Towards Systematic Model-based Testing of Self-adaptive Software

Georg Püschel, Sebastian Götz, Claas Wilke, Uwe Aßmann
Software Technology Group, Technische Universität Dresden
Email: {georg.pueschel, sebastian.goetz, claas.wilke, uwe.assmann}@tu-dresden.de

***Abstract*—Self-adaptive software reconfigures automatically at run-time to environment changes in order to fulfill its specified goals. Thereby, the system runs in a control loop which includes monitoring, analysis, adaptation planning, and execution. To assure functional correctness and non-functional adequacy, testing is required. When defining test cases, the control loop's tasks have to be validated as well as the adapted system behavior that spans a much more complex decision space than in static software. To reduce the complexity for testers, models can be employed and later be used to generate test cases automatically—an approach called Model-based Testing. Thereby, a test modeler has to specify test models expressing the system's externally perceivable behavior. In this paper, we perform a Failure Mode and Effects Analysis on a generic perspective on self-adaptive software to figure out the additional requirements to be coped with in test modeling.**

Keywords*—*self-adaptive software; problem statement; model-based testing; failure mode and effects analysis

I. INTRODUCTION

Self-adaptive software (SAS) reconfigures automatically at run-time according to sensed environment changes. Thus, it is able to effectively and efficiently fulfill its specified goals in the changed conditions. SAS runs in a control loop that frequently operates four tasks: monitoring, analysis, planning, and execution (MAPE, [1]).

Quality assurance for SAS includes validation or verification of the control loop's tasks. While, for instance, run-time self-testing of SAS was researched (e.g., in [2]), there still is a lack of black box testing approaches. However, such concepts are or will be required by the software industry, e.g., due to the need for certification. Systems are delivered and left to the customer and have to be reliable and evaluated in advance as far as possible under the respective project conditions. Hence, not only fault-tolerance but also fault-prevention is desirable.

The crucial challenge of SAS black box testing [3, p. 17] is to handle its complexity. The behavioral decision space is extensively expanded due to the impact of adaptation on adapted system structures and interacting running processes. A further problem is *error propagation*: errors produced in one component can be transformed into errors which manifest in other components [4] and—due to the looped control flow—as a permanent inner system state. Thus, manifold information have to be considered by testers, injected into the tested system, monitored, and evaluated for determination of behavioral correctness. In consequence, *manual* test case definition for an SAS is mostly hard to manage for test engineers.

In constructive phases of SAS engineering, this complexity is dealt with by using models. Modeling takes advantage of abstraction and convention (i.e., implicit information) to hide details from designers. For testing, an equivalent concept was developed: in *model-based testing* (MBT, [5]) the system under test's (SUT) interfaces are structually and behaviorally modeled and later test cases are automatically generated. For limitation and measurement of actually tested parts of the behavioral space, a test coverage criterion can be specified. Due to the focus on the SUT's interface, MBT is a *black box* approach.

As we previously constructed test models for run-time variable mobile systems [6], we are now focusing on generalization of our experience and provide reasonable test models for SAS. A prerequisite for the application of MBT methods is to gather the following information:

1) Scenarios, how failures can occur, have to be found such that the progress of system validation be can estimated, effort predicted, and testers are aware of the potential complexity of their task.
2) Properties that failures can comprise have to be identified to provide meaningful verdicts.
3) Potential error propagation has to be investigated to identify causal chains.

While the above points fit on arbitrary system types, it becomes necessary to find more specific test modeling requirements for SAS. To bypass the intuitive formulation of these requirements we decided to perform a systematic analysis of critical failure properties and scenarios based on *Failure Mode and Effects Analysis* (FMEA, [7]), which was developed to investigate potential failures in systems. The results of its analytic methods provide a solid foundation for our formulation of MBT requirements.

The contribution of this paper is twofold:

1) *Failure analysis*: We analyze which properties failures in adaptive systems can encompass and which scenarios may occur by performing a FMEA-based investigation process.
2) *Requirements identification*: We derive requirements for test models based on the discovered failure scenarios. The result is a reasonable foundation for test research concerning adaptive systems.

