Model-driven Continuous Experimentation on Component-based Software Architectures

1st Sebastian Gottschalk
Software Innovation Lab
Paderborn University
Paderborn, Germany
sebastian.gottschalk@uni-paderborn.de

2nd Enes Yigitbas
Software Innovation Lab
Paderborn University
Paderborn, Germany
enes.yigitbas@uni-paderborn.de

3rd Gregor Engels
Software Innovation Lab
Paderborn University
Paderborn, Germany
gregor.engels@uni-paderborn.de

***Abstract*—To build successful software products, developers continuously have to discover what features the users really need. This discovery can be achieved with continuous experimentation, testing different software variants with distinct user groups, and deploying the superior variant for all users. However, existing approaches do not focus on explicit modeling of variants and experiments, which offers advantages such as traceability of decisions and combinability of experiments. Therefore, our vision is the provision of model-driven continuous experimentation, which provides the developer with a framework for structuring the experimentation process. For that, we introduce the overall concept, apply it to the experimentation on component-based software architectures and point out future research questions. In particular, we show the applicability by combining feature models for modeling the software variants, users, and experiments (i.e., model-driven) with MAPE-K for the adaptation (i.e., continuous experimentation) and implementing the concept based on the component-based Angular framework.**

***Index Terms*—continuous experimentation, model-driven, component-based software architectures, self-adaptation**

I. Introduction

The development of new and improvement of existing software products is a cost- and resource-intensive task for a company [1]. In order to support that task and build product features that fulfill the user's needs, the developer has to continuously validate those needs and align the software product to it (see [2], [3]). This validation, in turn, can be performed by conducting controlled experiments where different software variants are displayed to distinct user groups, and the one with better quality metrics (e.g., registration rate) is deployed to all users after the experiment. This experimentation is a continuous process that constantly improves the product for the customer [4]. The performance of different variants can be measured in different ways. Here, split-testing (or A/B testing with only two variants) allows the analysis and comparison of a single variable over time, while multivariate testing allows the comparison of multiple variables simultaneously.

However, there is a lack of modeling languages "where engineers and scientists can formulate their problem with flexibility that goes beyond A/B testing" [5]. This lack of formal methods is also the result of a recent systematic literature review [6]. Using formal methods and especially modeling languages provides the advantage of a unification of the whole experimentation process [7]. Based on this unification, many new opportunities like the combination of multiple split-test to multivariate tests, the generation of code stubs for different software variants, or the tracing and reasoning of decision-making are possible. Moreover, these models can be linked to software architectures to support an efficient and effective continuous experimentation process. Here, major challenges to support the efficiency are the combination of different experiments and the generation of the components, while major challenges of the effectiveness are modeling the right granularity of experiments and evaluating against the appropriate criteria.

To support the formalization process, this paper presents our vision of model-driven continuous experimentation. For this, Sect. 2 shows related approaches that intersect with our vision. Sect. 3 presents an overview of our vision and applies it to the model-driven continuous experimentation of software architectures. Sect. 4 discusses the opportunities and risks of our vision together with providing future research questions. Finally, Sect. 5 summarizes our vision and the future work.

II. Related Work

The related work can be divided into *Conceptual Models for Experimentation*, *Adaptive Systems for Variant Management*, and *Modeling Languages for Experimentation*.

The *Conceptual Models for Experimentation* provide high-level abstractions of how the experimentation process can be integrated into the product development. Hypothesis Experiment Data-Driven Development (HYPEX) [2] develops an iterative approach to evaluate new features by analyzing the gap between the expected and actual behavior. Those features are chosen from a continuously updated product backlog. Rapid Iterative value creation Gained through High-frequency Testing (RIGHT) [3] is similar to HYPEX but has a stronger focus on the involved stakeholders during the process and the deployment of the software product. Qualitative/quantitative Customer-driven Development (QCD) [8], in contrast to HYPEX and RIGHT, combines qualitative customer feedback

This work was partially supported by the German Research Foundation within the CRC "On-The-Fly Computing" (Project Number: 160364472SFB901) and the German Federal Ministry of Education and Research through Software Campus grant "KOVAS" (Project Number: 01IS17046).

with quantitative customer observations to gain information that is not quantifiable. However, none of those approaches makes use of models to support the experimentation process.

