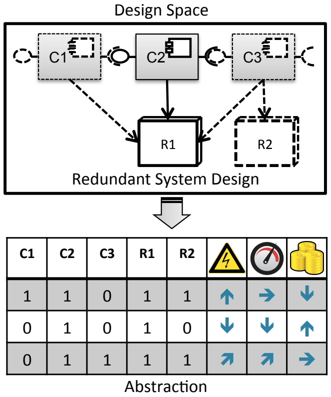
# Fault-aware Pareto Frontier Exploration for Dependable System Architectures

Lukas Märtin*, Hauke Baller*, Anne Koziolek†, Ralf Reussner†
*TU Braunschweig, Germany
Email: {maertin,baller}@ips.cs.tu-bs.de
†Karlsruhe Institute of Technology, Germany
Email: {koziolek,reussner}@kit.edu

***Abstract*—While designing dependable systems, a large number of asset combinations (system configurations) with contrary quality objectives needs to be investigated. Basically, each feasible configuration should be investigated. For fault-tolerant embedded systems this problem is extended by anticipating hardware faults leading to changed deployments of stressed resources in redundant constellations. The identification and evaluation of the best-fitting configuration remains a computationally intensive and difficult task at all.**

**We propose a multi-stage approach (1) to sample Pareto-optimal configurations for redundant system designs within hostile environments, (2) to check satisfiability of structural constraints and (3) to measure and identify quality degradation in fault scenarios. Thus, allowing developers to identify design flaws, leading to large quality degradations in case of emerging faults. We use genetic algorithms (NSGA-II) for sampling a wide range of system designs and demonstrate our approach by means of an exemplary fault-tolerant system.**



| C1 | C2 | C3 | R1 | R2 | ⚠ | ⏲ | 🪙 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 1 | ↑ | → | ↓ |
| 0 | 1 | 0 | 1 | 0 | ↓ | ↓ | ↑ |
| 0 | 1 | 1 | 1 | 1 | ↗ | ↗ | → |

Fig. 1. System Architecture for a Dependable System with Quality Ratings

## I. Introduction

In fault-tolerant software design, the provision of dependable systems is charged with high expenses. In particular, a diversity of replacement units (redundancies) needs to be specified and addressed in redundancy methods and distributed well to successfully maintain faults [1]. Thus, developers are concerned with distinguishing many feasible system configurations, mostly equipped with redundant hardware resources. To meet the functional requirements and simultaneously optimize multiple dimensions of system's quality, expensive explorations of the design space are inevitable to find a best-fitting system variant for deployment.

This challenge is getting even more complicated by further considering further variants for reconfiguration upon hardware resource faults. Such potential faults of stressed resources in hostile environments, e.g., cosmic radiation harming space crafts and satellites, might be predicted by methods like *Fault Tree Analysis*, but the consequences on quality and functional validity are still expensive to inspect appropriately for rich design spaces. Here, each feasible configuration should be evaluated in face of all alternatives, leading to a exponential complexity of comparisons. Even if the initial commit of a fault-tolerant system is usually more expensive than the initial commit of a regular system, the exploration of the design space has to be done in a systematic manner.

From an architecture-oriented point of view, the separation and modeling of software components and hardware resources lifts the exploration to an abstract level of component-based software engineering for embedded systems [2]. Here, the deployment of components describes use-relations to the platform (resources for execution). Figure 1 shows an example for a redundant system design with an excerpt of feasible configurations, defined as the *reconfiguration space* by the developer. Each configuration requires a different subset of hardware resources from the platform and result in varying ratings (values) for quality attributes. Figure 1 depicts the differing uses of hardware resources (R1, R2) by software components (C1, C2, C3) during execution. Configurations (rows) are defined by selecting (marked by 1) elements. Some selections are optional, denoted with dashed lines. Each defined configuration is rated, leading to rising, falling, or constant quality changes (arrow directions). Here, we consider the quality attributes *energy*, *performance* and *maintenance costs*. Each configuration is also validated against a set of structural constraints, representing the basic functional relations in the architecture model. As soon as one of the hardware resources is marked as faulty (defined as fault scenario), some configurations including fault-affected components may no longer be executable. To handle this partial loss of fault-tolerance, the developer needs to extend the former reconfiguration space by additional configurations, not relying on the faulty hardware resources. Each configuration has to be identified, quality-rated and compared to the existing (valid) configurations. This procedure supports the developer in identifying possible alternative configurations for an assumed fault scenario. In order to focus on significant degradations of quality attributes, a user-definable threshold for quality degradation is desirable from a developer's point of view.