The remainder of this paper is structured as follows: We start with related work in Section II. In Section III, we perform our FMEA-based investigation for SAS and in Section IV, we state the resulting modeling requirements. We finish in Section V and outline future work.

II. RELATED WORK

In this section, we present related approaches concerning testing adaptive systems from the perspectives of different research directions. On the one hand, *Self-adaptive Software* (SAS) [3] and *Models@run-time* (MRT) communities have to be considered as sources of adaptation concepts; on the other hand *Dynamic Software Product Lines* (DSPLs) [8] are an intersecting approach based on means like variability

 ISBN: 978-1-61208-274-5

management and features. Furthermore, *context-adaptivity* has been a long-term research field. In all these directions, several testing methods were established. In the following, we concentrate on approaches with models which can help to build up a reasonable information base for testing and on those which propose fitting coverage concepts.

A conceptional discussion on challenges concerning verification and validation of SAS was done in [3]. The authors proposed to focus on adaptive requirement engineering and runtime validation to assure adaptive software's quality. However, they also constructed an abstract model of adaptive software's states consisting of an inner system state plus a mode or phase. The latter ones describe in which variation/adaptation a system works. Each transition concerning either mode or state changes the overall configuration of the system and hast to maintain certain local or global properties. It is also discussed that a steady model as behavioral specification is insufficient such that the configurations have to conform to a dynamic selection of associated models. While these proposals are general enough to abstract from specific self-adaptive systems, several problems remain. Due to the enormous complexity in the behavioral space of adaptive software, an exact limitation of possible transitions to such which are correct and relevant for testing is a very hard task. In consequence, a much more expressive and usable model should be applied in test modeling.

Another approach by Munoz and Baudry is presented in [9]. The authors formalize context and variant models and generate sequences of context instances by using Artificial Shake Table Testing (ASTT). Thus, an adequate set of dynamic environment changes can be simulated. The case study's adaptation is designed by so-called policies, which also serve as an oracle (i.e., a mapping between inputs and outputs) in testing. Hence, with this approach testers are able to validate the correctness of the adaptation decision and to produce a sequence of re-configurations. While being a reasonable and basic approach to testing adaptive software, it lacks the means to deal with the discussed interaction challenge due to its exclusive consideration of environment changes.

An advanced research framework for adaptive software is provided by the DiVA project. DiVA had impact on both SAS and DSPL research. It also includes a methodology for testing [10][11]. DiVA's validation process is split into two phases: (1) The early validation is based on design time models (adaptation logic and context model) and executed as a simulation. A main focus in DiVA's test method is to generate reasonable context instances and associate "partial" solutions (using a test oracle), which can be used to find a set of valid configurations. The following coverage criteria are named: Simple (test each value of a variable), pair-wise (test each two-wise combination of variable values), dependency-based (reduce effort through constraints on variable values) and compound (composition of all). (2) Additionally, an operational validation method is proposed that also deals with context changes/transitions. Therefore, DiVA uses Multi-dimensional Covering Arrays (MDCA) including a temporal dimension. These describe multiple context instances that are scheduled as test sequences and provide a means for the definition of coverage criteria on sequences of adaptations. There are also fitness functions that help to minimize the test cases while sustaining a good coverage. While DiVA contributes many ideas for testing, it still does not consider interactions with the application's control flow.

Besides approaches from the DSPL and SAS research field, other work focuses on context-awareness and test data generation from context models [12][13]. For instance, Wang et al. construct in [13] control flow graphs of context changes and associate them by using point of high impact (Context-Aware Program Points, CAPPS) with the core control flow. They also provide three context-adequacy criteria. However, the proposed models of this approach are not extensive enough to express the behavior such that additional test coding is required.