The *Adaptive Systems for Variant Management* provide an adaptation logic for a software product based on a measured context. Data Acquisition and Control Service (DCAS) [9] applies the concept of architecture-based self-adaptation to the case of managing highly populated device networks. In [10], we combine the existing Interactive Flow Modeling Language with new languages for the context modeling and adaptation logic to provide an adaption of user interfaces. Automated Online Experiment-driven Adaptation (AOEDA) [11] predicts the outcome of a software product based on the conduction of online experiments using an adaptation logic. However, none of those approaches focuses on the continuous experimentation process in combination with variants of the software product.

The *Languages for Experimentation* provide different modeling languages to support the general experimentation process. Camara and Kobsa [12] present an approach for creating different variants of the software product based on aspect-oriented software development and software product lines. In [13], we provide a language to model a linkage of hypotheses and experiments together with a process to validate those hypotheses using the experiments efficiently. Auer and Felderer [14] present a taxonomy of experiment characteristics together with a concept of a platform-independent online controlled experimentation. However, none of those approaches provides a modeling language that is fully aligned with the variants of the software product.

III. Vision on Model-driven Experimentation

This section shows our overall vision of model-driven continuous experimentation and its application to the solution concept of feature models and the Angular framework.

A. Overall Vision Idea based on Adaptive Systems

Our overall vision is to increase the efficiency and effectiveness of continuous experimentation based on a model-driven approach. For that, we need to focus on the development of the models (i.e., *Model Development*), the linkage to the actual code (i.e., *Code Linkage*), and the continuous experimentation process (i.e., *Experimentation Process*).

The *Model Development* is needed to provide an adequate formalization of the involved environment based on modeling languages (ML). For that, we focus on the modeling of the different variants of the software product (i.e., *Variant Model*), the experiments to conduct (i.e., *Experiment Model*), and the involved users (i.e., *User Model*). Here, the *Variant Model* (based on *VariantML*) needs the ability to model the software product itself together with explicit variants and their relationships to each other. Moreover, there need to be defined measurements in the form of metrics to plan adjustments according to the *Experiment Model*. The *Experiment Model* needs the ability to model different experiments with a time range, a priority, and a link to the measurements for testing. For this testing, different variants need to be selected from the *Variant Model* and user groups from the *User Model*. The *User Model* needs the ability to model the unique identification of a user together with his characteristics.

The *Code Linkage* is needed to provide a bridge between the model and the actual source code of the software product. Here, the linkage can be divided into the usage of models as secondary (i.e., model-based) or primary (i.e., model-driven) artifacts. Based on that, the term model-based continuous experimentation describes approaches that use models, for example, as documentation for the software developers and their evolution as traceability mechanisms. In contrast, model-driven continuous experimentation describes approaches that provide a direct relationship between models and source code, for example, through code generation. In this paper, we focus on the model-driven connection through code generation. An overview of the code generation, based on model-driven UI adaptation [10], can be seen in Fig. 1. It consists of the three stages of *Modeling*, *Transformation*, and *Execution* based on the *Product Variant*, the *Experiment*, and the *User*. In the *Modeling*, the developed modeling languages (i.e., *VariantML*, *ExperimentML*, *UserML*) are used to model the current variants of the product (i.e., *Variant Model*), the experiments to conduct (i.e., *Experiment Model*), and the potential users (i.e., *User Model*). Here, the *Experiment Model* needs references to the different variants in the *Variant Model* and characteristics of users in the *User Model*. Based on that, the *Transformation* uses generators (i.e., *Variant Stub Generator*, *Experiment Service Generator*, *User Characteristics Stub Generator*) to create platform-specific code for the software architecture where it should be used. While the *Experimentation Model* is translated into a complete service, the other models are transformed through code stubs. Before the *Execution* of the code, the stubs are consisting of frames of the underlying architecture that need to be filled with specific source code. Here, each of the *Variant Stubs* needs to be filled with source code about the specific variant (e.g., list, view) that display the user interface and the *User Characteristics Stub* needs to be filled with code about the user characteristics (e.g., location, gender) that can be gathered for example from the user profile.