In previous work [3], we arranged alternative configurations as nodes in a graph structure called *Architecture Relation Graph*. Edges result from the reduction of available hardware resources caused by faults. For edge prioritization, the qualities of each configuration are investigated. Such a strict hardware-oriented procedure is not feasible to evaluate alternative configurations efficiently while considering quality attributes. In this paper, we therefore investigate the measurement of quality distances between configurations, including validity checking according to required hardware resources.

### *Foundations and Related Work*

Our work relates to the concept of *Degrees of Freedom* [4] to define and evaluate variability in architecture design. Possible variation points are specified as explicit part of the architecture model. A genetic algorithm explores the design space to find *Pareto-optimal solutions*, i.e., the supremum of all feasible solutions with respect to contrary objectives, respecting a set of quality attributes. This procedure can be applied to support design decisions and to explore potential reconfiguration options [5]. To apply the approach to its full extend, we need to create rich design models to gather ratings for quality attributes by simulation. Instead in this paper, we use simplified representations of design models and abstract quality measurements in order to provide a lightweight implementation of the basic concepts in our approach.

The sampling of system configurations is performed by the genetic algorithm *NSGA-II* [6]. Echtle et al. [7] also apply such algorithms to identify fault-tolerant system designs on a high level of abstraction. This work is focused on finding critical fault combinations leading to invalid system designs. We describe the variation space of the system explicitly to check validity of many sampled designs in a short period of time. More precisely, we use a propositional logic formula to describe the design space of examined systems and imitate faults by disabling operation-critical hardware resources to restrict feasible variations in configurations. Technically, we represent relevant parts of the architecture model as features in a feature model. Feature models provide a comprehensible graphical representation of a variant rich system. Relations between functional components and hardware resources are defined by constraints in the feature model. The fitness of a feasible solution, i.e., a configuration validated by the feature model, is based on quality assignments annotated as property to each feature.

Frey et al. [8] inspect reconfigurations as deployment options for cloud-based systems derived by genetic algorithms. The authors predefine rules at design time for systematically modified deployments of a system upon changed circumstances in operation, e.g., system overloads. Similar to that, Jung et al. [9] adapt a running system by policies derived at design time. For that, a decision-tree learner is trained with feasible system configurations, generated from queuing models. Both approaches guide the developer to identify alternatives for reconfiguring a system. However, the reconfiguration space is not explicitly explored to identify quality drops upon faults in unstable hardware/software systems. Our approach identifies such gaps and prioritizes near-by alternative configuration for recommendation and decision support. In relation to our distance measurement in neighborhood of faulty configurations, Barnes et al. describe relations between architectures as candidate evolution paths [10]. These paths specify a search-based reconfiguration process from a source to a pre-defined target architecture via a sequence of transient architectures. The goal is to shorten the paths to minimize reconfiguration efforts. However, we aim to retain as much system quality as possible without defining a target architecture manually. J. R. Schott [11] defines a metric called spacing to measure how well non-dominated individuals on the Pareto-frontier are distributed with respect to their neighbors. Following that idea, Gong et. al [12] also use a neighbor-based technique to inspect the crowding distances of non-dominated individuals and select minority isolated individuals. Thus, they refine recombination and mutation by determining nearest neighbors of less-crowded individuals for the next optimization iteration. In our approach, we also explore dominated neighbors to find design alternatives for individuals that became infeasible due to resource faults by comparing and minimizing distances in the objective space.