III. Failure Analysis

In this section, we analyze relevant failure characteristics and scenarios. For this purpose, we use *Failure Mode and Effects Analysis* (FMEA) [7]. FMEA is used in engineering of safety-critical systems to find relevant failure sources. The method was first applied for electrical and mechanical systems and later extended for the usage in software engineering [14][15]. Based on these experiences, our analysis is separated into three steps:

1) identification of SAS-specific failure dimensions and properties (presented as *Failure Domain Model*)
2) investigation of SAS-specific failure scenarios
3) visualization of error propagation among the found scenarios as *Fault Dependency Graph*

Step (3) is not an actual part of FMEA, but usually a *Fault Tree analysis* (FTA) [16] is performed to visualize the scenarios' dependencies. As our system runs a control *loop*, we customize the analysis process in this step by constructing a fault dependency graph instead.

A. A Common Process of Self-adaptation: MAPE-K

Before starting the analysis, a level of detail has to be specified to have a fix abstraction perspective on SAS and a well-defined system boundary. FMEA is designed to be run against an existing architecture, which we cannot assume to be widely similar in all existing or future developed SAS. Hence, we leave the strict understanding of FMEA by analyzing the MAPE-K process as common concept of minimal necessary data flows with the means of this investigation method. As seen in the previous section, there are several intersecting research directions coping with self-adaptivity. They have in common that the process of information gathering and utilization can be described in a known schema, mostly referred to as the MAPE-K (Monitor/Analyze/Plan/Execute–Knowledge) [1] control loop.

As illustrated in Figure 1, this process consists of four tasks to be fulfilled by any self-adaptive system framework. The system *monitors* a certain data source such as a *sensor*. The captured information is then forwarded to the *analysis* part where the system reasons about the necessity of adaptation. After these first two process tasks the system is able to determine *if* an adaption should be performed.

In the subsequent *plan* section, an adaptation plan is generated and later applied in the following *execute* task. *Effectors* may also manipulate external entities. The control

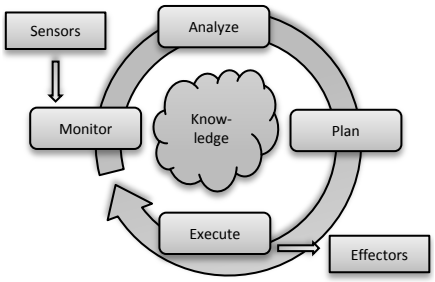

Fig. 1. MAPE-K control loop (cf. [1]).

cycle is re-run from this point periodically such that a self-adaptive system always has an optimal state (according to the supposed utility or goal function, and available knowledge) to fulfill its task. Analysis, planning, and execution interact through data organized as a central *knowledge* model.

MAPE-K encompasses the adaptation according to environment changes such as context, user input and system utilization. The sources of processed information can be abstracted by leaving out the concrete objects sensors and effectors work on. Hence, we use MAPE-K as common view point on SAS architectures.

B. Step 1) Failure Domain Model (FDM)

In this section, we provide a set of properties that failures in adaptive systems can comprise. As there are several classes of adaptive software [17], we cannot assume that each property is reasonable in every concrete system. Furthermore, we exclude failures, which originate in sensors and effectors to create a fixed boundary around the software's scope.

Each of an SAS' components provides a service (i.e., a perceivable behavior) through its interface. In the system's specification an expected behavior is defined. According to [4], a *failure* is an event of service deviation from this expectation. An *error* is the inconsistent part of the total system state (internal state plus perceivable external state) which lead to this failure. The cause of an error is a *fault*. Error propagation occurs if a failure causes a fault in another component.

The result is the fault-error-failure causal chain as presented in the FDM in Figure 2. Each concept has up to three variable dimensions. Although, in [4] several dimensions are listed, here we limit our investigation on those which are relevant to a development-independent tester who is not in charge of repairing the system. In consequence, concerning faults, the only relevant property is their *persistence* which may either be *permanent* or *transient*. For instance, intent or objective play no role when testers uncover and report defects. While persistence is a rather general dimension, this classification has an extended significance due to the cyclic nature of SAS. The source of a fault can be in one of the control loop's tasks and, additionally, originate in the system's knowledge. The latter case produces a cyclic failure propagation: If a failure manifests in the system's knowledge model, it may have harmful influence on future decisions such that the failure becomes a fault in following cycles.