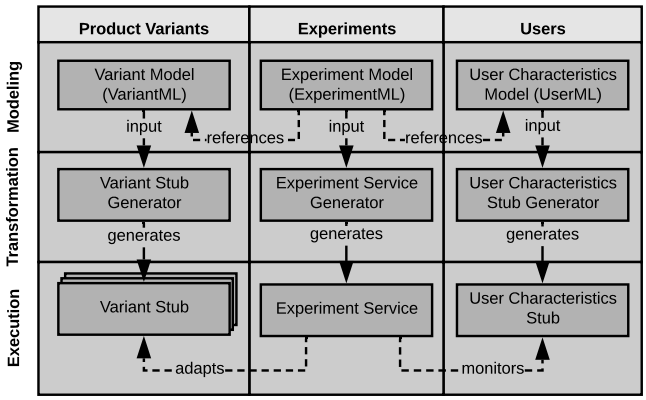

Fig. 1. *Code Linkage* of Model-driven Continuous Experimentation

The *Experimentation Process* is needed to test the different variants within the software product continuously. For that, an overview of our vision based on MAPE-K [15] is depicted in Fig. 2. Here, we have the *Software Product* (i.e., Managed Element) together with the *Experimentation Manager* (i.e., Autonomic Manager). The *Software Product* displays the product variant and can be changed with the *Experimentation Manager* by conducting different experiments over time. The *Software Product* provides sensors to measure the performance of the product variant and effectors to change the current product variant. Here, the *Experiment Data Service* provides access to the *Variant Measurement Data*, which stores the key measured metrics (e.g., number of registrations, number of clicks) of a specific variant, and the *User Characteristics Data*, which stores the important information about the user (e.g., unique identifier, location). Those models are filled up by the actual *Variant Stubs* and the existing *User Characteristics Stub* which are generated and coded within the last step. The *Experimentation Manager* provides the adaptation logic to change the product variant based on the measured data and the defined experiments. For that, the *Knowledge* base consists of all information about the user (i.e., *User Model*), the possible variants (i.e., *Variant Model*), and the experiments to be performed (i.e., *Experiment Model*) that are defined in the last step. The *Monitor* measures the information of the user characteristics and his usage of the variant (i.e., *Experiment Monitoring*). After that, the *Evaluate Experiment* analyzes that measured data and, based on the results (i.e., *Experiment Analysis*), recommends a change of the experiments conducted by the user (i.e., *Experiment Evaluation*). Those experiments are translated to a concrete product variant (i.e., *Experiment Adaptation*) and used to adapt the product during *Execute*. For that, the previously generated *Experiment Service* is triggered. During the whole adaptation, it is important to continuously show the different variants of an experiment to the same distinct user group and avoid the simultaneous execution of experiments that modify identical variants in different ways.

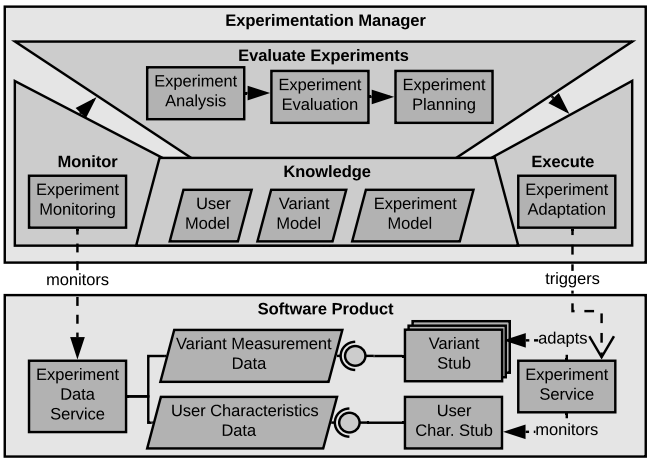

Fig. 2. *Experimentation Process* of Model-driven Continuous Experimentation

B. Applied Vision to Software Architectures

In order to show the applicability of our vision, we provide a preliminary solution on the model-driven continuous experimentation of component-based software architectures. For that, we combine the conceptual modeling of feature models [16] with the architectural modeling of the Angular framework[1]. Moreover, we use the Loopback API handling framework[2] and the MongoDB database[3] for the prototypical implementation. As shown in Fig. 3, we consider our three steps of *Model Development*, *Code Linkage*, and the *Experimentation Process*. We have applied the whole solution to the development of a streaming app.