In the area of search-based approaches, Garvin et al. [13] combine heuristic search with Feature Modeling. By using *simulated annealing*, the authors extend a test generation algorithm to determine valid feature configurations. Based on an array representation of a feature model, the algorithm perform pair-wise changes of feature selections. After each change a SAT check on the feature model is done. The fitness function of the optimization tries to maximize coverage of feature pairs. Similarly to that, Ensan et al. [14] apply a genetic algorithm to generate products (configurations) in accordance to a feature model. In both approaches, each gene of a chromosome represents a feature. The fitness of a product is coverage-oriented by evaluating the variability points and their constraints from the feature model. In our approach, a feature model provides the structure for variation points and restricts the selection of configurations by constraints. However, we do not consider coverage measurements, but guide our approach by minimizing distances between configurations. Furthermore, we assume, that the number of variation points is decreased by faulty features, potentially leading to faulty configurations. Several other tools from literature apply genetic algorithms to generate products from a feature model in testing Software Product Lines, e.g., PLEDGE [15].

## II. Multi-Stage Architecture Design Analysis

Our approach searches for appropriate architectural design alternatives for reconfiguration under the assumption of *predictable hardware resource faults*. The resulting set of configurations needs to be ordered and prioritized by multiple quality

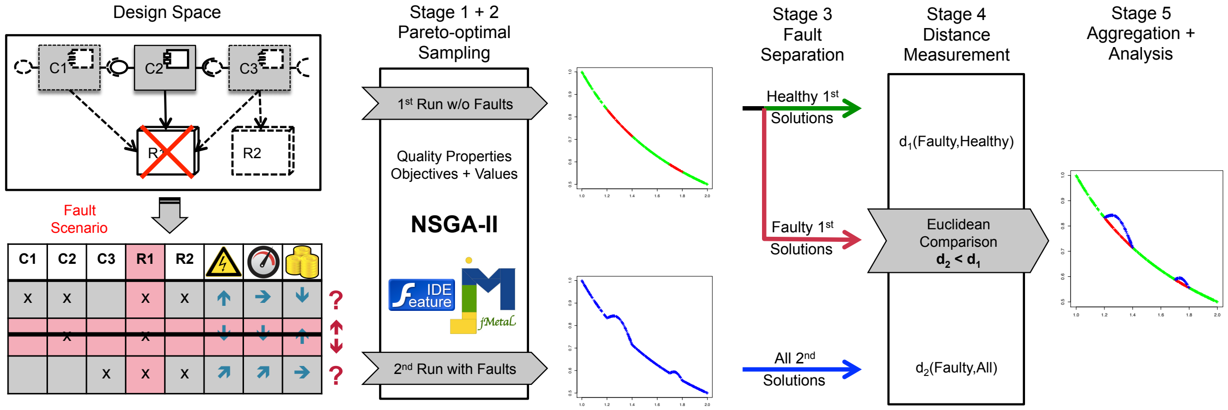


Fig. 2. Overview of our Multi-Stage Approach

dimensions. Thus, we explore the impact of resource faults with respect to the system's quality dimensions in multiple stages. For (re-)sampling, comparing and representing feasible configurations in a comprehensible manner, we provide tool support in five stages, depicted in Figure 2.