For errors we propose the dimensions type and localization. Types may be one of three: The caused error either manifests

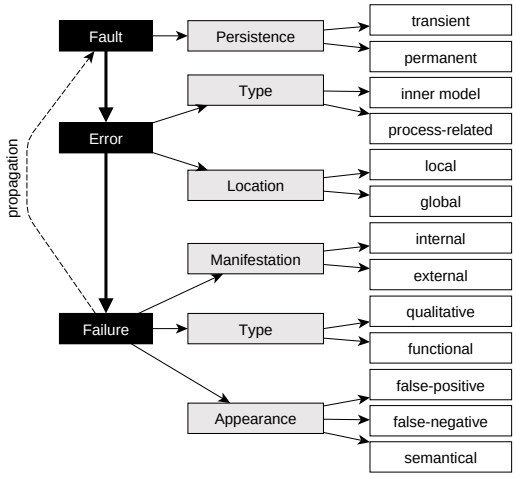

Fig. 2. Failure domain model.

as an inconsistency in (1) the knowledge representation of the perceived environment does not reflect its physical nature, (2) the model itself (e.g., it violates the model's constraints), or (3) as incorrect intermediate state in the computation process (process-related). Furthermore, errors can localize either locally or globally in the potentially distributed SAS.

Concerning failures, [4] distinguishes between content and timing (early/late) correctness. In contrast, an SAS' goal definition may aim on other non-functional properties like energy-usage. Thus, we alter the original distinction to qualitative (the system performs non-optimal) or functional (incorrect system behavior). Furthermore, failures in SAS can either manifest internally or externally (e.g., by using the effectors). The last dimension comprises the event-driven nature of the system. The sensed and analyzed information may lead to un-intentional (false-positive), missed (false-negative), or semantically wrong adaptation attempts.

This FDM is a valuable source when classifying failures in concrete adaptive systems. It describes abstract properties of failures that can be instanciated for real world systems. Verdicts (the classification of test results, for instance `Pass`, `Fail` or `Inconclusive`) can be parameterized by these information.

C. Step 2) Failure Scenarios

After defining all dimensions of failures, it is required to identify scenarios of failure occurrence in adaptive software. We cannot give a general method of priorization, as usually done in FMEA. For testing one of these scenarios, this strongly depends on domain specific conditions. As only general valuation standard, *criticality* can serve. In this case, according to [17], each adaptation operation can be either harmless, mission-critical, or safety-critical.

We decompose the MAPE-K control loop in five components: `Monitor`, `Analyzer`, `Executor`, `Planner` and `Scheduler`. The latter two are separated to enable the consideration of *interaction* between adaptation and running processes. After the `Analyzer` found *that* the system has to be adapted, the `Planner` decides *how* the adaptation is

processed. The Scheduler has the task to fill an Action queue (however it may be implemented in concrete systems) by arranging system process actions with adaptation actions. An adaptive system designer has to be aware of how he maintains consistency either through an actual implemented scheduler component or a transaction-like behavior. This issue also breaks the straight MAPE-K data flow because a scheduler requires information about the system actions (retrieved from the Executor) and composes them with adaptation intents.

All components are considered as black boxes. Components are connected by data flow edges (black arrows). Additionally, the process contains Sensors, Effectors and the central knowledge Models. The latter one contains information about the system structure adaptation logic and further knowledge relevant for adaptation decisions. Sensors and Effectors communicate with the external world (e.g., other systems or the physical reality).

Based on this structure, we list our found failure scenarios in Table I. It comprises *Failure Identifier* (FID), *Component Identifier* (CID), Fault, Error, Failure, and a *Propagation* column. All FIDs can be found in the architecture visualization in Figure 3 as well. In the following, each scenario is described in detail.

