals are based on a set of (non-functional) properties. More concretely, a goal defines a target value or target range for a specific property in the context of the software application, e.g., the response time of a request or the utilization of a specific component. These target values must be set in the range of the property under consideration, e.g., utilization can only take floating point values between zero and one. Each goal must be set into the context of a specific workload or usage scenario, as it is not feasible to show that a goal holds in all possible cases. Furthermore, the importance of a goal is given by a numeric priority, whereby a smaller

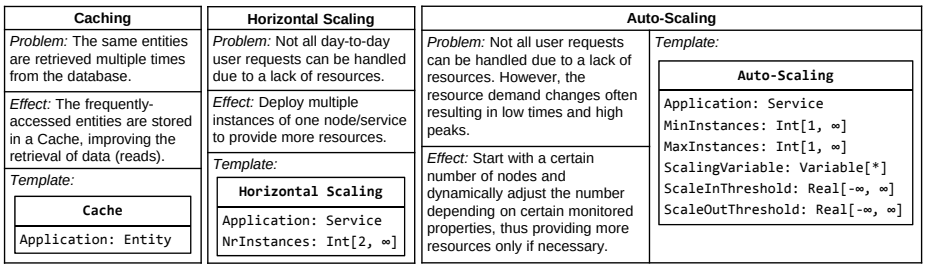

Fig. 2. Example Patterns and their Pattern Templates in a UML class-like notation

number indicates a higher priority. Summarizing, we consider goals to be Boolean conditions concerning NFPs in the context of a software system under a specific workload with a user-defined priority.

Example: The most important objective (priority 1) is that the average response time of a log in-request is less than 2 seconds when ten users log in at the same time.

2.2 Cloud Computing Patterns

Cloud computing patterns provide a generic solution to a reoccuring problem in a specific context in the cloud computing domain and need to be concretized by the developer when used. In the ARTIST project [1] we have collected over 30 of these cloud computing patterns from different sources [6, 13, 18]. In this work we focus on patterns that are applied in order to *optimize* the properties of an application that is to be deployed on the cloud. We therefore assume that the base architecture of the application is already suitable for the cloud and no major architectural refactorings need to be done. To use the informally described patterns in our approach, we translate them into so-called *pattern templates*, which specify where the pattern can be applied and how it can be configured. Figure 2 shows a small excerpt of the collected patterns and the resulting pattern templates.

Caching can be applied on any entity class that is persisted in a datastore, while scaling can be applied on any service class. In horizontal scaling, the number of instances of a service is fixed from the beginning and can range from two instances to a theoretically unlimited number of instances – in practice this number is limited by the specific cloud provider. By contrast, auto-scaling provides a lower and upper bound on the number of instances, and the actual number is adapted during the application runtime based on the value of the *ScalingVariable* and the two variable-specific scaling thresholds. If the value of the variable is less or equal than the specified *ScaleInThreshold*, one service instance is removed; if the variable value is greater or equal than the *ScaleOutThreshold*, an additional instance is created. Any numerical variable which can be evaluated during runtime can serve as auto-scaling variable, e.g., utilization.

When applying a cloud computing pattern in a concrete use case, we create an instance of the respective pattern template, i.e., we provide concrete values for all the parameters defined in the template. The set of the concrete values for a pattern is called a *pattern configuration*. Each applied pattern configuration has an impact on the (non-functional) properties of the system. This impact is usually specific to the software system. Estimations about the gained impact on the properties may be gained from more detailed pattern descriptions, experience, and cloud benchmarking services. An example can be found in Table 1.

2.3 Evolutionary Algorithms

The aim of our approach is to select a sequence of pattern applications that satisfies the goals modeled by the stakeholder. The pattern selection problem consists of a possibly infinite search space of configurations and a solution has to satisfy multiple, probably conflicting, objectives [11]. We therefore categorize our problem as a multi-objective combinatorial optimization (MOCO) problem, for which several methods have been discussed in the literature (cf. [5]). For our approach, we choose an evolutionary algorithm for the pattern selection problem, namely the nondominated sorting genetic algorithm II (*NGSA-II*) [4], guided by the estimated impact of a pattern on the NFPs.