In the first two stages individual optimization runs are performed with different settings. In *stage one*, no faults are considered and each locally optimal and valid alternative configuration is added to the Pareto-frontier. Although fault-prone configurations of the first run might be detected for reevaluation easily, another run is needed to determine previously dominated non-faulty configurations. A second Pareto-frontier without any faulty configurations results from the second run performed in *stage two*. In *stage three*, the comparison of both Pareto-frontiers is prepared by injecting the same fault scenarios in the results of the first run. Therefore, all faulty configurations are separated from the healthy (still fully operational) ones. Faults in resources might lead to side effects in quality evaluation, e.g., if a faulty resource is cold-redundant to another still healthy resource of the configuration. Thus, an a posteriori re-calculation of qualities of each healthy configuration is performed. In a reconfiguration process, an alternative configuration for a faulty configuration seems to be optimal if minimal quality losses is archived. As measurement for comparing *neighbors* of faulty configurations, the Euclidean distances between the faulty configurations and healthy ones are determined in *stage four*. Next the distance measurement between faulty configuration of the first run and the newly sampled configurations from the second run is performed to find new nearest neighbors again. From both comparisons distance matrices result. To figure out the best-fitting alternative configurations, the matrices are merged to find in the combined results the nearest neighbors for each faulty configuration. This action already leads us to a basic transition structure to judge reconfiguration decisions. Our approach is intended to support design and maintenance activities. Therefore, the final *stage five* refers to the presentation of results. This allows developers of fault-tolerant systems to identify design flaws leading to potentially large quality degradations in case of faults. The presentation consists of statistics about the amount of nearest neighbors of faulty solutions and quality differences in distance matrices. Furthermore, large degradations are highlighted to identify needs for design improvements. Ultimately, the developer has just to set a threshold for distance values as the upper limit for acceptable degradations in each quality dimension. In the result all best-fitting alternatives for a configuration, addressed by a fault scenario, are presented including quantified quality differences.

### *Implementation*

Our approach was prototyped as an ECLIPSE plug-in[1]. Thereby, we combined the plug-in FEATUREIDE [16] for variant-rich feature modeling and validity analysis and the JMETAL [17] library for multi-objective optimization with meta-heuristics. Each problem-essential software component and hardware resource of the system is identified with a feature within a feature model. Furthermore, constraints in the model describe cross-cutting concerns between components and resources, i.e., implications or excludes. In order to improve readability and to represent an architecture-oriented structure, abstract features with no corresponding architectural elements are used. To define objectives for the optimization, the root of the feature model is annotated with a list of considered quality attributes. Based on this list, each concrete feature holds discrete ratings of one or more of these quality attributes. During optimization these assignments are evaluated and summed up[2] for each feature contributing to the configuration under the fitness analysis. We do not consider additional side constraints to restrict the optimization objectives beyond the ones from the feature model.

We apply the NSGA-II implementation from the JMETAL framework to sample binary decision vectors. For each 1 occurring in that vector, a corresponding feature in the feature model is selected; without any propagation guided by the rules in the feature model. The whole selection is validated by the SAT checker of the FEATUREIDE core engine. If the

[1] https://github.com/lmaertin/modcomp

[2] Function for aggregation can be customized, e.g., Mean or Median

sampled configuration is valid, the solution is rated by the quality assignments of the configuration, gathered from the feature annotations before. If the SAT check fails, the solution is downgraded in each quality dimension.

For the given fault scenarios, we mark faulty resources by deselecting features that are related with deployments to such resources. During the first optimization run, such faults are initially ignored. After the run is completed, faults are injected and all configurations from the first run are re-validated in FEATUREIDE. By rechecking satisfiability, some alternative might be no longer valid. The resulting faulty (non-valid) and healthy (still valid) configurations are stored in two independent sets for further processing. In addition, also the quality assignments of healthy configurations are reevaluated according to the potentially changed number of addressed quality attributes affecting the aggregated sums.

The presentation of results provides all data about feasible alternative configuration to the developer in a comprehensive manner. In addition to general statistics (number of solutions of both runs, ratios faulty vs. healthy and faulty vs. second run), the data of the new reconfiguration space is aggregated for decision support.

Because of usually varying qualities in multi-objective optimization, it is reasonable to let the user define a threshold for qualities for alternative configurations. In this way, the identification of a rich neighborhood set of configurations for each faulty configuration is promoted. Without a distance threshold, just the (one) best-fitting neighbor would be computed. After the data processing is completed, all distances are ordered beneath the threshold. On the one hand, the subset of results is shown as a distance matrix and new neighbors from the second run are highlighted. On the other hand, gaps between the Pareto-frontiers are investigated to identify most significant quality impacts. For that, a list of largest distances between faulty configurations and nearest neighbors is created.