Inside the *Model Development*, we need to define the *Variant Model*, the *Experiment Model*, and the *User Model*. The *Variant Model* is the heart of our solution, which represents all different variants the software architecture can be changed to. For that, we extend the concept of feature models, which is a compact representation of all features within a Software Product Line [16], to the representation of different software variants. Moreover, we provide a visual notation of those extended feature models applied on a streaming app in Fig. 4. Inside our model, we have stand-alone *Features* (e.g., *List*) together with their different *Variants* (e.g., *ListView*, *GridView*). Like in feature models, those *Features* are structured within a tree, can be *Mandatory* (e.g., *View*) or *Optional* (e.g., *Register*), and *OR* and *XOR* relationships can exist within the tree. Moreover, there can be *Requiring* (e.g., *Comments* requires *Register*) and *Excluding* relationships. In order to support software architectures, we add the *Stereotype* with values for *containers* (just for structuring and not used in the architecture), *components* (represents a single component within the architecture), and *elements* (represent a single feature within a component of the architecture). Moreover, each *Feature* can have different parameters (e.g., *SubscriptionFee*) and measurements (e.g., *ClickRate*). Those parameters and measurements are also used within the experiments. The *Experiment Model* represents all adjustments of the software architecture that are displayed to the users. For this, each experiment consists of a start and an end date together with a priority of the corresponding experiment. Moreover, each experiment has an evaluation type to aggregate a single user's measurements connected to the measurements as overall evaluation criteria. This could be the minimum or maximum value, a median or average building, or the summing up. For the evaluation, two or more test variants are defined. Here, each test variant contains a number of influencing features and possible parameter changes of those features. Last, there can be different user characteristics defined for each experiment. The *User Model* shows the characteristics of the user that is accessing the architecture. Here, we provide a unique identification of the users together with the different characteristics each user can have for assigning the experiments.

[1]Angular Website: https://angular.io/
[2]Loopback Website: https://loopback.io/
[3]MongoDB Website: https://www.mongodb.com/

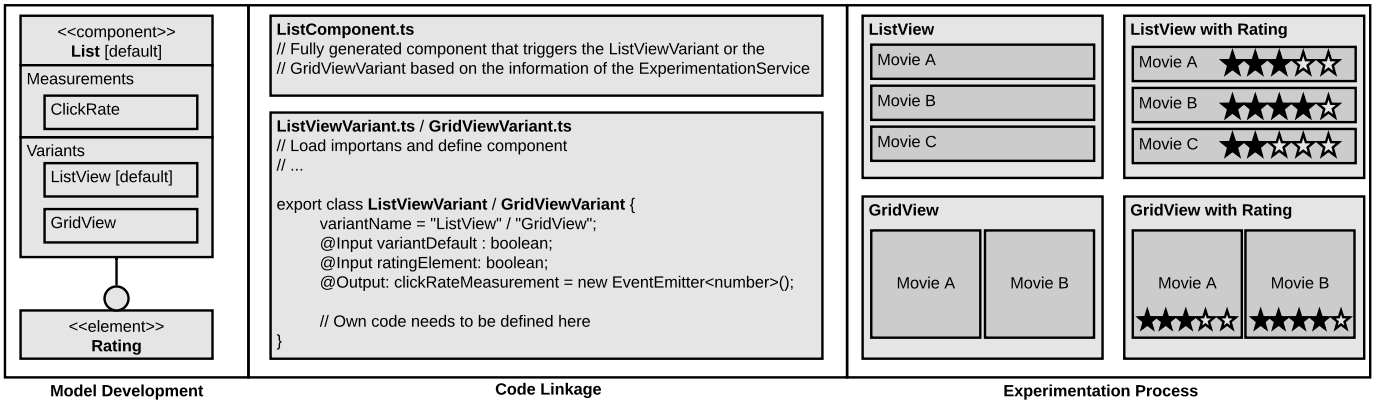

Fig. 3. Preliminary solution of the three steps of *Model Development*, *Code Linkage*, and *Experimentation Process*