Search Space. The search space consists of all possible patterns configurations as defined by the pattern templates and may be infinite, e.g., when considering floating point values. Therefore it is not possible to produce the complete search space in advance, but rather generate new random configurations based on the templates, if necessary.

Solution Space. A genetic algorithm maintains a set of solutions, called a *population*, and deploys selection, re-combination, and mutation operators to improve the quality of the solutions in the population in each iteration. In our approach, a (candidate) solution is a selected sequence of pattern configurations. To ensure the validity of candidate solutions, *solution constraints* requiring domain knowledge about the different patterns can be used to specify how configurations can be combined. As an example, it makes no sense to apply both, horizontal scaling and auto-scaling, on the same service, thus a constraint classifying such a solution as invalid may be specified. One drawback when using NSGA-II is that the length of the solution (n) must be fixed in advance, i.e., the number of pattern configurations appearing in a solution. To allow the calculation of solutions with less or equal than n pattern configurations, we introduce a pattern configuration *placeholder*, which may take one or more places in the solution, but has no influence on any of the NFPs.

Objective Space. To evaluate the quality (*fitness*) of a solution, the solution needs to be mapped to the objective space. In multi-objective optimization, this objective space consists of multiple dimensions, each dimension referring to one objective. Usually these objectives are competing, so that no single point in the objective space exists that dominates all other points, resulting in a set of optimal solutions. In our approach, the objective space is not pre-defined, but specified by the stakeholder implicitly by defining the goals. Each property that has a goal specified upon is one dimension in the objective space that needs to be evaluated. The evaluation of a solution candidate for each of these dimensions in the objective space is done by a so-called *fitness function*. This fitness function guides the algorithm into good areas of the solution space.

Fitness. We define the fitness of a solution in a specific dimension to be the sum of the weighted, relative distance between the property value resulting from applying the solution and the target value or target range set by the user for each goal of this property. The relative distance of a goal is the difference between the resulting property value and the user-defined target value or target range in relation to the target value or range. For target ranges, the mean of the range is taken as target value, however a fulfilled goal always results in a relative distance of zero. An additional penalty (weight) for each goal that has not been achieved is calculated by multiplying the relative distance with the proportional goal priority, resulting in a higher penalty for higher priority goals. The goal of the algorithm is to find a solution that minimizes the fitness values.

2.4 Model Simulation

Running NSGA-II gives us a set of solutions which form the Pareto front from the previously infinite solution space. These solutions can be evaluated in more detail using the more execution expensive, but also more precise, model simulation. For this, we build on our previous work [16] that is based on graph transformations supported by the *e-Motions* framework [15]. By using *e-Motions*, we run the modeled system and perform a more detailed evaluation also considering additional properties such as the contention of resources. The results from the model simulation are used to rank the solution set calculated by NSGA-II. The ranked solution set together with the approximate success of each solution to fulfill the goals is then presented to the stakeholders for the final decision about which configurations of the cloud computing patterns should be applied.

3 The Petstore Case Study

In this paper, we show the applicability and feasibility of our approach based on the Petstore case study. The case study is executed with the Java prototype we have developed using the NSGA-II implementation provided by the MOEA Framework[1]. The Petstore is a small web application allowing potential customers to create an account, log into this account, and order pets from a pre-defined pet catalogue. Previously the Petstore has been running on the local web server of the company, however now the company wants to move the Petstore application to the cloud to improve scalability and reduce cost. The Petstore architecture is realized with three entity classes and five services.

Entity Classes. The Petstore application maintains three entity classes, namely *Item*, *Customer*, and *Order*. All products in the Petstore are stored in the form of an item entity. Customers can create an account at the Petstore, log in, search for items, and place orders. An order consists of the registered contact information of a customer as well as the items and the quantity the customer has put into the shopping cart. Available service functionality is depicted in Figure 3, as explained later.