The aggregated information can be used by the developer to optimize the design, e.g., by adding additional resources, and to derive rules for reconfigurations during self-maintenance.

## III. EVALUATION

For evaluation purposes, we applied our tool-supported approach to a *fault-tolerant vending machine*. For simplicity, the system deals with a well-known application scenario enhanced by redundancies and a fair reconfiguration space. Thus, the scenario addresses the domain of fault-tolerant embedded system design regarding redundant sensors and actuators. In addition, the system relies on software-intensive sensing and control, instead of pure mechanical solutions.

### *Fault-tolerant Vending Machine*

The vending machine offers *still water*, *sparkling water* and *coke* in cups, optionally chilled. Payments are accepted by *coins*, *notes* and *money card*. Some sensors and actuators can partially emulate other ones to support a high degree of fault-tolerance without cost-intensive replication of resources. For instance, water can alternatively be served by the *coke injector* after that injector was cleaned by an additional resource. Thus, each of these reproductions leads to changes in required resources and system's quality.

The system contains the following sensors and actuators.
**Sensors:** Buttons (still water, sparkling water, coke, return money), counters (coins, notes), a money card terminal and filling-level meters (water, coke-mix, collector tray for cleaning, cups), and a thermometer for chilling-control.
**Actuators:** Mixers (coke, $CO_2$), flow controllers (pump, gravity), valves (water, coke) and injectors (water, coke), money changers (coins, notes).

As a baseline for complexity analysis and satisfiability checks during configuration validations, we specified our design by a feature model, depicted in Fig. 3. The resources are represented as **concrete features** (dark blue boxes) in the model and labeled with indexes from 0 to 20. In total our system provides a design space of about 7,700 valid configurations with a variety of degradations in all quality dimensions. We assume that customers prefer to drink chilled drinks and like coke more than sparkling as well as sparkling water more than still water.

### *Quality Attributes and Fault Scenarios*

For optimization, we defined a set of quality attributes to be minimized. (1) *pollution* to observe the compliance of hygienic value limits, (2) *taste deviation* according to company's standards, (3) *response time* representing the time to drink delivery, and (4) *energy consumption* of the machine.

In order to evaluate our approach, we use a fault scenario with significant impact on the design space, i.e., affecting more than just optional features. By defining the resources *CokeInjector* and *Pump* as faulty, one half of initially sampled configurations is no longer valid. Thus, the developer has to figure out which alternative configurations are best-fitting.

### *Results and Findings*

The evaluation is performed on an Intel i5 CPU at 2,5GHz with 16GB of memory running Mac OS El Capitan, Java 8 and ECLIPSE NEON (FEATUREIDE 3.0.1, JMETAL 4.5.2). The NSGA-II was set to a *population size* of *100* and *250 iterations* for demonstration purposes.

After the first three stages are performed, the following statistics result from our experiment.

- #Solutions first run: 4
  - #Faulty: 2
  - #Healthy: 2
- #Solutions second run: 2

Thus, the comparison of distances between faulties and healthies as well as faulties and new sampled is done both times in ratios of 2:2. We pruned duplicate solutions, leading to a small set of unique optimal solutions for the example system.

With a threshold set to a maximal distance of 0.65, a distance matrix for each faulty solution is generated. Due to lack of space, in Table I the neighbors for only one faulty configuration are shown. The configurations are shown as binary vectors (optimization solution) representing the