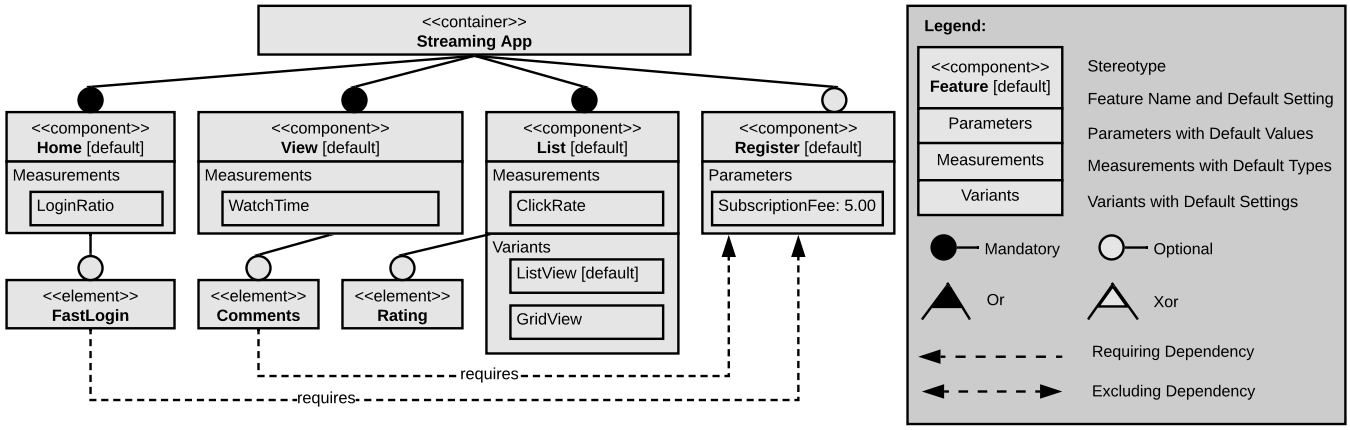

Fig. 4. Visual notation of the *Variant Model* based on the example of a *Streaming App*

Inside the *Code Linkage*, we need to generate the source code that will be used during the execution. For that, we use a web-based online editor based on Loopback to store the information of the *Variant Model*, the *Experiment Model*, and the *User Model*. This information, in turn, is transformed into code using custom schematics of Angular[4]. Here, we implemented a small code generator that crawls all information over the Loopback API and translates it into source code. The *Experiment Service* and the *Experiment Data Service* are standardized services that are connected through the *Experimentation Server* using the APIs of Loopback. Moreover, the *User Characteristics Stub* is a model with a predefined identifier for the user and an array of user characteristics that can be changed during the execution of the *Software Product*. The *Software Product* is generated in the following way: If a component-feature has no variants, it is directly translated into a stub where custom code can be inserted. If it has different variants, it is fully generated into a routing component that chooses the current variant with support of the *Experimentation Service* and the *Experimentation Data Service*, and passes parameters and measures of the variant to the *Experimentation Service*. Moreover, as shown in Fig. 3, we created a variable of the *variantName* together with using the standardized *@Input* and *@Output* concepts in Angular of communicating the parameters, measurements, and default variant to the parent component. Here, we defined also *@Inputs* for each element-feature that can be activated or deactivated within the experimentation. Before the execution, all stubs have to be filled up with customized code.

Inside the *Experimentation Process*, we need to provide testing the variants with the actual users. Here, the user initially accesses the *Software Product* and sends his user identification to the *Experimentation Server*. The experimentation looks up in the knowledge if there is already existing information about a variant for the user. If yes, the server controls that variant for finished experiments and adjustments of new experiments and sends the information to the product. If no, the server calculates a new variant based on the prioritization of the experiments, the combinability of experiments, and the current distributions among existing users and sends the information to the products. The product receives that information and arranges the variant of the software architecture according to that. While the user is accessing the variant, the product tracks the measurements and sends them to the server.

[4]Angular Schematics: https://angular.io/guide/schematics

IV. Discussion & Future Research Questions

In this paper, we have presented our vision of model-driven continuous experimentation for component-based software architectures. Here, our vision provides opportunities and risks we further want to discuss. One opportunity of the vision is the stronger consideration of the experimentation by explicitly formalizing it through models. This is connected to the point that by using code generation, the software developer can reduce the overhead of such experimentation. Last, the process efficiency can be increased by combining different experiments so that multiple split testings can be accumulated to multivariate testings. One risk of the vision is the overengineering of the experimentation based on fine-granular models, leading to longer development cycles. These fine-granular models can also lead to a high amount of code during the generation that needs to be initially developed, maintained, and removed. Last, the process effectiveness can be decreased through the wrong combination of different experiments with similar influencing variables.

Out of our vision, we derive future research questions around the three steps of *Model Development*, *Code Linkage*, and the *Experimentation Process*.

Research on *Model Development* needs to be done to find adequate formalizations for the variants, experiments, and users. Therefore, the following RQs needs to be considered:

- What information granularity is needed to model all variants of a software product?
- What characteristics of users are important to distinguish different user groups?
- What information space, including metrics, is necessary to describe the experimental setting?

Research on *Code Linkage* needs to be done to provide an effective connection between the models and the source code. Therefore, the following RQs needs to be considered:

- How to manage the relationships between the models and the source code?
- How to generate code stubs so that new variants of software products can be fastly created?
- How to automatically remove the code of old variants to improve the maintainability of the code?