SENS: The first scenario comprises test input received from the Sensors and misinterpreted by the Monitor component (i.e., it does not produce corresponding events or produces events without being indicated by sensor inputs). For instance, a context reasoner could output an physically causeless temperature change due to a poorly designed interference rule.

TRIG: An event produced by Monitor does not or unintentionally lead to a corresponding adaptation initiation or to a wrong one. Unintended initiations let the situation *appear* (cf. appearance dimension of failures in the FDM, Figure 2) as if an adaptation is necessary. TRIG may also be caused by a propagated failure from SENS or EVENT in integration testing. Here an example is a wrong implemented adaptation rule condition.

PRE: Though the set of operated Models contains information which (according to the adaptation logic) either leads to a specific adaptation or prohibits one, the Analyzer decides differently. At this point, potential failures in the adaptation logic itself have to be considered—either due to a wrong specification or adaptation (cf. POST scenario below). For instance, consider a recorded sound level which is taken into account while reasoning about rising a sound output of the system itself. If the recorded information is erroneous, non-intended adaptions may be initiated. Such failures can also be observed while testing the Analyzer.

Both TRIG and PRE scenarios may interact, because we did not decompose the analyzer in more detailed components. Hence, these scenarios have to be tested together as both data sources are required for each test case and just a probabilistic estimation can be stated which one is actually defective.

ADAPT: The adaptation initiation may again be of false-positive, wrong or missing appearance. Like TRIG and PRE, this scenario relies on the assumption that the adaptation logic is correct. Such an scenario may occur if the adaptation reasoning mechanism misses to execute a rule's adaptation directive.

PLAN: The Analyzer determines *if* an adaptation is required but *not how* to perform it. This task is operated in the planning phase. A Planner reasons over the variability and the current system state. Its output have to be a correct adaptation plan that can be applied in the system and leads to a consistent state. The PLAN scenario encompasses that the compiled plan is incorrect, e.g., its order.

SCHED: Reconfiguration actions potentially interact with the system's control flow. Such problems arise because variability cannot be completely orthogonal to the system's inner behavior. For instance, if the system is a database, there could be a potential conflict when adapting to a situation where a speedup is required by deactivating the transaction feature in exactly the same time period when running a transaction. We get an active SCHED scenario if the compiled action sequence of the Scheduler is inconsistent.

The Executor is a complex interpretation engine that produces multiple outputs and thus, has multiple potential failure scenarios. All Executor-related scenarios may also be the outcome of a propagated SCHED failure.

RECONF: The reconfiguration may run into a failure itself. If any reconfiguration mechanism fails without being recognized, the actual system structure is out of synchronization with its model representation. Here failures can be constituted that can be of both types: functional or qualitative (cf. FDM).

POST: On the other hand, the Model's part that represents the reconfigured systems may be inconsistent after the execution because a model manipulation was performed erroneously by the the Executor. For, instance if for further adaptation decisions knowledge about past ones is required, a missed recording causes problems. POST can have complex consequences because the manipulated model is assumed to be correct in PRE and PLAN. Hence, POST may propagate faults to these two scenarios. Additionally, this scenario can also be a "starting point" failure because its semantics also resemble a defective Model at the system's start-up.

EFFECT: Another output of the Executor can be actions that have to be run in external systems by physical Effectors. If actions are not generated correctly and forwarded to the effectors (e.g., due to corrupt drivers), the representation of these externals lose synchronization with potential internal model representations. As the Sensors may perceive data from the manipulated system, we produce a possible propagation of this failure to the SENS scenario.

EVENT: The last failure scenario is related to events that are produced in the system (e.g., user interactions) and are propagated to the Analyzer component. In this case, the generated events can be erroneous.

D. Step 3) Failure Dependency Graph

As final FMEA artifact we construct a failure dependency graph as depicted in Figure 4. The visualization illustrates the potential cyclic failure propagation through inner system events, model manipulation, or physical Effectors/Sensors correlations (the latter one is visualized by the dashed edge). Furthermore the PRE and TRIG scenarios may influence each other in both directions, which makes them hard to test in isolation.