Services. Internally the Petstore uses different services to provide the necessary functionality to customers. The *Application Service* is the only service that a customer directly interacts with. It uses the *Customer Service*, *Catalog Service* and *Order Service* to handle the customer data, item data, and order data, respectively. All of these three services use the *Entity Service* to handle the persistence and the retrieval of data from a permanent data store.

3.1 Setup

Patterns. For this case study, we select the three patterns already introduced in Section 2.2: *Caching*, *Horizontal Scaling*, and *Auto-Scaling*. Considering the application conditions, caching can be applied on any of the three entity classes, while scaling can be applied on any of the five service classes. We assume that both scaling patterns improve performance (the more instances, the faster they process data) and worsen cost (each instance is billed by the cloud provider). Estimations about the gained speedup or utilization can be partially retrieved from a more detailed pattern description, but can also be gained from experience or dedicated cloud benchmarking services. Pricing information can be gathered from the website of the specific cloud provider. The resulting estimated impact for each pattern is summarized in Table 1.

[1] MOEA Framework, Version 2.1: `http://moeaframework.org/`

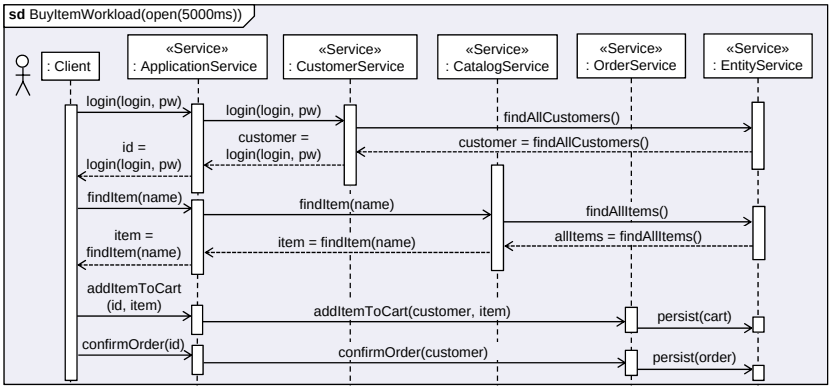

Fig. 3. The Petstore Scenario Workload

Furthermore, we define application constraints to guarantee that a pattern is not applied on the same entity or service multiple times and that the two scaling patterns are not applied on the same service at the same time.

Table 1. Estimated impact of considered patterns

Caching Impact				**Scaling Impact**	
Price per TimeUnit	SpeedUp Item	SpeedUp Customer	SpeedUp Order	Price per TimeUnit and service instances	SpeedUp n Instances
0.0015	5	3	1	0.0010	n

Goals. As mentioned in the previous section, all goals must be set in the context of a specific workload, as it is not feasible to show that certain goals hold in all possible use cases. For this case study, we consider a scenario where a single user connects to the Petstore application, logs into his or her account, searches for a specific item by name and then places an order on this item. Ten requests arrive in the application (modeling the connection of ten users) with an exponential distribution of five seconds (5000 time units). The scenario is summarized in Figure 3 with a sequence diagram. The main reason for moving the Petstore application to the cloud is to reduce cost and improve scalability, or more precisely, to reduce the overall *cost* and improve the *response time* of customer requests and the *utilization* of different services. Cost and response time are both properties which can have values in the range of $[0.0, \infty]$, with a lower value being considered better than a higher value. Utilization has a value range of $[0.0, 1.0]$ with neither lower values nor higher values being clearly better, making utilization suitable for a target range instead of a single target value. Too low utilization can suggest an idle resource, which produces cost and brings no benefit. Too high utilization can indicate an overloaded resource, resulting in a slower performance or a situation where consumers of the application are not served.

In this case study we assume that the following goals should be fulfilled within the context of the Petstore scenario. The application of a property is indicated by the property name and the applied element in parenthesis, an asterisk ($\star$) marks the whole application. The priority of a goal is given in square brackets after the condition.