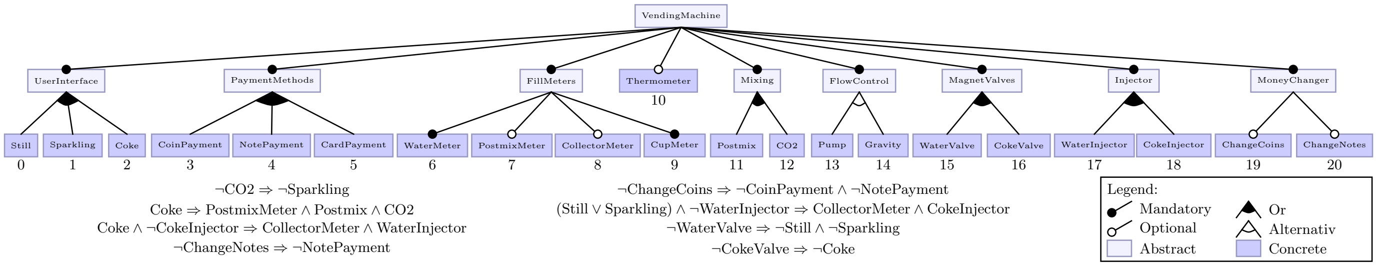


Fig. 3. Feature Model of Fault-Tolerant Vending Machine

TABLE I
DISTANCES FOR FAULTY SOLUTION (100100100100110101010)

| Dis. | Qual. Assignments | Solution Vector | Origin |
|---|---|---|---|
| 0.58 | 0.3 1.4 1.1 0.0 | 100001100100101101000 | healthy |
| **0.58** | **0.3 1.4 1.1 0.0** | **100001100100101101000** | **second** |
| 0.64 | 0.3 0.9 1.2 0.5 | 100100100100101101010 | healthy |
| **0.64** | **0.3 0.9 1.2 0.5** | **100100100100101101010** | **second** |
| ... | ... | ... | ... |

TABLE II
LARGEST GAPS IN VALUE ASSIGNMENTS TO OBJECTIVES

| Objective | Gap | Assignment 1 | Assignment 2 |
|---|---|---|---|
| 1 | 0.00 | 0.3 1.4 1.1 0.0 | 0.3 1.4 1.1 0.0 |
| 2 | 0.00 | 0.3 1.4 1.1 0.0 | 0.3 1.4 1.1 0.0 |
| 3 | 0.09 | 0.3 0.9 1.2 0.5 | 0.3 1.4 1.1 0.0 |
| 4 | 0.54 | 0.3 0.9 1.2 0.54 | 0.3 1.4 1.1 0.0 |

selection of concrete features in order of depth-first search corresponding to the indexes given in Fig. 3. Newly appearing nearest neighbors from the second run are highlighted in bold font. The quality assignments refer to the quality attributes and their order introduced before. Further neighbors with distances above the threshold value are hidden by "...".

By inspecting the values, the developer can figure out the configuration (100001100100101101000) as the nearest therefore and best-fitting neighbor of the faulty configuration. According to the feature model, the resulting vending machine sells still water via card payment, monitors water fill level and number of remaining cups, supports $CO_2$ mixing and uses a gravity-based water flow control towards a water value and a water injector. In this case, this configuration was randomly sampled in first and second optimization run. To assure fault-tolerance, the nearest neighbors shall also be considered in the rule set for run-time reconfiguration. Thus, all neighbors under the given distance threshold are added to a new set of solutions, representing the reconfiguration space. For optimization of the design space, the largest distances between solutions are also investigated. Our implementation performs pair-wise comparisons to find the largest distance between quality dimensions in objective space, i.e., gaps in Pareto-frontiers. In our experiments, we figured out the largest distances in solutions for all quality dimensions as listed in Table II. Subsequent to those results a developer may use an appropriate tool to visually explore gaps in the solution, e.g., by *hierarchical cluster analysis* in GNU R.

To visually present our idea, we performed an optimization with two quality dimensions (*Response Time* and *Energy Consumption*), resulting in the 2D-plot in Figure 4. Faulty solutions are colored in red, still healthy solutions are green and new solutions from second sampling are shown in blue. The plot shows a large gap between the Pareto-frontier of faulty solutions and the frontier containing all alternative solutions from the second run. Here, we suggest to recapture the design to minimize that gap by resource changes contributing to meeting the objectives.