 ISBN: 978-1-61208-274-5

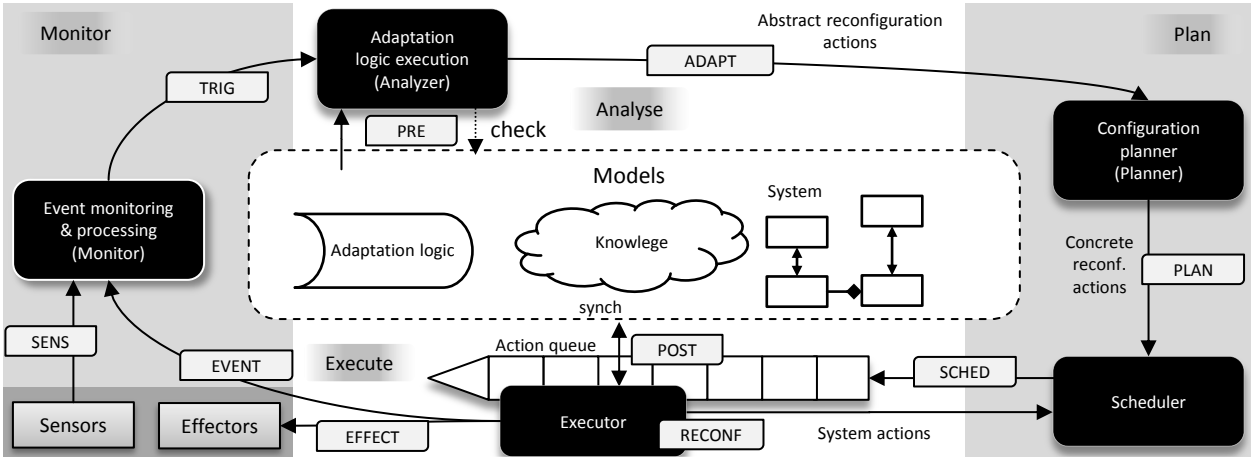

Fig. 3. Conceptional architecture of an adaptive system.

TABLE I. ADAPTIVE SOFTWARE'S FAILURE SCENARIOS.

FID	CID	Fault	Error	Failure	Propagation
SENS	Monitor	corrupt sensor interpreter	misinterpreted sensor data	no/wrong/unintended event	TRIG
TRIG	Analyzer	corrupt event interpreter	misinterpreted event	no/wrong/unintended adapt.	PRE—ADAPT
PRE	Analyzer	corrupt model interpreter	misinterpreted model	no/wrong/unintended adapt.	TRIG—ADAPT
ADAPT	Analyzer	corrupt reasoning	wrong adaptation derived	no/wrong/unintended adapt.	PLAN
PLAN	Planer	corrupt planner	inconsistent planning	inconsistent plan	SCHED
SCHED	Scheduler	corrupt scheduler	inconsistent scheduling	wrong order of actions	POST—EVENT—EFFECT—RECONF
POST	Executor	corrupt model manipulator	corrupt model construction	model inconsistent	PRE—PLAN
RECONF	Executor	corrupt configurator	reconfiguration fails	configuration↔model unsynch	–
EVENT	Monitor	corrupt event producer	wrong event production	no/wrong/unintended event	TRIG
EFFECT	Executor	corrupt forwarding	processing wrong effector operations	model↔externals unsynched	(SENS)

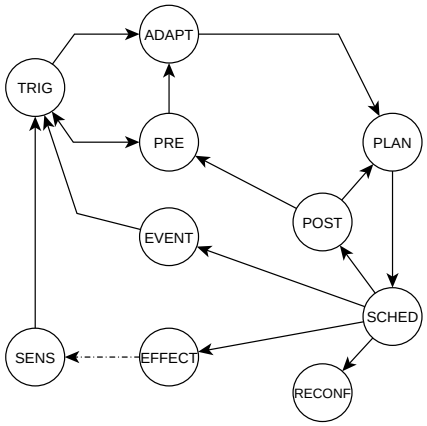

Fig. 4. Interdependencies of failure scenarios.