Goal 1: `Cost(*) <= 900 [3]`
Goal 2: `ResponseTimePerRequest(*) <= 30000 [2]`
Goal 3: `0.15 <= Utilization(EntityService) <= 0.25 [1]`
Goal 4: `0.15 <= Utilization(CustomerService) <= 0.25 [3]`

NSGA-II Configuration. As mentioned in the previous section, genetic algorithms use selection, re-combination, and mutation operators to evolve the population into a good area of the solution and thus objective space. For selecting candidate solutions, we use a so-called *tournament selection strategy*, which takes n random candidate solutions from the population and allows the best one to be considered for re-combination (in our case, $n = 4$). Two candidate solutions are re-combined into two new candidate solutions by means of a single point crossover operator. This operator splits each solution at a random point into two parts and merges the first part of the first solution with the second part of the second solution and vice versa. After re-combination the validity of the resulting solutions is checked and mutation can take place. Invalid solutions are given the worst possible fitness and should eventually be removed from the population. Mutation occurs at a low rate (1.5%) in a solution and changes one of the pattern configurations concrete values slightly. In our case, this means that each parameter of a pattern configuration has a slight chance of being modified, e.g., the number of instances for horizontal scaling. Furthermore, we define a solution length of eight, as there are only eight classes on which at most one pattern can be applied. The algorithm should maintain 200 solutions per population and continue for at most 1000 iterations.

Fitness Function. To evaluate the quality of the solutions produced by the NSGA-II, we need to provide values for *response time*, *cost* and *utilization* by incorporating the impact estimations. As the fitness function is executed many times, we use a very simple model analysis technique, which may not be very precise, but is very fast to execute. First, we retrieve the configured number of instances for each of the services. Then we execute the scenario for all requests and services and calculate the runtime of each service by summing up the reduced execution times (original execution time divided by number of instances) of each operation call that has been made to this service during the execution. The sum of all operation executions is the total runtime of the application. Each request is seen as independent and no contention of resources is considered. Based on the runtime, we calculate both the utilization and the cost for each service using the provided pricing and speedup information. The resulting response time for each request is the total runtime divided by the number of requests.

3.2 Results

After running the NSGA-II algorithm, we are faced with 3 solutions, one of which is depicted in Figure 4. On this set of solutions, we run the model simulation as described in Section 2.4 to gain more detailed information about how close the solutions are to fulfilling the goals set by the user. For this, we need to define a metamodel and behavioral in-place rules that model the system at runtime. For each solution, an instance of this metamodel containing the applied patterns must be created and executed. The result of the model simulation is shown in Table 2. The first line presents the original configuration (no patterns applied), while the other three have some patterns applied. The left-hand side of the table shows the values for the NFPs of interest, while the middle part shows the distance to each goal, and the right-hand side displays the overall distance to the goals and the rank of the solutions.

Regarding the solutions, (1) and (2) use four patterns, while (3) uses three. Solution (1) auto-scales the *Entity Service* and the *Application Service* depending on the queue length. The first service ranges between 3 and 7 instances, while the second one does

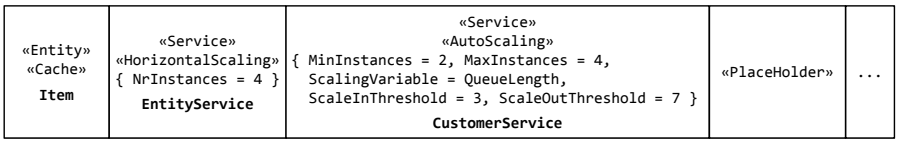

Fig. 4. Solution (3) with pattern configurations and placeholders

between 1 and 4. Solution (1) also has horizontal scaling for *Customer Service* and *Order Service*, with two instances for each one. Solution (2) auto-scales the *Order Service* depending on the queue length between 1 and 4 instances, and it also applies horizontal scaling in the *Entity Service* and *Customer Service*, with 4 and 3 instances, respectively. Caching on *Item* is applied as well. Finally, Solution (3), also depicted in Figure 4, applies caching on *Item* and horizontal scaling for *Entity Service* with 4 instances. It auto-scales the *Customer Service* between 2 and 4 instances depending on the queue length.