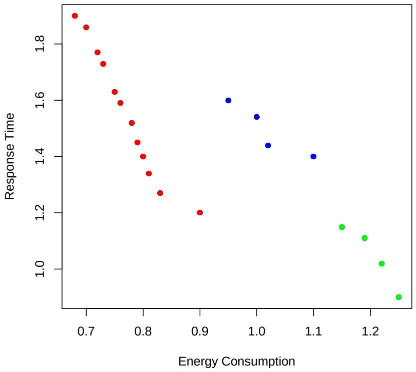


Fig. 4. Plot for two Quality Dimensions

### *Discussion*

During evaluation we were faced with some algorithmic characteristics in optimizing multiple objectives. Despite of differing assignments of quality attributes and a complexity of about 7.700 feasible configurations, the generation led to just a few unique configurations and many duplicates. We presume, that this is caused by the small-scaled application scenario and side-effects by similarities in quality assignments. Furthermore, we do not consult constraints for objectives, e.g., minimal acceptance values as lower bounds. Nevertheless, our gap investigations were also applicable by considering only widely spread objective values. Following the idea of Deb et al. [18], an $\epsilon$-dominance might support the reduction of such gaps by a better diversity to be maintained in a population.

## IV. Conclusions

Even if existing techniques for fault-tolerant system design assist the developer in identifying necessary redundancies, additional best-fitting configurations have to be figured-out. Our approach guides the developer through the subset of the remaining design options after a hardware resource fault is injected. Under the consideration of such a fault scenario, Pareto-optimal solutions are sampled and decision support to identify nearest alternates to a faulty configuration is provided in five process stages. In addition, large gaps in a system's quality can be shown with our tooling to find flaws in redundant system design.

### *Future Work*

We provide lightweight tooling for mass-generating solution set and gap exploration of Pareto-frontiers. Based on our previous experience, we plan to integrate our exploration concept within the Palladio toolset and its add-ons. For full-fledged architecture modeling [19] we will apply Palladio Bench[3] and PerOpteryx[4] to define variability in models. Also the sampling with genetic algorithms is performed there. Ratings of quality attributes are gathered by the simulation engine SimuCom. We will make use of the results for distance comparison and gap exploration proposed in this paper.

In particular, the findings of this paper will contribute to the improvement of our decision structure *Architecture Relation Graph* [3] and on-going work in area of *hierarchical cluster analysis*. The final selection of which configurations from the Pareto-frontiers are added to the graph still relies on trade-off settings preferred by the developer. Such trade-off analysis is supported by our tool AReva2[5]. Based on the work of Florentz et al. [20] contrary quality properties are normalized by conversion function and ordered hierarchically with weightings. This analysis needs to be integrated with the distance measurements from our tool prototype presented here.

We plan to comprehensively evaluate the integrated tool-supported approach at whole with a case study. For that, we will build upon our previous findings in the domain of space systems [21]. As a real-world case study, we have access to a system design of a micro-satellite provided by one of our industrial partners. The system has a high degree of inherent availability implemented by autonomy mechanisms and a large number of redundant hardware resources. We will extend our idea from previous work [22] in enhancing the system by replication-redundant capabilities as addressed here.

## Acknowledgment

This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future  Managed Software Evolution. The authors would like to give special thanks to Axel Busch for fruitful discussions on tool integration within PerOpteryx.

[3] http://www.palladio-simulator.com

[4] https://sdqweb.ipd.kit.edu/wiki/PerOpteryx

[5] https://github.com/lmaertin/areva2

## References

[1] A. Avizienis, "Toward systematic design of fault-tolerant systems," *Computer*, vol. 30, no. 4, pp. 51–58, 1997.

[2] L. Grunske, P. Lindsay, E. Bondarev, Y. Papadopoulos, and D. Parker, "An outline of an architecture-based method for optimizing dependability attributes of software-intensive systems," in *Architecting Dependable Systems IV*, ser. LNCS. Springer, 2007, vol. 4615, pp. 188–209.

[3] L. Märtin, A. Koziolek, and R. H. Reussner, "Quality-oriented Decision Support for maintaining Architectures of fault-tolerant Space Systems," in *9th Europ. Conf. on Software Architecture Workshops*, September 2015, pp. 49:1–49:5.