IV. REQUIREMENTS TO MODELS FOR SAS TESTING

All following requirements for adaptation correctness test methods are based on one or more of the presented failure scenarios. In the following, we list these mapped requirements and give each one a name for later reference. The requirements are formulated as *assurance tasks* that have to be fulfilled by employing MBT methods.

1) **Correct sensor interpretation:** Assure that the sensor data is correctly interpreted and transformed into system events. Potential sensor data has to be specified together with context identifications.(↦SENS)
2) **Correct adaptation initiation:** Assure that events initiate the correct adaptation if all preconditions in the model hold. Events, conditions, and adaptation decisions (goals) have to be associated in the models.(↦TRIG/PRE/ADAPT)
3) **Correct adaptation planning:** Assure that the generated adaptation plan is consistent w.r.t. target configuration and action order. Build a model to map adaptation goals to possible plans.(↦PLAN)
4) **Consistent interaction between adaptation and system behavior:** Assure that the generated adaptation plan is correctly scheduled with the systems' control flow. A model is required to define which adaptation is allowed in which state of application control.(↦SCHED)
5) **Consistent adaptation execution:** Assure that (1) the generated adaptation schedule is applied to system structure and (2) the synchronization between system and models is consistent after adaptation. Thus, we need a set of assertions to be checked after the adaptation execution.(↦POST/RECONF)
6) **Correct system behavior:** Assure that the system correctly commits events or actions to the effectors. As in the previous requirement, here we need to specify events to be observed in the system when running any operation.(↦EVENT/EFFECT)

 ISBN: 978-1-61208-274-5

Additionally, in testing, coverage criteria are required to restrict the combinatorial search space of the system under test and, nevertheless, have a reasonable and meaningful test result. For less-complex systems, many criteria for test coverage were found. Mostly, they refer to a graph representation like a state machine. Known criteria are statement, branch or path coverage. However, as we have seen, there is a complex set of requirements and aspects to be tested in the context of SAS. In consequence, we have to use multiple models which are more expressive then state machines (as assumed in the mentioned coverage criteria) to represent all testable aspects. In consequence, the known criteria cannot be applied directly. Hence, the last requirement is to find a set of proper coverage criteria for adaptation mechanisms which can be composed:

7. **Adaptive coverage criteria:** Find constructive coverage criteria metamodels/languages to describe *which*, *when* (in relation to system behavior), and *in which order* adaptation scenarios have to be tested and analytic coverage criteria to measure how adequate a test suite is.

V. Conclusion and Future Work

In this paper, we applied a customized Failure Mode and Effects Analysis (FMEA) to a conceptional self-adaptive software system based on the minimal structural assumptions of MAPE-K. We derived a failure domain model to provide a system in which faults, errors and failures can be classified. Subsequently, we derived ten distinct failure scenarios that occur in the process of adaptation. By building a fault dependency graph we visualized potential cyclic propagation of failures in such systems. In consequence, six *founded* modeling requirements were stated that all can be mapped to one or more of the described failure scenarios. A seventh requirement is established by the coverage question. Based on these foundations a systematic analysis of SAS is possible comprising failure properties, occurrence, and propagation. A well-designed MBT framework is comprehensive if all presented requirements are fulfilled and the all respective assurances are considered.

For further investigation, it is necessary to instantiate the found requirements for a real-world adaptive software infrastructure. If we can map this implementation to several adaptivity frameworks and express the majority of necessary test cases, our approach can be attested substantial and generic. Despite our work on mobile software testing, we recently started several research projects coping with adaptivity, namely SMAGS[18] and VICCI[19]. SMAGS (Smart Application Grids) proposes a role-based architecture, which is able of adaptive composition. VICCI searches for approaches to build adaptive Cyber-physical Systems (CPS) like Smart Homes or robots to improve our daily live in an intelligent manner. Both projects are possible test targets to our MBT strategy. Furthermore, from our experience in software testing, we should thereby sustain usability of used modeling concepts despite the complexity of the process. Especially in adaptive software this task is crucial due to the potentially huge complexity and variability in configurability and behavior.