Research on *Experimentation Process* needs to be done to provide an efficient combination of the modeled and executed experiments. Therefore, the following RQs needs to be considered:

- How to optimize the combination of different experiments to allow multi-variate testing?
- How to define reasonable stop criteria to evaluate different, partly combined, measurements?
- How to combine qualitative feedback with our quantitative approach?

V. Conclusion & Future Work

The development of successful software products is a challenging task that can be supported by continuous experimentation. However, existing approaches do not focus on the usage of unified modeling languages. For that, we present our vision on model-driven continuous experimentation together with its application on component-based software architectures. We discussed the opportunities and risks of such an approach and pointed out future research questions around the steps of model development, code linkage, and the experimentation process. In the future, we want to work on our vision by providing an enhanced concept for model-driven continuous experimentation together with a fully-fletched toolchain for component-based software architectures. Moreover, we want to explore the usage of our approach on other software architectures like microservices.

References

[1] E. Ries, *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. USA: Crown Business, 2014.

[2] H. H. Olsson and J. Bosch, "The HYPEX Model: From Opinions to Data-Driven Software Development," in *Continuous Software Engineering*. Springer, 2014, vol. 14, pp. 155–164.

[3] F. Fagerholm, A. Sanchez Guinea, H. Mäenpää, and J. Münch, "The RIGHT model for Continuous Experimentation," *J. Syst. Softw.*, vol. 123, pp. 292–305, 2017.

[4] A. Fabijan, P. Dmitriev, H. H. Olsson, and J. Bosch, "The Evolution of Continuous Experimentation in Software Product Development: From Data to a Data-Driven Organization at Scale," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 770–780.

[5] I. Gerostathopoulos, H. H. Olsson, T. Brand, R. Chatley, N. Diamantopoulos, A. Friedman, M. Jiménez, J. O. Johanssen, P. Manggala, M. Koseki, J. Melegati, M. Konersmann, N. Munaiah, G. Tamura, V. Theodorou, J. Wong, I. Figalist, S. Krusche, D. I. Mattos, J. Bosch, T. Bures, B. Fitzgerald, M. Goedicke, and H. Muccini, "Continuous Data-driven Software Engineering - Towards a Research Agenda," *ACM SIGSOFT Software Engineering Notes*, vol. 44, no. 3, pp. 60–64, 2019.

[6] F. Auer, R. Ros, L. Kaltenbrunner, P. Runeson, and M. Felderer, "Controlled experimentation in continuous experimentation: Knowledge and challenges," *Information and Software Technology*, vol. 134, p. 106551, 2021.

[7] J. Bézivin, "On the unification power of models," *Software & Systems Modeling*, vol. 4, no. 2, pp. 171–188, 2005.

[8] H. H. Olsson and J. Bosch, "Towards Continuous Customer Validation: A Conceptual Model for Combining Qualitative Customer Feedback with Quantitative Customer Observation," in *Software Business*. Springer, 2015, vol. 210, pp. 154–166.

[9] J. Cámara, P. Correia, R. de Lemos, D. Garlan, P. Gomes, B. Schmerl, and R. Ventura, "Incorporating architecture-based self-adaptation into an adaptive industrial software system," *Journal of Systems and Software*, vol. 122, pp. 507–523, 2016.

[10] E. Yigitbas, I. Jovanovikj, K. Biermeier, S. Sauer, and G. Engels, "Integrated model-driven development of self-adaptive user interfaces," *Software & Systems Modeling*, vol. 19, no. 5, pp. 1057–1081, 2020.

[11] I. Gerostathopoulos, F. Plasil, C. Prehofer, J. Thomas, and B. Bischl, "Automated Online Experiment-Driven Adaptation–Mechanics and Cost Aspects," *IEEE Access*, vol. 9, pp. 58 079–58 087, 2021.

[12] J. Cámara and A. Kobsa, "Facilitating Controlled Tests of Website Design Changes: A Systematic Approach," in *Web Engineering*. Heidelberg: Springer, 2009, vol. 5648, pp. 370–378.

[13] S. Gottschalk, E. Yigitbas, and G. Engels, "Model-Based Hypothesis Engineering for Supporting Adaptation to Uncertain Customer Needs," in *Business Modeling and Software Design*. Cham: Springer, 2020, vol. 391, pp. 276–286.

[14] F. Auer and M. Felderer, "An Approach for Platform-Independent Online Controlled Experimentation," in *Software Quality: Future Perspectives on Software Engineering Quality*. Cham: Springer, 2021, vol. 404, pp. 139–158.

[15] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[16] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines*. Heidelberg: Springer, 2013.