[4] A. Koziolek and R. Reussner, "Towards a generic quality optimisation framework for component-based system models," in *14th Int. ACM Sigsoft Symp. on Component based Software Engineering*, 2011, pp. 103–108.

[5] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya, "Software Architecture Optimization Methods: A Systematic Literature Review," *IEEE Trans. on Software Engineering*, vol. 39, no. 5, pp. 658–683, 2013.

[6] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[7] K. Echtle and I. Eusgeld, *A Genetic Algorithm for Fault-Tolerant System Design*. Springer, 2003, pp. 197–213.

[8] S. Frey, F. Fittkau, and W. Hasselbring, "Search-based genetic optimization for deployment and reconfiguration of software in the cloud," in *35th Intern. Conf. on Software Engineering*, 2013, pp. 512–521.

[9] G. Jung, K. Joshi, M. Hiltunen, R. Schlichting, and C. Pu, "Generating Adaptation Policies for Multi-tier Applications in Consolidated Server Environments," in *5th Int. Conf. on Autonomic Computing*, 2008, pp. 23–32.

[10] J. M. Barnes, A. Pandey, and D. Garlan, "Automated planning for software architecture evolution," in *28th Int. Conf. on Automated Software Eng.*, 2013, pp. 213–223.

[11] J. R. Schott, "Fault tolerant design using single and multicriteria genetic algorithm optimization." DTIC, Tech. Rep., 1995.

[12] M. Gong, L. Jiao, H. Du, and L. Bo, "Multiobjective immune algorithm with nondominated neighbor-based selection," *IEEE Trans. on Evolutionary Computation*, vol. 16, no. 2, pp. 225–255, 2008.

[13] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "Evaluating improvements to a meta-heuristic search for constrained interaction testing," *Empirical Software Engineering*, vol. 16, no. 1, pp. 61–102, 2010.

[14] F. Ensan, E. Bagheri, and D. Gašević, "Evolutionary search-based test generation for software product line feature models," in *24th Intern. Conf. on Advanced Information Systems Engineering*. Springer, 2012, pp. 613–628.

[15] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon, "PLEDGE: a product line editor and test generation tool," in *17th Intern. Software Product Line Conf. co-located Workshops*. ACM, 2013, pp. 126–129.

[16] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, "FeatureIDE: An extensible framework for feature-oriented software development," *Sci. Comput. Program.*, vol. 79, pp. 70–85, 2014.

[17] J. J. Durillo and A. J. Nebro, "jMetal: A Java Framework for Multi-objective Optimization," *Adv. Eng. Softw.*, vol. 42, no. 10, pp. 760–771, Oct. 2011.

[18] K. Deb, M. Mohan, and S. Mishra, "Towards a quick computation of well-spread pareto-optimal solutions," in *2nd Intern. Conf. on Evolutionary Multi-criterion Optimization*. Springer, 2003, pp. 222–236.

[19] R. Reussner, S. Becker, E. Burger, J. Happe, M. Hauck, A. Koziolek, H. Koziolek, K. Krogmann, and M. Kuperberg, "The Palladio Component Model," KIT Department of Informatics, Tech. Rep., 2011.

[20] B. Florentz and M. Huhn, "Embedded systems architecture: Evaluation and analysis," in *Quality of Software Architectures*, ser. LNCS. Springer, 2006, vol. 4214, pp. 145–162.

[21] L. Märtin, M. Schatalov, M. Hagner, O. Maibaum, and U. Goltz, "A Methodology for Model-based Development and Automated Verification of Software for Aerospace Systems," in *34th IEEE Intern. Aerospace Conf.*, 2013.

[22] L. Märtin and A. Nicolai, "Towards Self-Reconfiguration of Space Systems on Architectural Level based on Qualitative Ratings," in *35th IEEE Intern. Aerospace Conf.*, Big Sky, USA, March 2014.