Acknowledgments

This research has received funding within the project #100084131 by the European Social Fund (ESF) and the German Federal State of Saxony, by Deutsche Forschungsgemeinschaft (DFG) within CRC 912 (HAEC) as well as T-Systems Multimedia Solutions GmbH.

References

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

[2] T. M. King, D. Babich, J. Alava, P. J. Clarke, and R. Stevens, "Towards self-testing in autonomic computing systems," in Proceedings of the Eighth International Symposium on Autonomous Decentralized Systems, ser. ISADS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 51–58.

[3] B. H. C. Cheng, D. Lemos, H. Giese, P. Inverardi, and J. M. et al., "Software engineering for self-adaptive systems: A research roadmap," in Dagstuhl Seminar 08031 on Software Engineering for Self-Adaptive Systems, 2008.

[4] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," Dependable and Secure Computing, IEEE Transactions on, vol. 1, no. 1, 2004, pp. 11–33.

[5] M. Utting, Practical model-based testing: a tools approach. Morgan Kaufmann, 2007.

[6] G. Püschel, R. Seiger, and T. Schlegel, "Test Modeling for Context-aware Ubiquitous Applications with Feature Petri Nets," in Modiquitous workshop, 2012.

[7] H. E. Roland and B. Moriarty, System Safety Egnineering and Managemnent, 2nd edn. John Wiley & Sons, Chichester, 1990, ch. Failure Mode and Effect Analysis.

[8] S. Hallsteinsen, M. Hichey, S. Park, and K. Schmid, "Dynamic software product lines," IEEE Computer, 2008, pp. 93–95.

[9] F. Munoz and B. Baudry, "Artificial table testing dynamically adaptive systems," 2009.

[10] V. Dehlen and A. Solberg, "DiVA methodology (DiVA deliverable D2.3)," 2010.

[11] A. Maaß, D. Beucho, and A. Solberg, "Adaptation model and validation framework final version (DiVA deliverable D4.3)," 2010.

[12] T. Tse, S. Yau, W. Chan, H. Lu, and T. Chen, "Testing context-sensitive middleware-based software applications," 28th Annual International Computer Software and Applications Conference, 2004, pp. 458–466.

[13] Z. Wang, S. Elbaum, and D. S. Rosenblum, "Automated generation of context-aware tests," 29th International Conference on Software Engineering (ICSE), 2007, pp. 406–415.

[14] H. Sozer, B. Tekinerdogan, and M. Aksit, Archtitecting dependable systems IV. Springer, 2007, ch. Extending failure models and effects analysis approach for reliability analysis at the software architecture design level.

[15] B. Tekinerdogan, H. Sozer, and M. Aksit, "Software architecture reliability analysis using failure scenarios," Journal of Systems and Software, vol. 81 (4), 2008, pp. 558–575.

[16] J. Dugan, Handbook on Software Reliability Engineering. McGraw-Hill, New York, 1996, ch. 15. Software System Analysis Using Fault Trees, pp. 615–659.

[17] J. Andersson, R. Lemos, S. Malek, and D. Weyns, "Software engineering for self-adaptive systems," B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, ch. Modeling Dimensions of Self-Adaptive Software Systems, pp. 27–47.

[18] C. Piechnick, S. Richly, S. Götz, C. Wilke, and U. Aßmann, "Using Role-Based Composition to Support Unanticipated, Dynamic Adaptation – Smart Application Grids," in Proceedings of ADAPTIVE 2012, The Fourth International Conference on Adaptive and Self-adaptive Systems and Applications, 2012, pp. 93–102.

[19] D. B. Martin Franke and T. Schlegel, "Towards a flexible control center for cyber-physical systems," in Modiquitous workshop, 2012.

 ISBN: 978-1-61208-274-5