Table 2. The Petstore Scenario Workload Results

#	Cost	RespT	Util ES	Util CS	G1	G2	G3	G4	Sum	Rank
B	921	134500	0.95	0.217	0.024	5.225	7.5	0	12.749	-
(1)	935	29018	0.176	0.257	0.039	0	0	0.284	0.322	3
(2)	992	31698	0.215	0.168	0.102	0.085	0	0	0.187	1
(3)	948	32845	0.243	0.187	0.053	0.142	0	0	0.196	2

While we have a clear ranking according to the model simulation and the calculated distances, we still provide the user with all possible solutions and their detailed evaluation values to allow additional human reasoning. A user could still decide to apply solution (1) instead of the other solutions if she wanted the utilization of the *Entity Service* to be closer to the smallest target value or she could also decide to apply solution (3) instead of solution (2), because cost may still be the driving factor of the migration. Despite the ranking, we can note that none of the solutions is surprising and they probably could have been found by an expert using the estimated impact on the patterns and the knowledge about the system execution. However, we assume that with a more complex application and a higher number of goals and/or patterns, the manual derivation of solutions becomes harder.

4 Related Work

In software engineering, patterns are important ingredients to document knowledge on how to solve reocurring problems since the well-known book by the Gang of Four [9] describing patterns in the context of object-oriented design. With the appearance of the cloud computing paradigm, the community has already started working on cloud computing patterns [6, 13, 18]. For our approach, we studied different pattern descriptions, created pattern templates, and estimated the effect of each pattern on the different NFPs.

Optimization techniques are used to solve a variety of different problems [3]. Research in metaheuristics for combinatorial optimization problems aims to optimize the techniques applied in evolutionary algorithms [19]. At the same time, the focus of research has shifted from being rather algorithm-oriented to being more problem-oriented [2]. This is also reflected in the emerging search-based software engineering

paradigm [10, 12], which considers cloud computing as one of its application fields to tackle several multi-objective optimization problems [11]. Furthermore, the combination of model-driven engineering with search-based techniques is also investigated in several studies [14]. Following this path, we have applied a specific genetic algorithm to our optimization problem. To the best of our knowledge, there is only one prior work that applies optimization techniques to come up with an optimal configuration of a cloud application. In [8], the authors also use a combination of multi-objective search and simulation for finding an optimal deployment strategy for a given set of components of an application. In our approach, we go one step further and aim to optimize not only the deployment of the components, but also the usage of cloud computing patterns that are applicable on class-level granularity, what is of major interest when moving to PaaS providers.

An orthogonal optimization of cloud applications is targeted in the MODAClouds[2] and Passage[3] projects, where the multi-cloud deployment of applications is studied by the application of the models@runtime notion [7]. Our approach currently does not foresee any support for the multi-cloud deployments, but may be extended by additional patterns supporting such scenarios as well in the future.

5 Conclusions and Future Work

In this paper we have introduced a pattern-based optimization approach for cloud applications. We follow a model-based approach to select configurations of cloud optimization patterns that satisfy some restrictions in terms of non-functional properties, and we determine the best configuration using model simulation.

Currently, our approach faces some limitations, some of which we want to address in the future. First of all, we have assumed that the base architecture is suitable for the cloud. This might not be the case for all applications and additional architectural refactoring patterns may be applied before our approach. Also, the simulation of the results through *e-Motions* is not straightforward as we need to create a new meta-model for each system the approach is applied upon. Furthermore, *e-Motions* presents some scalability issues when the models to be simulated grow in size. Other simulation tools might not have these drawbacks and might be more easy to use. For now, we have not evaluated the scalability of our approach in detail. More use cases, also industrial-sized use cases, need to be evaluated to experiment with more complex patterns as well as a larger number of patterns, goals, and trade-offs involved. Regarding the input, we need initial estimates on the impact a pattern has on an application. It may prove difficult to get these estimates manually from experts. Automation support based on benchmarks, partial application execution or log analysis could be integrated to support the user in collecting the estimates.

In the paper we have presented a proof-of-concept of our approach, from which we will address several future lines of work next. Firstly, we will apply benchmarks to measure the improvement associated with optimization patterns in large-scale applications provided as use cases in the ARTIST project. Secondly, we also plan to consider more optimization patterns from our catalogue, as well as study their influence after the ap-

[2] MODAClouds: `http://www.modaclouds.eu/`

[3] Passage: `http://www.paasage.eu`

plication is deployed on the cloud. This would allow us to evaluate the feasibility and scalability of our approach in a more realistic setting. Thirdly, we plan to extend our goal modeling language to represent NFPs that are not taken into account in the current version, such as security properties. Finally, we plan to further study the application of different evolutionary algorithms for selecting the best configuration of optimization patterns.

References

1. Bergmayr, A., Brunelière, H., Canovas Izquierdo, J.L., Gorronogoitia, J., Kousiouris, G., Kyriazis, D., Langer, P., Menychtas, A., Orue-Echevarria, L., Pezuela, C., Wimmer, M.: Migrating Legacy Software to the Cloud with ARTIST. In: Proc. of CSMR. pp. 465–468 (2013)
2. Blum, C., Puchinger, J., Raidl, G.R., Roli, A.: Hybrid metaheuristics in combinatorial optimization: A survey. Applied Soft Computing 11(6), 4135–4151 (2011)
3. Coello, C.A.C.: A Comprehensive Survey of Evolutionary-Based Multiobjective Optimization Techniques. Knowl. Inf. Syst. 1(3), 129–156 (1999)
4. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. Trans. Evol. Comp 6(2), 182–197 (2002)
5. Ehrgott, M., Gandibleux, X.: A survey and annotated bibliography of multiobjective combinatorial optimization. OR-Spektrum 22(4), 425–460 (2000)
6. Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P.: Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications. Springer (2014)
7. Ferry, N., Rossini, A., Chauvel, F., Morin, B., Solberg, A.: Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-cloud Systems. In: Proc. of CLOUD. pp. 887–894 (2013)
8. Frey, S., Fittkau, F., Hasselbring, W.: Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In: Proc. of ICSE. pp. 512–521 (2013)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1 edn. (1994)
10. Harman, M.: The current state and future of search based software engineering. In: Proc. of ICSE. pp. 342–357 (2007)
11. Harman, M., Lakhotia, K., Singer, J., White, D.R., Yoo, S.: Cloud engineering is search based software engineering too. Journal of Systems and Software 86(9), 2225–2241 (2013)
12. Harman, M., Mansouri, S.A., Zhang, Y.: Search-based software engineering: Trends, techniques and applications. ACM Comput. Surv. 45(1), 11:1–11:61 (2012)
13. Homer, A., Sharp, J., Brader, L., Narumoto, M., T., S.: Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications. Microsoft Patterns & Practices (2014)
14. Kessentini, M., Langer, P., Wimmer, M.: Searching models, modeling search: On the synergies of SBSE and MDE. In: Proc. of CMSBSE@ICSE. pp. 51–54 (2013)
15. Rivera, J., Duran, F., Vallecillo, A.: A graphical approach for modeling time-dependent behavior of DSLs. In: Proc. of VL/HCC. pp. 51–55 (2009)
16. Troya, J., Vallecillo, A., Duran, F., Zschaler, S.: Model-driven performance analysis of rule-based domain specific visual models. Inf. and Soft. Technology 55(1), 88–110 (2013)
17. West, D.M.: Saving Money Through Cloud Computing. Brookings Institution (2010)
18. Wilder, B.: Cloud Architecture Patterns. O'Reilly (2012)
19. Zitzler, E., Deb, K., Thiele, L.: Comparison of Multiobjective Evolutionary Algorithms: Empirical Results. Evolutionary Computation 8, 173–195 (2000)